A Property-based Testing Framework for Multi-Agent Systems

Extended Abstract

Clara Benac Earle
ETSIINF, Universidad Politécnica de Madrid
cbenac@fi.upm.es

Lars-Åke Fredlund
ETSIINF, Universidad Politécnica de Madrid
lfredlund@fi.upm.es

ABSTRACT

In this article we describe a framework that we have developed for testing multi-agent systems written in the Jason agent programming language, using the testing technique known as property-based testing, a form of randomised automatic model-based testing.

KEYWORDS

Testing; multi-agent systems; Jason

ACM Reference Format:
Clara Benac Earle and Lars-Åke Fredlund. 2019. A Property-based Testing Framework for Multi-Agent Systems. In *Proc. of the 18th International Conference on Autonom ous Agents and Multiagent Systems (AAMAS 2019), Montreal, Canada, May 13–17, 2019,* IFAAMAS, 3 pages.

1 RELATED WORK

The testing of aspects of multi-agent system (MAS) has received substantial attention in the past; for a thorough survey see [5]. Many of the works on testing multi-agent systems focus on unit testing. An example is [7], where a model-based approach is used to test basic units such as events, plans and beliefs. Another body of works, such as [4], test agents goals as the smallest testable units in MAS. In [6], a model-based oracle generation method for unit testing belief-desire-intention agents is discussed.

2 PROPERTY-BASED TESTING

Property-based testing (PBT) [3] is a testing methodology which focuses on generating, automatically, test cases from a more abstract description of the behaviour of the system under test (the *property*). That is, property-based testing can be considered a form of model-based testing. In this article the testing tool used is Quviq QuickCheck (henceforth simply QuickCheck), which uses the Erlang programming language [1] to specify models of behaviour.

The basic functionality of QuickCheck is rather simple: to check e.g. a Java method without side effects we specify *generators* for the method arguments. A generator is capable of generating an infinite number of values of some data type, according to a probability distribution. Given generators of method parameters, the test *property* examines the result of applying the method-under-test to the generated argument, and judges if the result value (or launched exception) is correct given the input parameters. Thus, given a set of generators for input data, and a test property, QuickCheck generates a random instantiation of the variables, executes the method under test, and checks that the resulting boolean property is true.

Proc. of the 18th International Conference on Autonom ous Agents and Multiagent Systems (AAMAS 2019), N. Agmon, M. E. Taylor, E. Elkind, M. Veloso (eds.), May 13–17, 2019, Montreal, Canada.

For checking reactive code, e.g., a multi-agent system, a test case is not a simple call to a method, but rather a sequence of method calls (or in the case of multi-agent systems, communications, belief updates, etc). To generate such test cases, and to judge whether the execution of a test case is correct, QuickCheck provides a library which codes generators as state machines, and reuses the same state machine for judging the correctness of an execution.

Normally QuickCheck computes test cases before testing begins. However, for testing non-deterministic software (e.g., most multi-agent systems) this can lead to generating of a large number of "uninteresting" test cases. Instead, for checking such non-deterministic software, QuickCheck can interleave the generation of a test case with its execution, such that after a test case command is derived, the command is executed, and the next state of the state machine is computed using the *actual result* of executing the command.

3 TESTING A MULTI-AGENT SYSTEM

Our approach is to replace a subset of the agents in the multi-agent system with a QuickCheck state machine. The QuickCheck state machine plays the role of these agents, interacting with the remaining real agents by sending messages and modifying the environment, and judging whether these remaining real agents are correctly implemented by examining the messages sent to any replaced agent, and the belief perceptions that they receive.

3.1 Example: a domestic beer serving robot

Our approach to testing multi-agent systems will be illustrated by focusing on the "domestic robot" multi-agent system, which is present in the the Jason distribution, and is also explained in [2]. Basically, a domestic robot continuously serves beer to an owner. To serve a beer a robot should go to the fridge, take out a bottle of beer, and bring it back to the owner. However, the fridge may run out of beer, and thus the robot should remember to restock it by interacting with a "supermarket". Moreover, the robot agent protects the owner from getting too drunk: if the number of beers handed over to the owner exceeds some limit during a day, no more beers should be provided to the owner.

This is a simple multi-agent system, comprised of just three agents: the owner, the robot, and the supermarket agents. However, the code implementing the multi-agent system environment should also be considered part of the system. The environment implements a number of environmental actions, including the owner sipping from a beer (bottle): "sip(beer)" and the robot handing a new beer to the owner: "hand_in(beer)". Other environmental actions include opening and closing the fridge, and so on. In total the environmental handling code comprises around 230 lines of Java code, while the code for the agents comprises around 140 lines of Jason code.

Which agents should be modelled by a QuickCheck state machine? The obvious choice is to replace the owner agent by a QuickCheck state machine. This entails testing that the combination of the robot agent, and the supermarket, provides a correct service to the owner. For example, that beers are eventually provided to an owner which asks for them, unless the owner has already received too many.

4 EXPERIMENTS

As an experiment we tested both the original Jason code for the robot and a second, more dynamic, variant.

Checking the original code. The first state machine for the domestic robot example has a single extended state $owner(NumSips, NumHandedIn)$. The $NumSips$ parameter details how many sips remain until all beers have been consumed, while $NumHandedIn$ counts the number of beers that have been handed in to the owner *during the present day*. An example transition is the following one:

$$\begin{array}{l} sip(beer)\ \textbf{when}\ NumSips = 0 \implies \\ \quad \textbf{obs} = -!envAction(sip(beer)) \supset \\ \quad\quad \textbf{next-state} = owner(NumSips, NumHandedIn) \end{array}$$

The meaning of the transition is as follows. For a transition to be executed by the state machine the condition predicate ($NumSips = 0$) has to hold in the current state. When the transition is executed the corresponding action $sip(beer)$ is issued, and the observable results **obs** (i.e., the communications sent to the agents represented by the state machine, and the new beliefs, etc) are collected. The transition is deemed *correct* if exactly $-!envAction(sip(beer))$ was observed, and otherwise testing *fails*. For this transition the state does not change. This captures the intuition that, if the owner attempts to sip a beer when there is no beer left, the observable observation should be the negative achievement action event $envAction(sip(beer))$.

Testing using QuickCheck revealed two errors in the code: (i) when the owner had already asked for a beer, and asked for more beers, the number of sips left was incorrectly calculated; the consequence was that the environmental action "sip(beer)" failed too soon. Note that this is an error not in a Jason agent itself, but rather in the environmental Java code; (ii) the robot agent permitted the owner to ask for one additional beer when the limit was already reached, due to using the '$>$' operator instead of '$\geq$'.

Checking a more dynamic domestic robot. The domestic robot example considered so far is admittedly quite unrealistic as a model of a MAS as the principal agents tested, the robot and the fridge, act deterministically. To experiment with a more realistic MAS we developed a more "caring" robot which asks, non-deterministically, the owner whether he/she is well. The robot then refrains from providing beers to the owner until the owner gives a positive answer. To test this new behaviour (the original state machine quickly signals an error) we (i) modify all transitions to accept asynchronous queries by the robot, checkpointing the existing testing state, and (ii) include a new transition which replies to a robot query, and then recovers the existing testing state. With these changes the resulting state machine finds no errors in the modified domestic robot example. A second change to introduce more non-determinism is to let the robot decide unilaterally to modify the limit of the number of beers permitted per day; such behaviour can easily be handled in our testing framework through belief inspection.

5 FRAMEWORK DESIGN

The central function of the library is to permit the QuickCheck state machine, which runs in an Erlang runtime system, to act as a subset of Jason agents $ag_1, \ldots, ag_n$; we say that the state machine *handles* an agent ag_i. This entails that the state machine must, on behalf of each handled agent ag_i, be able to communicate with other Jason agents, and to issue environmental actions. Moreover, the QuickCheck state machine has to be informed of any observable event that occurs in any Jason agent ag_i it handles.

To accomplish this, each agent handled by the state machine is instantiated as a generic agent from the Jason agent code *proxy*. Such a proxy agent receives events from the MAS, and simply forwards them to the state machine (the state machines keeps an ordered queue of events per simulated agent) using the new internal action fromProxy. The queues of simulated agents are read by the QuickCheck state machine after invoking an action, to determine what the observable results of invoking the action are. If the state machine wishes to communicate with another agent on behalf of one of its simulated agents, or update the environment, it forwards a request to the Jason agent proxy, which realises it.

Handling Asynchronous Results. So far we have assumed that the observable results of actions by the state machine (new beliefs, etc) can be deterministically returned to the state machine. Unfortunately this is very unrealistic: the library simply cannot know how long to wait for an observable event which was caused by an action by a handled agent; maybe the MAS is simply very slow.

To handle such situations the following strategy is used: if the action is expected to cause observable results, the QuickCheck state machine tries to read, from the queue of the invoking agent, at least as many observable events as are expected. A timer is set, which releases after a relatively long interval, if enough observable events are not received (leading the state machine to signal a testing error). If, on the other hand, no observable event is expected as a result of invoking the action, the state machine only waits a short interval of time. If, contrary to expectations, an observable action is received, a testing error is signalled. Waiting only a short interval risks missing late erroneous observable results. However, this is not catastrophic in the sense that such observable results will be reported instead for the next action by the state machine, which likely does not expect them either (and will thus signal a testing error, which will be misreported to have been caused by a later action). In a sense this is unavoidable: due to the autonomous nature of agents we cannot be sure to always identify errors precisely unless we wait indefinitely (or a very long time), and this is contrary to the idea of property testing where we aim to run a large number of tests.

The library provides a number of additional functionalities such as the possibility to inspect the belief base of agents. This facility is useful in situations when agents do not announce decisions publicly, but rather record decisions in their belief base.

The resulting library, excluding QuickCheck, will be released as open source.

ACKNOWLEDGMENTS

This work has been partially supported by the Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union.

REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. 1996. *Concurrent Programming in Erlang*. Prentice-Hall.

[2] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.

[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[4] Erdem Eser Ekinci, Ali Murat Tiryaki, Övünç Çetin, and Oguz Dikenelli. 2008. Goal-Oriented Agent Testing Revisited. In *Agent-Oriented Software Engineering IX, 9th International Workshop, AOSE 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers*, Michael Luck and Jorge J. Gómez-Sanz (Eds.). 173–186. https://doi.org/10.1007/978-3-642-01338-6_13

[5] Cu D. Nguyen, Anna Perini, Carole Bernon, Juan Pavón, and John Thangarajah. 2009. Testing in Multi-Agent Systems. In *Agent-Oriented Software Engineering X - 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers*. Springer, 180–190. https://doi.org/10.1007/978-3-642-19208-1_13

[6] L. Padgham, Zhiyong Zhang, J. Thangarajah, and T. Miller. 2013. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. *IEEE Transactions on Software Engineering* 39, 9 (Sept. 2013), 1230–1244. https://doi.org/10.1109/TSE.2013.10

[7] Zhiyong Zhang, John Thangarajah, and Lin Padgham. 2009. Model based testing for agent systems. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*, Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman (Eds.). 1333–1334. https://doi.org/10.1145/1558109.1558280