Seamless Integration and Testing for MAS Engineering

Mostafa Mohajeri Parizi[1], Giovanni Sileno[1], and Tom van Engers[1]

Informatics Institute, University of Amsterdam, Amsterdam, the Netherlands
{m.mohajeriparizi,g.sileno,t.m.vanengers}@uva.nl

Abstract. Testing undeniably plays a central role in the daily practice of software engineering, and this explains why better and more efficient libraries and services are continuously made available to developers and designers. Could the MAS developers community similarly benefit from utilizing state-of-the-art testing approaches? The paper investigates the possibility of bringing modern software testing tools as those used in mainstream software engineering into multi-agent systems engineering. Our contribution explores and illustrates, by means of a concrete example, the possible interactions between the agent-based programming framework ASC2 (AgentScript Cross-Compiler) and various testing approaches (unit/agent testing, integration/system testing, continuous integration) and elaborate on how the design choices of ASC2 enable these interactions.

Keywords: Multi-Agent Systems · Multi-Agent Systems Engineering · Testing · Continues Integration.

1 Introduction

Software testing is attracting increased interest in industry [1] and it is one of the most used methods of software verification. One of the reasons of this success lies in the advancement and popularization in the software engineering community of methodologies commonly known as *DevOps*, in particular of techniques of automated testing in *continuous integration* (CI). Generally, CI refers to the facilitation provided by third-party tools for automating the build/test process of a software. In recent years, online DevOps services such as TravisCI[1] and CircleCI[2] have been increasingly used by software engineers to improve the efficiency of their testing process, a practice which plausibly resulted in increased quality of the developed software.

Very recently, Fisher et al. [18] have suggested that testing approaches would be an important complement to formal approaches to MAS verification, if they

[1] https://travis-ci.com/
[2] https://circleci.com/

could be automated and integrated in a seamless way into MAS development. In our view, seamless integration does not mean only that agent programmers are able to use the vast amount of software testing tools available to mainstream languages like Java or Python, but, more importantly, that they are also able to use (almost) language- and framework- agnostic online services as those used for CI. This paper explores this idea, aiming to illustrate what the MAS community could gain by using industry standard testing tools and discussing what would be the theoretical and practical trade-offs for this choice. We investigate possible interactions of testing with agent-based programming, and its relation with other verification techniques. More concretely, we demonstrate various approaches to enhance the productivity of MAS development cycle in the AgentScript Cross-Compiler (ASC2) framework [27] via mainstream software testing and integration tools, and elaborate on the design choices of ASC2 that affect the testability of agent-programs with the mentioned tools. Then, we explore on how this approach can be generalized for other MAS frameworks.

The motivation for this work arises from research conducted on data-sharing infrastructures (e.g. data marketplaces). At functional level, a data-sharing application corresponds to a coordination of several computational actors distributed over multi-domain networks. Those actors generally include certifiers, auditors, and other actors having monitoring and enforcement roles, ensuring some level of security and trustworthiness on data processing [42]. Typically distributed across several jurisdictions, networks may be subjected to distinct norms and policies, to be added to various infrastructural policies provided at domain level and *ad-hoc* policies set up by the users. Some of these norms, as for instance the GDPR, bind processing to conditions and specific purposes, but, more in general, all compliance checking on social systems requires to know and to infer (in case of a failure on expectations) *why* an actor is performing certain operations. Agent-based programming, and particularly the Belief-Desire-Intention (BDI) model [35], by looking at computational agents as *intentional agents*, provides the "purpose" level of abstraction available by design, and for this reason it is a natural technological candidate for this application domain.

The BDI model been extensively investigated as basis to represent computational agents that exhibit rational behaviour [19] and multiple programming languages and frameworks have been introduced based on it, as AgentSpeak(L)/Jason [34, 6], 3APL/2APL [11], and GOAL [22]. Recent works as e.g. [23, 27] investigated various issues holding when mapping logic-oriented agent-based programs into an operational setting. In contrast, this paper focuses instead on the *development practice* aspect: as soon as we attempted to program data-sharing applications as agents, we experienced the lack of mature software engineering toolboxes, thus hindering a continuous integration with the infrastructural-level components developed in parallel by our colleagues.

The document proceeds as follows: section 2 provides a background and related works on verification of MAS, in section 3 we introduce our approach on MAS testing in ASC2 framework with mainstream tools. An illustrative exam-

ple of this approach is presented in section 4. Finally, section 5 provides the discussion and comments on possible extensions and future developments.

2 Verification of (Multi-)Agent Systems

Verification is a crucial phase in any software (and system) development process, and as such it has been addressed also by the Multi-Agent Systems (MAS) community. The survey presented in [2] provides an empirical review of over 230 works related to verification of MAS.

At higher level, approaches for the verification of autonomous systems fall into five categories [18]: (a) *model checking*, (b) *theorem proving*, (c) *static analysis*, (d) *run-time verification*, and (e) *(systematic) testing*. While the first four approaches (a-d) are considered formal or at least semi-formal, testing (e) is deemed to be an informal approach to verification. Further, MAS verification can be targeted at different levels, varying from fine-grained verification of agents at a logical level [3] to verification of emergent properties in a system [12]. Ferber [16] identifies three levels: (i) *Agent level* considers internal mechanisms and reasoning of an agent (ii) *Group level* consists in testing coordination mechanisms and interaction protocols of agents, and (iii) *Society level* checks for emergent properties or if certain rules and/or norms are complied within the society. In general, the choice of a verification method depends on the required level of verification, as e.g. formal methods may not be applicable for the verification of a large MAS with non-deterministic characteristics at the society level.

Most of the works on MAS verification point out that testing agent programs is far harder than testing normal software, on the grounds that agents tend to have more complex behaviors, and deal with highly dynamic and often non-deterministic environments (including other agents), on which they have only partial control [30]. A series of recent empirical results [38, 37] was used to conclude that, with respect to certain distinct test criteria, testing BDI agents can be practically infeasible. The *all-paths* criterion requires the test suite to cover all the paths of the agent's goal-plan graph; its application shows that the number of tests needed to run is intractable [38]. In subsequent work, the same authors study the minimal criterion of *all-edges*, requiring all edges of the goal-plan graph to be covered. While not *per se* infeasible, results show that even this criterion requires a (too) high number of tests [37].

These observations can explain why much of the work in verification of autonomous systems and specifically of BDI agents have been towards the *formal verification* of agent programs, a mathematical process for proving that the system under verification matches the specification given in formal logic [4]. One of the most successful formal methods for verification of software agents has been *model checking* [9]. Model checking of BDI agents can be done as e.g. in [5] by translating a simplified version of AgentSpeak(L) to Java programs and using the Java Path Finder (JPF) verification tool. Probably the most notable works that adopt a (semi-)formal model checking approach are those of the AJPF/MCAPL framework [13, 17]; AJPF/MCAPL also relies on JPF to perform program model

checking on agent programs developed in multiple JVM-based BDI frameworks by utilizing an implementation of the target language's interpreter. Nevertheless, although formal verification techniques as model-checking provide a high level of guarantee, they are typically both complex and slow to deploy [39].

A number of approaches to testing (that is, *informal verification*) have also been considered in the MAS literature. Some of those utilize model-based testing [33, 41] and rely on *design artifacts* such as Prometheus design diagrams [32] to generate tests and automate the testing process. Others consider a more fine-grained approach to verify intentional agents [15, 31], focusing on *white box* tests involving in the testing process the inner mechanisms of BDI agents (like plans and goals). This method of testing has however been criticized in [25] as being "too fine-grained", proposing instead to perform testing at a *module* level, that is, considering a set of goals, plans, and/or rules as a single unit. Still other works refer to *software testing* techniques applied on MAS development, focusing on testing agents and their interaction patterns as the main level of abstraction [10, 24]. At implementation level, such *unit testing* is performed in a *Jade* multi-agent system via the JUnit library. The distinct agent-roles that are present in the MAS are tested by means of *mock* agents that communicate with the implemented Jade agents to verify their behavior.

Levels of Testing Software testing is generally categorized in four levels or activities: (a) *Unit testing* is done to verify different individual components of the software system in focus, (b) *Integration testing* verifies the combination of different components together, (c) *System testing* is done to test the system as a whole, and (d) *Acceptance testing* is done to check the compliance of the software with given end-users' and/or relevant stakeholders' requirements.

A categorization for MAS testing from a development-phase activity perspective has been proposed in [28], consisting of five levels: (i) *Unit testing* targets individual components of an agent, (ii) *Agent testing* aims at the combination of the components in an agent including capabilities like sensing its environment, (iii) *Integration or Group testing* includes the communications protocols and the interactions of the agent with its environment or other agents, (iv) *System or Society testing* considers the expected emergent properties of the system as a whole (v) *Acceptance testing* for a MAS stays the same as their counterpart in software testing.

All these categorizations can be seen as guidelines to draw a conceptual line between what should be tested for what purpose and when, in the different phases of software development. This means that for each project it is up to the designer to decide e.g. what counts as units, what interactions are considered group and what are the properties of the system/society. Indeed, testing libraries like JUnit or online continuous integration services like TravisCI or CircleCI stay relatively agnostic on what type of tests are being done. We will follow here the same principle by allowing the designer to create each test suite with different scenarios containing one or multiple agents with varying types and allowing for flexible success/failure criteria.

Coverage An important measure giving insights on the quality of a certain test suite in a given system is *coverage*. Software engineering proposes different criteria for coverage [29], varying from simple *line coverage* (denoting the percentage of the code that is covered by the test cases), to more sophisticated metrics like cyclomatic complexity [26], more commonly known as *branch coverage*. Intuitively, the more a program is covered by a test suite the more confident the designer can be about the behavior of the software. In fact it is a common approach to set a minimum coverage boundary for software projects and if coverage is below this limit the build chain is considered a failure even if the code compiles correctly.

Several works have studied criteria for testing in Agent-Oriented Software Engineering, and particularly in BDI-based agent programming [31]. However, the abstract mechanisms underlying any BDI-based reasoning cycle concerning e.g. treatment of plan context conditions, plan selection and failure handling, alongside the procedural specifications given in one agent's script (e.g. the agent's plans), result in complicated branching in the agent's effective code, a fact that makes defining what is actually covered by a test suite difficult [38, 37].

3 Approach

Instead of investigating dedicated tools for testing BDI agents, our motivation is to study under what conditions and how we can take advantage of existing software testing coverage tools, so as to enable an integration of BDI agent-based development with other types of development, occurring concurrently on a production-level system. This practical (and unavoidable) necessity motivated us to overlook or put aside the warnings and issues indicated in the literature.

Our study focuses in particular on the BDI framework AgentScriptCC (Cross-Compiler) [27], here denoted ASC2. A short overview of ASC2 is presented in section 3.1, whereas section 3.2 presents our approach to testing.

3.1 AgentScript Cross-Compiler (ASC2)

The ASC2 framework is a BDI agent programming framework centred around a *cross-compiler* performing a *source-to-source* translation of a high-level Domain Specific Language (DSL) inspired by AgentSpeak(L)/Jason [34, 6] into executable JVM-based programs. Cross-compilation is not unique to ASC2 and has been used by other recent agent-oriented frameworks such as Astra [14] and Sarl [36]. ASC2 consists of: (1) a logic-based Agent-Oriented Programming DSL; (2) an abstract execution architecture; (3) a translator that generates executable models from models specified by the DSL; (4) tools that support the execution of models.

AgentScript DSL The AgentScript DSL has a very close syntax to AgentSpeak(L)/Jason [34, 6]. The main components of the DSL are (1) initial beliefs, (2) inferential rules, (3) initial goals, and (4) plan rules. The initial beliefs and

goals express the mental state of the agent at the start of the execution. Initial beliefs are a set of Prolog-like facts, and the initial goals designate the first intentions to which the agent commits. Inferential rules are potentially non-grounded declarative rules (Prolog-like), used to infer beliefs from beliefs. Plan rules are potentially non-grounded reactive rules in the form of $e : c \Rightarrow f$ in which f is a sequence of executable steps called the *plan body* that the agent has to perform in response an internal (e.g. *goal adoption*, *belief-update*) or external (e.g. *message reception*, *perception*) event e, if a context condition c is believed to be true by the agent.

While the AgentScript DSL is very close to Jason, the translation-based nature of ASC2 produces some disparities with respect to execution. An important characteristic of this approach is how ASC2 agents access and perform primitive actions [27]. Typically, in interpreter-based BDI frameworks primitive actions need to be properly defined before they can be used by the agent. In ASC2 such redefinitions are not needed and the agent program can directly access any entity on the JVM's class path. An example of this would be the `.print` function in Jason, defined in the standard agent library and that underneath calls Java `print`. In contrast, in an ASC2 program there is no need to define the primitive action; the agent program can call Java/Scala's print function by simply using `#print` (where `#` is the prefix for calling any primitive action).

AgentScript Translator The ASC2 translator generates concurrent programs in a lower-level executable language from agent scripts written in AgentScript DSL. The reasoning cycle of ASC2 follows the same principles of what is proposed for AgentSpeak(L) and further extended by Jason. This reasoning cycle generally includes steps to iterate over internal and external events, find relevant and applicable plans to react to these events, creating intentions to perform the plans and executing the intentions. But, while Jason and many other BDI frameworks implement an interpreter and a reasoning engine to drive the execution the of the agent programs as run-time, in ASC2, all the mechanisms needed for execution with the exception of the externalized plan selection function are generated as part of the agent's executable code in form of control flow statements.

AgentScript Execution Architecture The ASC2 implements an abstract execution architecture that is used as a template for the Translator to generate the concurrent agent programs. The architecture introduced in [27] defines each agent as a modular and extendable *actor-based* micro-system. The Actor model, introduced in [21], is a mathematical theory that treats *actors* as the primitives of computation [20]. Actors are essentially reactive concurrent entities, when an actor receives a message it can send messages to other actors; *spawn* new actors; modify its reactive behavior for the next message it receives. In the current implementation of ASC2, the underlying language is *Scala* and the agents utilize the actor model implementation of *Akka*[3]. The ASC2 architecture also defines multiple components of the agents like their belief base and communication layer

[3] https://akka.io

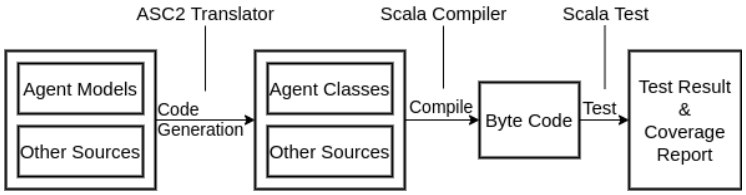

Fig. 1. Compile/Test process of an ASC2 program with sbt

as external dependencies, enabling modularity with respect e.g. automated reasoning or transportation functions.

3.2 Testing Approach

In a typical *unit* or *integration test* of a computational entity under test (e.g. a class, a web service), the designer sets up an initial setting (e.g. one or multiple object instances, web services, a client), and then, based on certain invocations (e.g. function calls, access/service requests), a set of *assertions* are checked to verify the internal state, or some observable behavior of the tested entity, or its effect on the environment (e.g. function results, service responses, modifications of other entities).

Internal attributes (of objects or services) are generally harder to access and therefore to verify. Best practices of Test-Driven Development (TDD) address this issue by means of *Dependency Injection* (DI): the dependencies of each entity should be instantiated from outside the entity and then passed to it e.g. as parameters (typically to the class constructor in object-oriented programming). This allows the tester to isolate and observe the internal mechanisms of the entity under test by using "mocked" dependencies. To enhance testability, multiple components of ASC2 agents, including their belief base and communications layer, are injected as external dependencies.

In any certain situation, we can look at a single agent or multiple agents (a MAS) as a computational entity under test, and this entity has also a set of internal attributes, observable behavior, and possible interactions with its environment. The single agent or multiple agents under test can be instantiated from one or more scripts. The setting could include any other types of entities e.g. other possibly mocked agents, external objects, etc. The initial state of the agent(s) and of the other related entities defines the initial setting of the test, the invocation/probing action of a test suite is typically a series of messages sent to the agents. The expected effect(s), behavior(s) or state(s) of an entity rely heavily on the entity under test. For a small system including one or only a few agents, each message or the beliefs of the agent(s) may be needed to be verified, whereas in a complex system, the designer may only need to verify emergent pattern in the interactions of the agents or major shifts in the state of the system.

```
+!init(W) : W > 1 =>
    Nbr = "worker" + ((#name.replaceAll("worker","").toInt % W) + 1);
    +neighbor(Nbr).

+!token(0) =>
    #coms.achieve("master", done).

+!token(N) : neighbor(Nbr) =>
    #coms.achieve(Nbr, token(N - 1)).
```

Listing 1: Token ring `worker` script in AgentScript DSL

In our approach, we aim to allow the designer to utilize any off-the-shelf testing tool (library, service, etc.) directly into their development chain, even more so to enable the designer to test their program via any standard build chain. In the case of the ASC2 framework, its current implementation is based on Scala, and we considered as target build tool *sbt*[4], which enables us to also use JVM/Scala testing libraries like *JUnit* or *ScalaTest*. We have then developed a sbt plugin[5] that —as part of the compile task—iterates over the scripts written in AgentScript DSL in the project sources and uses the AgentScript Translator to generate Scala implementations of the agents. Code generation is a standard part of build tools like sbt or maven, therefore, the generated sources are also managed by the build tool and are immediately available to rest of the project. The general overview of the *Compile/Test* cycle of an agent-based system developed via ASC2 and built by sbt is presented in figure 1. Note that this process is fully automated by sbt.

A MAS of this type can be started in two ways. After bootstrapping it as an empty instance of the MAS infrastructure, the designer can either use configuration files (e.g. JSON) to specify the agents of the system or alternatively, use lower-level code (e.g. Scala/Java) to manually spawn agents via their respective class in the generated code. In this work, we preferred the latter approach, as it provides better control over the test scenarios.

To complete our Compile/Test process, in addition to the *ScalaTest* library, we also used the *Akka Testing* library: at run-time, ASC2 agents are essentially Akka actor micro-systems and this library provides many convenient tools for testing actors. Both libraries are used out of the box and no modifications have been done to adapt them to the framework. With this configuration, each scenario to be verified can the written as a test suite in ScalaTest to test whether one or multiple agents behave as expected.

4 Illustrative Example

To illustrate an application of our testing approach we consider a MAS constructed around a Token Ring system, commonly used in both distributed sys-

[4] https://scala-sbt.org/
[5] https://github.com/mostafamohajeri/sbt-scriptcc

tems and MAS [27, 8]. This system consists of one master agent and W worker agents; at the start of the program the master sends an $init(W)$ message to all worker agents to inform them of the total number of the workers in the ring, each worker upon receiving this message finds its neighbor, forming a closed ring. Then, T tokens are distributed among the workers, each token has to be passed N times in the ring formed by workers. When all T tokens have been passed N times and this was reported to the master, the program ends.

4.1 Unit/Agent Testing

We will focus in particular on the script of the worker agents shown in listing 1. We perform the tests taking the standpoint of a *whitebox* test engineer, meaning that we test the script of the agent knowing its internal workings; nevertheless, the tests are still performed externally, we do not modify the script in order to test it[6].

Testing Successful Scenarios By viewing the script in listing 1, we can see that the agent has a total of 3 plans for 2 separate goals. Theoretically, we need at least 3 tests to cover the successful execution of all the plans. However, while the success criteria for plans is simple (completion of execution), achievements of goals can be more complicated and the testing framework needs to provide the flexibility to define them. The success criteria for the `init(W)` and `token(N)` goals are quite different. In the latter the expected behaviour in both plans is an observable event, i.e. a certain `achieve` message sent by the agent to another specific agent. In the former case there is no observable behavior and the success criterion is a specific update of the agent's belief base.

The test specification we used for the worker agent can be seen in listing 2. In line 3 an empty MAS object is created. The criterion of success for `init(W)` plan depends on the agent's beliefs, therefore we need to be able to verify the internal state of agent's belief base. First we create an instance of `BeliefBase` class (line 4) and when the agent under test (`worker1`) is being instantiated (line 10), this object is injected in the agent as its belief base; with this approach at any point in the tests we can simply access the agent's beliefs to query them for verification purposes or even modify the agent's belief base for setting up test scenario states.

Only one agent (`worker1`) is under test and the other agents present in the suite can be mocked. As ASC2 agents are actor micro-systems, an agent can be mocked by a single actor. In lines 5 and 6, two *probe* actors are created to be the stand-ins for the master agent and (`worker1`)'s neighbor in the tests and they are then registered to the system (lines 11 and 12). This type of mocking gives us the ability to verify all the interactions that the agent under test may have had with these probe actors.

The rest of the test suite contains 3 tests, in the first test in line 18 a goal event `init(50)` is sent to the `worker1` agent and it is expected that after this

[6] https://github.com/mostafamohajeri/agentscript-test

```
1  class TokenRingWorkerSpec extends ... {
2
3    val mas = new MAS()
4    val verifiableBB  = new BeliefBase()
5    val mockedMaster = testKit.createTestProbe[IMessage]()
6    val mockedNeighbor = testKit.createTestProbe[IMessage]()
7    val worker
8
9    override def beforeAll(): Unit = {
10     mas.registerAgent(new worker(bb = verifiableBB), name = "worker1")
11     mas.registerAgent(mockedMaster, name = "master")
12     mas.registerAgent(mockedNeighbor, name = "worker2")
13     worker = mas.getAgent("worker1")
14   }
15
16   "A worker agent" should {
17     "have its neighbor in its belief base after `!init(N)`" in {
18       worker.event(achieve,"init(50)").send()
19       mockedMaster.expect(GoalAchievedMessage())
20       assert(verifiableBB.query("neighbor(worker2)") == true)
21     }
22
23     "send a `!done` to master on `!token(0)`" in {
24       worker.event(achieve,"token(0)").send()
25       mockedMaster.expect(event(achieve,"done").source(worker))
26     }
27
28     "send a `!token(N-1)` to its neighbor on `!token(N)`" in {
29       worker.event(achieve,"token(10)").send()
30       mockedNeighbor.expect(event(achieve,"token(9)").source(worker))
31     }
32   }
33 }
```

Listing 2: Test suite for the `worker` agent

goal is achieved (line 19), the belief base of the agent contains the belief defined by the term `neighbor(worker2)` which is verified in line 20. In the next test, a goal message `token(0)` is sent to the agent (line 24) and then it is verified that the agent sends a `done` message to the master (line 25). The final test follows the same pattern by sending a goal message `token(10)` (line 30) and the verification includes a `token(10-1)` message to its neighbor (line 30). Note that in all the tests, the messages sent to the `worker1` agent do not specify any source, this is because in the script in listing 1, the source of the messages is not checked meaning it is not necessary to specify the source. As these tests are written in a standard testing library, build tools such as *sbt* can execute them in their build chain. By running the tests in the sbt shell we are able to see the output presented in listing 3 that indicates our program has passed this test.

```
[info] A worker agent should
[info] - have its neighbor in its belief base after `!init(N)`
[info] - send a `!done` to master on `!token(0)`
[info] - send a `!token(N-1)` to its neighbor on `!token(N)`
...
[info] All tests passed.
```

Listing 3: Output of the `worker` agent test suite

```
1  "A worker agent" should {
2    "send a `NoApplicablePlan()` on `!init(-1)`" in {
3      worker.event(achieve,"init(-1)").source(mockedMaster).send()
4      mockedMaster.expect(NoApplicablePlan())
5    }
6
7    "send a `NoRelevantPlan()` on `!unknown`" in {
8      worker.event(achieve,"unknown").source(mockedMaster).send()
9      mockedMaster.expect(NoRelevantPlan())
10   }
11 }
```

Listing 4: Failure tests for `worker` agent

Testing Failure Scenarios Successful executions are only a part of the full story. Indeed, in software testing it is acknowledged that covering *failures* is both more important and challenging, and thus requires more critical thinking by the test engineer [29]. Interestingly, failure tests are especially important in agent-based programming because failing under certain conditions may sometimes be the correct behavior for an agent.

Two failure tests are presented in listing 4. The first test sends a `init(W)` goal message to the agent with `W=-1` (line 3) but the first plan is applicable only for `W > 1` and the expected behavior of the agent in this situation is a failure which is verified by expecting a `NoApplicablePlan` message. In the second test, a goal message `unknown` is sent to the agent (line 8) for which the agent does not have any plans and it should reply with a `NoRelevantPlan` (line 9). Note that failure of a goal is not only reflected by the absence of an applicable plan or more generally failure in execution of a plan; similar to the success scenarios, the designer can define any other arbitrary criteria for a failure scenario.

Although we acknowledge that testing an agent program for every possible failure can easily become an infeasible task [38, 37], certain failures may be particular important for the designer to test, therefore there is value in enabling this possibility.

4.2 Coverage

We explore at this point whether and how off-the-shelf coverage tools such as *scoverage*[7] can be used for code coverage analysis of agent programs written in ASC2, considering both statement and branch coverage aspect. To perform this we simply add the scoverage plugin to our project and generate a coverage report.

The coverage report produced for the worker agent by means of the previous tests is presented in Table 1. The `worker.Agent` row shows the coverage for the internal mechanisms of the agent, like e.g. *event handling*, while the other rows show the coverage report for each separate event, as an example, the `worker.token_1` refers to an event `token` in `worker` agent with 1 parameter. The branch coverage report mainly concerns conditional statements in the generated Scala code of the agent and should be regarded only as informal information about the coverage of the main script.

These results show that our tests indeed covered most of the behaviors that the agent might have. In fact, by exploring the coverage analysis we can see the reason for which the `worker.token_1` has less coverage: the missed branch can be explained by the fact that the tests did not include any scenario in which the `token(N)` plan fails. Also note that while the example script did not contain any sub-goals or conditional statements in the plans, ASC2 Translator generates sub-goal adoptions as function calls and translates conditional statements to their counterpart in the underlying language, therefore, coverage tools like *scoverage* are able to calculate the correct number of covered and total possible branches for deeper goal-plan trees.

Component	Statement Coverage %	Branch Coverage (Covered/Total)
`worker.Agent`	93.5	6/6
`worker.init_1`	93.5	2/2
`worker.token_1`	80.2	3/4

Table 1. Coverage analysis of the `worker` agent

4.3 Integration/System Testing

Even following the guidelines on categorizing different levels of testing in MAS [28], there is no definite technical distinction in place. Typically test libraries provide mechanisms such as annotations for the designer to label test suites with its (their) related level(s) to orchestrate their execution. As illustration, we consider an integration test to verify a token ring MAS system consisting of the previously mentioned `worker` agents and a `master` agent. The test suite is reported in listing 5.

[7] http://scoverage.org/

```
1  class TokenRingIntegrationSpec extends ... {
2
3    //a communication layer that records a trace of the interactions
4    object recordedComs extends AgentCommunicationsLayer { ... }
5
6    val token_pattern = "token\\([0-9]+\\)".r
7    val done_pattern = "done".r
8
9    "A token ring MAS with W = 100, T = 50 and N = 4" should {
10     "have 250 `token(X)` and 50 `done` message" in {
11       // create the agents
12       mas.registerAgent(new worker(coms = recordedComs), num = 100)
13       mas.registerAgent(new master(coms = recordedComs), name = "master")
14       // invoke the system
15       mas.getAgent("master").event(achieve, "start(50,4)").send()
16       // verify the interactions
17       watchdog.expectTerminated( mas, 10.seconds )
18       assert(recordedComs.trace.count(token_pattern.matches) == 250)
19       assert(recordedComs.trace.count(done_pattern.matches) == 50)
20     }
21   }
22 }
```

Listing 5: Integration test suite for the token ring system

The test will be centered around the interactions between agents and the state of the system in a specific setting of our token ring. The token ring is defined with 100 `worker` agents and 1 `master` agent (lines 12-13), and, to be able to verify the exhibited interactions, we use dependency injection to initialize all the agents by means of an overridden instance of the communication layer (line 4), created to record every message passed in the system into a list.

To invoke the system, a `start(T,N)` is sent to the `master` agent (line 15). We are interacting with the `master` from a *black box* perspective: although the event `start(T,N)` is exposed, the internal mechanisms of this agent are assumed to be unknown.

Three criteria are verified for this system. Firstly, we consider a system level performance based criteria as we expect the system to be terminated under 10 seconds (line 17). Next, we use two known expectations from a token ring system to verify the correct execution of the system: at the end of execution, there should be (a) T number of `done` messages and (b) $T \times (N + 1)$ number of `token(X)` messages in the trace. The interaction verification statements are presented respectively in lines 18-19. Recalling the flexible definitions of testing levels, note that these integration/system test could be considered from the perspective of `master` agent as a unit/agent level test possibly with mocking the `worker` agents. Similar to previous tests, running this suite via sbt yields the output in listing 6.

```
[info] A token ring MAS with W = 100, T = 50 and N = 4 should
[info] - have 250 `token(X)` and 50 `done` message
...
[info] All tests passed.
```

Listing 6: Output of the token ring integration test suite

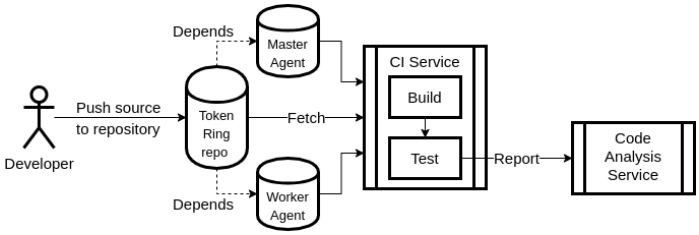

Fig. 2. Continuous integration applied on a Token ring program whose master and worker agent scripts are located on other repositories.

4.4 Continuous Integration

The proposed approach for testing can be easily combined with online CI services. This process generally includes utilizing source repositories like Github[8], CI services like TravisCI and code analysis services like Coveralls[9]. The only step needed to set the CI cycle for an ASC2 project is to configure the source repository of the project in a way that the automated CI cycle is triggered on every `push` to the repository. This can be done by adding a configuration file that provides information for the CI service how to compile and test the project via sbt.

Following this method, a MAS project does not need to be only located in a single source repository. For instance, different types of agents can be developed in different projects by separate teams and only be used as dependencies in the development of the system. We believe this is an interesting practical innovation, improving the scalability of MAS projects with respect to their development.

An overview of an example CI process for the token ring is presented in Figure 2 in which the sources of `worker` and `master` agents are located in separate repositories, and a third token ring repository uses them as dependencies. When the system designer pushes the project to the repository, the CI service fetches the source and compiles and tests it via sbt and records the results[10]. Then, the code coverage report is committed to the code analysis service[11].

[8] https://github.com/
[9] https://coveralls.io/
[10] https://travis-ci.com/github/mostafamohajeri/agentscript-test
[11] https://coveralls.io/github/mostafamohajeri/agentscript-test

5 Discussion and Future Developments

Despite the critical points/observations concerning MAS testing raised in the literature, in this paper we provide several support arguments for using mainstream testing tools for MAS and agent-based programming, by means of a concrete use case. We implemented a multi-agent system reproducing a token ring benchmark with the framework ASC2, and then we run tests (success, failure, coverage) at unit/agent level as well as at integration/system level.

At the unit and agent level (unit testing) we performed tests concerning events, plans and goals. The somehow unexpected result of the experiment is that such an approach does not neglect the theoretical complexity of BDI agents but it truly offers a complementary tool for their development. We were able to test successful (plan) completions, internal states and the belief base, failures, and fine-grained interactions. These possibilities can be seen as offering constructs mapping e.g. to declarative and procedural goals in BDI agents [40]: the designer can define the achievement/failure of a goal not only in terms of completion/exception of a plan, but also as determined by any arbitrary indicator internal or external to the agent. This showed that testability of agent programs defined in a framework is closely related to the design choices of that framework.

At the integration/group and system/society level (integration testing) we performed tests with simple verification criteria, but these criteria can easily be extended to more sophisticated and realistic interaction analysis and verification methods developed by the MAS community [7]. Additionally, we illustrated how the proposed approach enables the MAS designer to take advantage of continuous integration (CI) services without extra effort. This is particularly important for MAS designers that require to integrate and test their work continuously with other projects.

There is an additional benefit of using mainstream test tools for BDI agents, and especially for frameworks that are based on higher-level logic-based DSLs. Those frameworks generally map primitive actions to constructs specified in a lower-level programming language like Java. By using a testing process compatible with both higher level models and lower level implementations, the testing process can be more efficient and seamless for the designer specially if the agent models are only a part of a project that includes other computational entities that are being developed alongside the agents.

An issue in using mainstream test libraries for a BDI framework with a logic-based DSL is the disparity between the high-level agent DSL and the lower-level language used for the tests. This can be addressed by either developing approaches to write tests in the high-level DSL or creating interfaces for the low-level language to enable the test engineer to implement tests at a proper level of abstraction. In this work we have taken the latter approach. The intuition behind this choice was that frameworks based on cross-compila-tion [14, 36] produce source codes that can be directly integrated within standard build tools.

Can our results be generalized to other agent programming frameworks? Motivated by the success of works like AJPF/MCAPL [13] that provides model checking for multiple BDI frameworks, as a future study we intend to explore

how to apply this approach to a wider range of MAS frameworks. Yet, we can already trace some higher-level considerations. The answer, at the unit/agent level, depends on compilation and the execution model of those frameworks. For frameworks like Jade and JS-son [23], that use mainstream programming languages to define agents, these tools should be compatible out of the box with minor effort [24]. For cross-compilation-based frameworks like Astra [14] and ASC2 [27] it is only the matter of tooling (e.g. build tool plugins) to allow them to use mainstream testing tools. For interpreter-based frameworks like Jason [6] and GOAL [22], because they require their own dedicated reasoning engines and execution environment, testing via such tools may prove to need more work and possibly modifications to the framework. This issue may be not so problematic, as there are already many works that propose dedicated testing and debugging approaches for interpreter-based frameworks [25].

At the integration and system level, and also with respect to compatibility with CI services, generally externalized to the execution of the tested entity, we believe it is possible to consolidate other frameworks regardless of their compile/interpret model. This could lead to seamless integration testing of systems defined in each framework with mainstream software testing tools or dedicated ones.

In perspective, our overarching research concerns socio-technical and complex multi-domain infrastructures; we believe that Agent-Oriented Software Engineering can be a powerful technical tool with robust theoretical foundations for designing, modelling, implementing and testing such systems. Enhancing their development cycle goes with a seamless integration of multi-agent systems into modern infrastructures. This is a critical requirement to utilize the full potential of MAS in a real production-level setting.

Acknowledgements

The work as presented in this paper has been done as part of the Dutch Research project 'Data Logistics for Logistics Data' (DL4LD), supported by the Dutch Organisation for Scientific Research (NWO), the Dutch Institute for Advanced Logistics 'TKI Dinalog' (http://www.dinalog.nl/) and the Dutch Commit-to-Data initiative (http://www.dutchdigitaldelta.nl/big- data/over-commit2data) (grant no: 628.009.001).

References

1. Software testing services market by product, end-users, and geography - global forecast and analysis 2019-2023 (Aug 2019), https://www.industryresearch.co/software-testing-services-market-14620379
2. Bakar, N.A., Selamat, A.: Agent systems verification : systematic literature review and mapping. Applied Intelligence **48**(5), 1251–1274 (2018)
3. Behrens, T.M., Dix, J.: Model checking multi-agent systems with logic based Petri nets. Annals of Mathematics and Artificial Intelligence **51**(2-4), 81–121 (2007)

4. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifiable multi-agent programs. Lecture Notes in Artificial Intelligence **3067**, 72–89 (2004)
5. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Autonomous Agents and Multi-Agent Systems **12**(2), 239–256 (2006)
6. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of Agent-Oriented Programming. In: Multi-Agent Programming: Languages, Platforms and Applications, pp. 3–37 (2005)
7. Botía, J.A., Gómez-Sanz, J.J., Pavón, J.: Intelligent data analysis for the verification of multi-agent systems interactions. Lecture Notes in Computer Science **4224 LNCS**, 1207–1214 (2006)
8. Cardoso, R.C., Zatelli, M.R., Hübner, J.F., Bordini, R.H.: Towards benchmarking actor- and agent-based programming languages. In: Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control. p. 115–126. AGERE! 2013, Association for Computing Machinery, New York, NY, USA (2013)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (2000)
10. Coelho, R., Kulesza, U., von Staa, A., Lucena, C.: Unit testing in multi-agent systems using mock agents and aspects. In: International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. p. 83–90. SELMAS '06 (2006)
11. Dastani, M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3), 214–248 (2008)
12. David, N., Simão Sichman, J., Coelho, H.: Towards an emergence-driven software process for agent-based simulation. Lecture Notes in Computer Science **2581**(301041), 89–104 (2003)
13. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Practical verification of decision-making in agent-based autonomous systems. Automated Software Engineering **23**(3), 305–359 (2016)
14. Dhaon, A., Collier, R.W.: Multiple inheritance in AgentSpeak(L)-style programming languages. In: Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control. Association for Computing Machinery (2014)
15. Ekinci, E.E., Tiryaki, A.M., Çetin, Ö., Dikenelli, O.: Goal-Oriented Agent Testing Revisited. In: Luck, M., Gomez-Sanz, J.J. (eds.) Agent-Oriented Software Engineering IX. pp. 173–186 (2009)
16. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. USA, 1st edn. (1999)
17. Ferrando, A., Dennis, L.A., Ancona, D., Fisher, M., Mascardi, V.: Verifying and validating autonomous systems: Towards an integrated approach, vol. 11237. Springer International Publishing (2019)
18. Fisher, M., Mascardi, V., Rozier, K.Y., Schlingloff, B.H., Winikoff, M., Yorke-Smith, N.: Towards a framework for certification of reliable autonomous systems, vol. 35. Springer US (2020)
19. Herzig, A., Lorini, E., Perrussel, L., Xiao, Z.: BDI Logics for BDI architectures: Old problems, new perspectives. KI - Künstliche Intelligenz **31**(1), 73–83 (2017)
20. Hewitt, C.: Actor model of computation: Scalable robust information systems (2010)
21. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on

Artificial Intelligence. p. 235–245. IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973)
22. Hindriks, K.V.: Programming Rational Agents in GOAL. In: Multi-agent programming: Languages, platforms and applications, chap. 4, pp. 119–157 (2009)
23. Kampik, T., Nieves, J.C.: JS-son - A lean, extensible JavaScript agent programming library. In: Engineering Multi-Agent Systems. vol. 12058 LNAI, pp. 215–234 (2020)
24. Khamis, M.A., Nagi, K.: Designing multi-agent unit tests using systematic test design patterns (extended version). Engineering Applications of Artificial Intelligence **26**(9), 2128–2142 (2013)
25. Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Automating failure detection in cognitive agent programs. Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS pp. 1237–1246 (2016)
26. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering **SE-2**(4), 308–320 (1976)
27. Mohajeri Parizi, M., Sileno, G., van Engers, T., Klous, S.: Run, agent, run! architecture and benchmarking of actor-based agents. In: proceedings of Programming based on Actors, Agents, and Decentralized Control (AGERE 2020). pp. 11–20 (2020)
28. Moreno, M., Pavón, J., Rosete, A.: Testing in agent oriented methodologies. Lecture Notes in Computer Science **5518 LNCS**, 138–145 (2009)
29. Myers, G.J., Sandler, C.: The Art of Software Testing. John Wiley & Sons, Ltd (2012)
30. Nguyen, C.D., Perini, A., Bernon, C., Pavón, J., Thangarajah, J.: Testing in multi-agent systems. Lecture Notes in Computer Science **6038 LNCS**, 180–190 (2011)
31. Padgham, L., Zhang, Z., Thangarajah, J., Miller, T.: Model-based test oracle generation for automated unit testing of agent systems. IEEE Transactions on Software Engineering **39**(9), 1230–1244 (2013)
32. Padgham, L., Winikoff, M.: Developing intelligent agent systems: A practical guide, vol. 13. John Wiley & Sons (2004)
33. Poutakidis, D., Winikoff, M., Padgham, L., Zhang, Z.: Debugging and Testing of Multi-Agent Systems using Design Artefacts. No. May 2014 (2009)
34. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away. pp. 42–55 (1996)
35. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS1995). pp. 312–319 (1995)
36. Rodriguez, S., Gaud, N., Galland, S.: Sarl: A general-purpose agent-oriented programming language. In: International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). vol. 3, pp. 103–110 (2014)
37. Winikoff, M.: BDI agent testability revisited, vol. 31. Springer US (2017)
38. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. IJCAI International Joint Conference on Artificial Intelligence pp. 4217–4221 (2015)
39. Winikoff, M., Dennis, L., Fisher, M.: Slicing Agent Programs for More Efficient Verification, vol. 11375 LNAI. Springer International Publishing (2019)
40. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: 8th International Conference on Principles of Knowledge Representation and Reasoning. p. 470–481. KR'02 (2002)
41. Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing for agent systems. pp. 10–18. No. May 2014 (2007)

42. Zhou, X., Cushing, R., Grosso, P., Engers, T.V.: Policy Enforcement for Secure and Trustworthy Data Sharing in Multi-domain Infrastructures. In: Engineering Multi-Agent Systems. pp. 104–113 (2020)