Automating Failure Detection in Cognitive Agent Programs

Vincent J. Koeman
Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
v.j.koeman@tudelft.nl

Koen V. Hindriks
Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
k.v.hindriks@tudelft.nl

Catholijn M. Jonker
Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
c.m.jonker@tudelft.nl

ABSTRACT

Debugging is notoriously difficult and extremely time consuming but also essential for ensuring the reliability and quality of a software system. In order to reduce debugging effort and enable automated failure detection, we propose an *automated testing framework* for detecting failures in cognitive agent programs. Our approach is based on the assumption that modules within such programs are a natural unit for testing. We identify a minimal set of temporal operators that enable the specification of test conditions and show that the test language is sufficiently expressive for detecting all failures in an existing failure taxonomy. We also introduce an approach for specifying *test templates* that supports a programmer in writing tests. Furthermore, empirical analysis of agent programs allows us to evaluate whether our approach using test templates detects all failures.

Keywords

multi-agent systems; testing; verification

1. INTRODUCTION

Debugging is notoriously difficult and extremely time consuming [23] but also essential for ensuring the reliability and quality of a software system. Manual testing, using, for example, a debugger for single-step execution to identify differences between observed and intended behaviour, however, is not an efficient failure detection method and heavily relies on the programmer to identify the failure. In order to reduce debugging effort and enable automated failure detection, we propose an *automated testing framework for cognitive agent programs*. Automated testing yields a reduction in the effort needed to detect a failure and is more effective than code inspection methods [24]. In addition, it also facilitates running tests repeatedly at no additional costs.

The aim of this paper is to introduce and develop a test framework that supports *automated failure detection* for programs written in rule-based agent programming languages that use a logic for representing the cognitive state of the agent, including e.g. its beliefs. A *failure* is an event in which a system does not perform a required function within specified limits [16]. Failures thus are manifestations of undesired behaviour. They are caused by a *fault*, an incorrect step, process, or data definition in a program [16] or mistake in a program [27]. Upon detecting a failure, a programmer needs to locate and correct the fault that causes the failure. Our focus is on automating the detection of failures and on dynamic analysis, i.e., the process of evaluating a system or component based on its behaviour during execution [16]. We will see, however, that the test framework introduced also provides support for fault localization.

The main contribution of this paper is an *automated test framework* for cognitive agent programs that provides support for detecting frequently occurring failure types. As a first step towards such a framework, we argue that program modules instead of the top-level agent program are the most natural unit for testing, and that test conditions should be associated with modules. Second, we introduce two basic temporal operators that in practice are sufficient for specifying test conditions to detect failures. Third, using this generic framework, we propose test templates for failure types that have been identified in a previously developed taxonomy in [27]. Finally, we introduce a test approach for deriving test templates given some initial functional requirements. In order to empirically evaluate and demonstrate that our framework is expressive enough to detect all failure types, we verify that we can reproduce and identify all failures found in the sample used in [27]. The test templates that we introduce can be considered as a refinement of the failure taxonomy proposed in [27]. We show that by automating testing, we are able to identify more failures. We also show that our work is not biased towards this sample by demonstrating that the same test approach is also able to identify all failures in different samples of programs.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces our automated test framework. Section 4 introduces the test templates and approach. Section 5 evaluates the test framework. Section 6 concludes.

2. RELATED WORK

In general, different techniques for detecting failures of program code are available, ranging from *inspection* of source code and logs to automated *testing* tools [24]. The need for debugging techniques and test approaches for agent-oriented programming has been broadly recognized [2, 9, 10]. Techniques for agent-oriented programming need to be based on the underlying agent paradigm [21, 28]. However, this is a significant challenge, as they should for example take into account that agents execute a specific decision cycle and operate in non-deterministic environments [1, 4, 14].

Appears in: *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016), J. Thangarajah, K. Tuyls, C. Jonker, S. Marsella (eds.), May 9–13, 2016, Singapore.*

To facilitate code inspection, a (source-level) debugger that supports single-step execution of an agent can be used. [19], for example, provides a useful tool for detecting failures in a program. Debugging is particularly useful for zooming in on a bug of which the location is already more or less clear; it requires a programmer to go through the execution steps of an agent program one-by-one and to observe any mistakes in the program manually. This method of code inspection is not only inefficient, but also subjective, as it depends on observations made by a programmer. It requires the same effort repeatedly since, after correcting an identified fault, a programmer needs to manually evaluate program behaviour again to verify that a fix does not introduce new defects. Automated testing offers a method that is complementary to debugging, able to find different kinds of failures, and thus increasing the overall number of detected failures [23, 24]. Moreover, debugging techniques that support the localization of the fault that causes a failure that has been identified by automated tests are still needed. Such techniques include state inspection, mechanisms for browsing and searching (historical) agent states, etcetera [7, 19].

Five levels of testing a multi-agent system (MAS) are distinguished in [20]: *unit*, *agent*, *integration*, *system*, and *acceptance*. In this classification, unit testing targets components that are part of an agent. Testing an agent means to test the integration of these components as well as its ability to interact with its environment. Integration testing focuses on agent interaction and communication protocols, system testing concerns the target operating environment, and acceptance testing takes the customer's perspective into account. The focus of this paper is on unit and agent testing.

Testing can target different artifacts [24]. The test framework of [28], for example, targets Prometheus design models. The paper presents a mechanism for generating suitable test cases. Similarly, the methodology of [12] is based on specific 'meta-models' and 'protocol descriptions'. Other methods focus on the interaction between agents specifically, often using 'mock agents' to check if specific interaction has taken place [6, 18]. The concept of a 'mock agent', i.e., a specific agent that is used for testing, is also used in the work of [5] to test an agent's behaviour based on 'agent stories'. Most of these approaches consider an agent to be the smallest possible artifact to test. Other approaches target more specific artifacts like plans ([22, 25]) or goals ([11]). In this work, however, we argue that modules are the most suitable target artifact. This is partly motivated by the work of [15], which states that techniques that rely heavily on formal methods or sophisticated program analysis are valuable, but difficult to apply in practice, and thus not always effective in finding real bugs. *Bug patterns* (code idioms that are often errors) can be used to facilitate the development of simple detectors (e.g., test templates) that facilitate efficient bug detection in real applications. Therefore, unlike these existing methods, we focus on *how to create tests that can detect specific bug patterns in cognitive agents.*

3. AUTOMATED TESTING FRAMEWORK

We introduce an automated test framework for rule-based agent programming languages that use some knowledge representation (KR) for representing the cognitive state of an agent. We aim for a framework that is as generic as possible, but in order to introduce a concrete framework we need to make a number of assumptions about these languages. First, agents have a *perception processing component* that they use for processing received percepts, which are simple facts that we denote by $p(\vec{t})$ (or simply p if parameters do not matter). We assume that percepts are stored each agent cycle when received from an environment for further processing. Second, we assume that the KR used supports a *negation* operator, denoted by not. Third, we assume that the cognitive state includes a *belief state* that can be queried and which allows to make percepts persistent. Fourth, we assume that agents somehow can represent *goals* that they want to achieve as part of their state; agents can do this, for example, by means of events, plans, or declarative goals. Otherwise, the structure of an agent's cognitive state depends of the specifics of the agent programming language, which almost always includes more, e.g., also events, messages, and/or plans. Fifth, we assume that the language provides support for aggregating or encapsulating basic language elements such as knowledge, goals, plans, and/or rules in what we call *modules* in this paper. Finally, we assume that an agent has a top-level module, which enables associating test conditions with the agent itself.

Three important design questions need to be addressed to define an automated test framework for cognitive agent programs. First, we need to specify what the *basic unit or program component* is that tests are performed on or are associated with. Second, we need to define a *test language* for specifying test conditions. And, third, we need to specify an *infrastructure that automates running tests.*

3.1 Modules Basic Unit for Testing

An important question that we need to address when setting up a test framework is what the *target unit* for testing is in an agent-oriented program. We argue that a test framework for agent programs should not focus on knowledge bases to avoid reinventing the wheel, but developers should rather re-use existing (unit) test frameworks for the underlying KR technology of an agent programming language. For example, a language that uses SWI Prolog for KR should aim at re-using the available unit test framework [26]. We have also found that, in practice, testing at the level of individual goals, plans, or rules is too fine-grained and not that useful. Writing tests at the level of individual rules, for example, would not only result in more test than source code, but even worse, would not focus on the failures that need to be detected. A more suitable level is the aggregate level that collects multiple goals, plans, and/or rules in a single unit. We call such units *modules* and assume agent programming languages provide some level of support for modules. We will therefore *associate test conditions with modules* and introduce a test language that supports this.

Modules thus can be perceived of as components in an agent program that set goals or plans and generate actions to be executed, much like individual basic actions but at a higher level of abstraction. A module thus provides an execution unit smaller than the agent program itself but larger than other basic language elements. By using modules as targets for testing, we provide a developer with some additional control over which behaviour of the agent program is tested, as only the behaviour generated by the module will be evaluated. This allows a developer to write tests that target only specific parts of an agent program.

An agent enters a module when it starts executing the module and exits the module again when module execu-

test	:=	[*timeout*] *moduletest** *agenttest*$^+$
timeout	:=	`timeout =` *integer* `.`
agenttest	:=	*id* (`,` *id*)* `{` *testaction*$^+$ `}`
testaction	:=	`do` (*action* | *id*) [`until` ψ] `.`
moduletest	:=	`test` *id* `with` [`pre{` ψ `}`] [`in {` χ^+ `}`] [`post{` ψ `}`]
ψ	:=	*stateop*(ϕ) | `done(`*action*`)` | $\psi \wedge \psi$
χ	:=	`never` ψ | ψ `leadsto` ψ `.`

Table 1: Test Language Grammar

tion is finished. These execution points provide two natural places for introducing test conditions. We will associate *pre-conditions* and *post-conditions* with a module that is evaluated, respectively, when entering and when exiting the module. Although useful, in practice pre- and post-conditions are not sufficient for monitoring the trace, i.e., behaviour and states, generated while a module is executed. To be able to evaluate a module's behaviour, we therefore also introduce so-called *in-conditions* that are associated with a module. An in-condition is a temporal property evaluated on the trace generated by a module. These conditions allow the detection of failures that occur during module execution. These conditions also provide better support for fault localization as a test is terminated upon a failure at the code location where the failure was detected.

3.2 Test Language

A test language should provide support for two main tasks: *setting up a test* and *specifying* which *test conditions* should be evaluated. To this end, we introduce the test language in Table 1. We have developed a test framework[1] that, when provided with a *test* program as specified by the grammar, will initialize and set up the infrastructure for running an agent system and (external) environment in which test will be automatically performed. A *timeout* can be specified to ensure termination of the test after a specified time. A timeout is global and specifies how many time (in seconds) is allowed to pass before the entire test should have been completed. If a timeout happens, the test is aborted.

Test Setup The agents that are part of a test need to be referenced explicitly in a test program by means of their *id*'s. These agents are launched when the test is started and automatically connected to an environment, if available, to receive percepts from and perform actions in that environment. The fact that agents are launched, however, does not mean that the program code of these agents is executed. Instead, the *testaction*s that are specified in an *agenttest* clause (see Table 1) are performed when the test is run. Test actions can be preparatory actions `do` *action* for, e.g., initializing an agent's state, where *action* can be a sequence of actions that are available in an agent language. Test actions can also be instructions `do` *id* to execute a module, with *id* the module's name. Note that it is possible to run an agent itself by executing its top-level module (each agent is assumed to have one). Finally, an `until` ψ condition can be associated with a module (or, less usefully, an action) that terminates execution when a state condition ψ holds. An agent test thus determines which actions and modules are executed and when they should be terminated. An agent test can be shared by multiple agents, but it is also possible to define different actions for different agents, which will then be executed in parallel.

The environment that agents operate in is not explicitly referenced in our test language because the specifics of starting an external environment are very different for each agent language. Here, we simply assume that a language-specific mechanism is used for connecting agents to an environment.

Test Conditions As explained, test conditions are associated with modules. Which conditions should be evaluated when a module is executed is specified by a `test` *id* `with` statement, where *id* is a module name. Optionally, a pre-condition `pre{` ψ `}`, a post-condition `post{` ψ `}`, and an in-condition `in {` χ^+ `}` can be associated with the module. The pre-condition of a module is a state condition ψ that should hold when a module is entered (otherwise, the test fails). Similarly, a post-condition is a state condition that should hold when a module is exited (terminated). An in-condition is a temporal condition that specifies which behaviour is expected of a module.

A state condition ψ is a condition of the form *stateop*(ϕ) on the cognitive state of an agent, where ϕ is an expression in a KR language, or a condition of the form `done(`α`)` on the action α that an agent has just performed. The state conditions *stateop*(ϕ) that are supported will be different for each agent language, as they each use a specific cognitive state structure and associated state operators *stateop* for inspecting that state. Querying the beliefs of an agent, for example, may be written as ϕ`?` in one language and as `bel(`ϕ`)` in another.

A temporal property or condition is a statement of the form `never` ψ or ψ `leadsto` ψ. Conditions `never` ψ can be used to specify safety conditions, i.e., things that never should occur. Conditions ψ `leadsto` ψ can be used to specify liveness conditions, i.e., things that are supposed to occur sooner or later after something else has happened [3].

Our test language is based only on the two basic temporal operators `never` and `leadsto`. Most importantly, this is so because it turns out to be sufficient for detecting failures (as we show below as well). We also do not want to complicate our test language more than strictly necessary, as we aim for our language to be used by trained agent programmers. We therefore want to minimize the level of acquaintance with concepts from temporal logic that is needed. In particular, we want to avoid nesting of temporal operators because conditions would quickly become difficult to read in that case. Finally, we note that some temporal operators such as `always` ψ can be introduced as syntactic sugar for `never not(`ψ`)`. Note that this assumes, however, that the `not` can be applied to a state condition or moved inwards into the KR expressions used in the state condition, which is not the case in most agent languages. Similarly, `eventually` ψ can be introduced as shorthand for `true leadsto` ψ.

Although it is outside the scope of this paper to provide a formal semantics of the test conditions, we briefly introduce the basic semantic model that we assume informally. A run or trace of an agent program consists of a (finite or potentially infinite) sequence of cognitive states of the agent. Test conditions associated with a module are evaluated on (partial) traces generated by that module. For testing purposes, these conditions are assigned one of three values: *undetermined*, *passed*, or *failed*. Initially, all test conditions of a module have the value *undetermined*. The pre-condition of a module, if specified, is evaluated on the current state when

[1]The source and implementation of the test framework are available at `http://goalhub.github.io/eclipse`

P1:	failure to deal with percept
P2:	other incorrect percept processing
G1:	failure to add a goal that should be added
G2:	failure to drop a goal that should be dropped
G3:	adding a goal that shouldn't be added
G4:	incorrectly adding a second goal of the same type
G5:	dropping a goal that shouldn't be dropped
A1:	selecting the wrong (user-defined) action
A2:	beliefs not updated correctly when action performed
A3:	action selected when should be doing nothing
A4:	action interface mismatch
O:	other failure not classified above

Figure 1: Failure Taxonomy of [27]

entering the module and assigned *passed* when the condition succeeds, and *failed* otherwise. Similarly, the post-condition is evaluated on the current state when a module is exited or terminated. The value of in-conditions is (re-)evaluated every time the cognitive state of the agent changes while the module is being executed. The temporal operator of the condition determines whether the value is updated:

- `never` ψ: the value is changed to *failed* if ψ holds in the state; the value is changed to *passed* if the module (or test) is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- ψ `leadsto` ψ': if the module (or test) is terminated, the value is changed to *passed* if every state where ψ holds has been followed by a state where ψ' holds (and vacuously so if ψ did never hold); otherwise, the value is changed to *failed*. If the module (or test) has not been terminated yet, the value is *undetermined*.

A test is aborted as soon as a condition is assigned the value *failed*. In that case, the entire test is regarded as failed, indicating that something needs to be fixed. Note that when a test is terminated (whether aborted or not), all conditions will have been assigned the value *passed* or *failed*. If a test is terminated because of a timeout, this does not always imply that the test is a failure; if all conditions are passed, the test is considered to have passed as well. Finally, our test framework keeps track of the last code instruction in an agent program that is executed and points at the corresponding code location when testing is terminated (aborted), which is useful for fault localization.

4. TESTING FOR FAILURES

In order to verify that we can detect all failures in agent programs with our framework, we show that we can detect the same failures that were identified by [27] in a given agent program sample that was used to define the failure taxonomy of Figure 1. We provide a systematic approach for detecting failures by introducing test templates that target specific types of failures in the taxonomy, and a systematic method for using these templates for testing.

4.1 Test Templates

A test template consists of one or more templates for individual test conditions. We provide test templates for each failure type in the taxonomy, except for *A4*, which calls for another detection method and raises a specific design issue for test frameworks. We note that a test condition specifies expected behaviour, and a violation indicates a failure.

P1, P2: Failure to process percept (correctly).

In order to define test conditions for percept processing, we assume a state operator $\mathsf{percept}(p)$, indicating that p is perceived, that can be used to inspect the contents of a percept store. This does not mean that these conditions must be supported in the programming language itself, but only that it should be possible to somehow inspect which percepts have been received.

In order to support various options for percept processing, we distinguish four percept types and associate specific test templates with each type. Although in theory alternatives can easily be conceived of, we have specified test templates below based on the assumption that the percept information needs to be made persistent in the agent's belief state, which worked well in practice. We also assume that test conditions for percepts can be associated with a percept processing module called *ppm*, and we can write test statements of the form `test` *ppm* `with in` { χ^+ }. Each agent language provides some kind of support for this, although perhaps not in the language itself but 'under the hood'. It is important that test conditions for percepts are associated with and evaluated while executing the percept processing module. Because this module is executed once each cycle of the agent, in order to not violate the test conditions, percepts must have been processed and beliefs updated accordingly at the end of percept processing.

Template *P-once*: concerns percepts that are only received *once*, typically when the agent is launched to inform about static information such as locations on maps. The test template expects that after receiving the percept, it will be made persistent such that the agent believes it.

$\mathsf{percept}(p)$ `leadsto` $\mathsf{bel}(p)$

Template *P-always*: concerns percepts about facts p that are *always* received when p is the case, meaning that not receiving the percept implies that p does not hold. The test template consists of two test conditions. The first is identical to the one for ***P-once***; the second condition expects that not perceiving p, which would indicate that p does not hold, should lead to removing the belief p (if present).

$\mathsf{percept}(p)$ `leadsto` $\mathsf{bel}(p)$
$\mathsf{not}(\mathsf{percept}(p)) \wedge \mathsf{bel}(p)$ `leadsto` $\mathsf{not}(\mathsf{bel}(p))$

Template *P-on-change*: concerns percepts that are sent only when parameters change. A percept $loc(place)$, for example, is sent each time when an agent's location changes.

$\mathsf{percept}(p(\vec{t}))$ `leadsto` $\mathsf{bel}(p(\vec{t}))$
$\mathsf{percept}(p(\vec{t})) \wedge \mathsf{bel}(p(\vec{t'})) \wedge \vec{t} \neq \vec{t'}$ `leadsto` $\mathsf{not}(\mathsf{bel}(p(\vec{t'})))$

Template *P-on-change-with-negation*: concerns percepts p that are received once when p becomes true, and percepts $not(p)$ that are received once when p becomes false (again). For example, $in(room)$ is received when an agent enters a room, and $\mathsf{not}(in(room))$ is received when it leaves again.

$\mathsf{percept}(p)$ `leadsto` $\mathsf{bel}(p)$
$\mathsf{percept}(\mathsf{not}(p))$ `leadsto` $\mathsf{not}(\mathsf{bel}(p))$

G1: Failure to add a goal that should be added.

The taxonomy in Fig. 1 includes five failure categories related to goal handling. We therefore assume that a state

operator goal is available for checking for the presence or absence of a goal. Each of these failure categories, with the exception of *G4*, suggest that a *reason* for (not) having a goal has not been adequately taken into account.

Template *G-adopted*: concerns a goal ϕ that the agent is expected to adopt because of some (sufficient) reason ψ.

ψ leadsto goal(ϕ)

G2: Failure to drop a goal that should be dropped.

The failure to drop a goal is taken here as an indication that the agent did not adequately reconsider the goals that it has. As one would expect an agent to reconsider its goals if the environment has changed outside the control of that agent, these failures would most likely only occur in dynamic environments or in a multi-agent context.

Template *G-reconsideration*: concerns a goal that should be reconsidered and dropped as a result for some reason ψ.

ψ leadsto not(goal(ϕ))

G3: Adding a goal that should not be added.

The counterpart of *G1*, reflected by the fact that we use a safety (never) instead of liveness (leadsto) condition here.

Template *G-incorrect*: concerns a situation in which there is a reason ψ for not adopting (having adopted) the goal.

never goal(ϕ) $\wedge$ ψ

G4: Adding a second goal of the same type.

Some goals should only occur once and it should never be the case that the goal is instantiated twice. For example, an agent might have a goal of visiting a room but should never have two of those goals simultaneously.

Template *G-duplicate*: concerns a single-instance goal that should be instantiated at most once.

never goal($\phi(\vec{t})$) $\wedge$ goal($\phi(\vec{t'})$) $\wedge$ $\vec{t} \neq \vec{t'}$

G5: Dropping a goal that should not be dropped.

Similar to *G3*, *G5* is the counterpart of *G2*.

Template *G-maintain*: concerns a situation in which ψ is a reason why an agent should have a goal, and should maintain it for that reason.

never not(goal(ϕ)) $\wedge$ ψ

A1: Failure to select an action.

Fig. 1 includes four failure categories related to actions. Categories *A1* and *A3* suggest that a *reason* for (not) selecting an action has not been adequately taken into account.

Template *A-selected*: concerns an action α that the agent is expected to have selected because of some reason ψ.

ψ leadsto done(α)

A2, A3: Incorrect action (selection).

We provide one test template for categories $A2$ and $A3$, which both suggest that something happened that should (never) have happened, and thus can be viewed as the counterpart of $A1$. $A2$ indicates that beliefs should not have been updated the way they are, and $A3$ indicates there is a situation in which an action should never have been selected.

Template *A-incorrect*: concerns an action α that should never be (immediately) followed by ψ ($A2$), or a situation ψ in which an action should never have been selected ($A3$).

never done(α) $\wedge$ ψ

A4: Action interface mismatch.

This failure concerns an agent that attempts to perform an action in an environment that is not supported by that environment. An environment is expected to raise an exception or use another event-mechanism to indicate this issue. To detect these failures, rather than providing a test template, the test framework should abort testing immediately after receiving the exception or event. Aborting upon receiving an exception is a general principle that is supported in our framework.

4.2 Taxonomy Refinement

The test templates that we have introduced do not offer a perfect match with the failure categories, but provide a refinement of the taxonomy in Fig. 1. Instead of two categories *P1,P2* for percept processing failures, we introduced four templates (***P-once***, etc.). The main reason for this difference is that in a test approach we should specify what kind of percept is expected, instead of indicating that no processing was done at all (*P1*) or something went wrong (*P2*). For a similar reason, we renamed *A1* from 'selecting the wrong action' to 'failure to select an action', which also highlights the similarity with the template for *G1*. We merged *A2* and *A3* which are covered by a single test template; note again the structural similarity with the template for ***G-incorrect*** (*G3*). As discussed, we do not have a corresponding template for *A4*, as tests should rather be aborted when this failure happens. Finally, we have no templates for the 'other' category; our empirical results suggest there is no need for an additional template, as we show below.

4.3 Test Approach

In order to test and use the test templates, we need a *systematic test approach* that tells us what to do (steps) and, more specifically, provides clues on how to instantiate the templates for a specific application. An important question, for example, is how to find reasons to instantiate the ψ conditions that occur in the ***G***- and ***A***-templates. Table 2 lists information resources that are particularly useful for testing.

Source	Type of Information
Agent program (comments)	Clues for reasons & design
Agent trace (screen, logs)	Observable behaviour
Agent design & specification	Functional requirements
Environment (documentation)	Percepts, actions available

Table 2: Information sources for testing

Step 1: Defining success.

The first step is to identify functional requirements from available agent design documentation (Table 2). These requirements define success and provide a concrete method for checking that a program does what it is supposed to do. A program can be considered free of failures if it meets requirements. In order to automatically check this, functional requirements must also be specified in the test language. Typically, these requirements will be associated with

the top-level module, called `main` here, that is executed by the agent, and we can specify them as pre-, post-, or in-conditions of that module using `test main with` statements or by using a statement `do main until` ψ. The latter is particularly useful for checking that some overall objective ψ is realized (and, if so, the test will be automatically terminated; a `timeout` should be specified to guarantee termination if this is not the case). For example, the requirement or objective to pickup and deliver n packages, which was used in [27] and below will be used to reproduce results, can be specified by `do main until` bel($delivered(n)$).

Step 2: Testing cognitive state updating.

It is important to test that updating of an agent's cognitive state works as expected. The state conditions used in test conditions are evaluated on this state and will fail for unclear reasons when state updating has not been implemented correctly. For example, a condition `never` done($putDown$) $\wedge$ not(bel($in(Room)$)), which expresses that a package should never be put down when not in a room, could simply fail because beliefs about *in(Room)* were not updated correctly.

Identify percepts, actions, and goals used As a preparatory step, it is useful to collect and list all percepts, including their type (*once*, etc.), that may be received and the actions that may be performed from environment documentation (Table 2). Similarly, all goals that an agent may have should be collected from the agent program code.

Validating percept processing The most basic step is to instantiate the appropriate test templates ***P-once***, etc. for each percept according to type. The resulting test conditions should be associated as in-conditions with the percept processing module, and the test framework used to evaluate these conditions. Tests should be repeated sufficiently often as percepts generated will differ per run, if only because environments are more often than not non-deterministic. To gain confidence that percepts are correctly handled, it is important to check against the list of actions created above whether a sufficient variation of actions has been performed during runs, as different actions often yield other percepts.

Check single-instance goals Based on design, and intended use of goals in comments in a program (Table 2), for example, and using the goal list created, the subset of goals that are single-instance goals should be identified. For each of these goals, the test template ***G-duplicate*** should be instantiated and associated as an in-condition with the module where the goal is adopted.

We note that Step 2 can be performed by almost mechanically instantiating templates once relevant information has been collected. If these initial tests succeed, therefore, this will give a high level of confidence that cognitive states are updated correctly. For all templates other than those used above, however, a state condition ψ needs to be derived using insights gained from the design and behaviour of an agent.

Step 3: Classifying failures.

After testing state updating and checking that failures still are present (see Step 1), there are two approaches for classifying those failures. A bottom-up approach would start with the action and goal list created and try to instantiate templates for all these actions and goals, using the agent program itself as the main source of information. A top-down approach would rather start by analysing agent traces (Table 2) and observing agent behaviour in order to identify which action should (not) have been performed or goal should (not) have been added. Although a top-down approach suggests to start with testing goals, this order is not fixed and testing might just as well proceed with actions.

Failures concerning actions In order to instantiate ***A***-templates, apart from the action, a state condition should be identified that provides a reason ψ for (not) selecting it. For ***A-selected*** (resp. ***A-incorrect***), the question is in which situations ψ an action should (never) be executed. The instantiated conditions should be associated as in-conditions with the module(s) where the action might (not) be selected.

There are two basic approaches for identifying ψ:

1. By inspecting the agent program, clues may be obtained for useful test conditions ψ. In particular, the triggering conditions of rules can be useful, as they typically indicate reasons for selecting an action. For example, a condition bel($in('DropZone') \wedge holding(Block)$) that triggers execution of an action $putDown$ suggests that an agent should execute $putDown$ when it is holding a block in the $'DropZone'$. By simply using this condition for ψ we can instantiate ***A-selected*** as follows: bel($in('DropZone') \wedge holding(Block)$) `leadsto` done($putDown$). This simple approach can already detect failures, e.g., in case the rule order prevents the rule for $putDown$ to ever be applied. Similarly, by simply negating conditions found in a program, we can instantiate ***A-incorrect***. This assumes that the condition must hold for the action to be selected. Although these assumptions may be too strong, they provide a useful starting point and can be weakened or strengthened based on further analysis.

2. If an action failure is suspected because, for example, a functional requirement is not satisfied, observing an agent's behaviour may provide clues for identifying a useful condition ψ for instantiating a test template for that action. Suppose a requirement bel($in(Room)$) `leadsto` not(bel($in(Room)$)) fails, i.e., an agent does not always leave a room after entering it. By observing that the *goTo* action is never performed, a failure to select this action is suggested. To test for this, the ***A-selected*** template can be instantiated by instantiating ψ with the reason for leaving and α with the *goTo* action, which yields: bel($in(Room) \wedge Room \neq OtherRoom$) `leadsto` done($goTo(OtherRoom)$). This process can be continued until the root cause has been identified.

Failures concerning goals The approach for instantiating ***G***-templates, apart from identifying the goal that might cause the failure, is similar to that for ***A***-templates. The associated questions for the templates are, for ***G-adopted***: for which ψ should the goal be added?; for ***G-reconsideration***: for which ψ should the goal be dropped?; ***G-incorrect***: for which ψ should a goal never be added?; and for ***G-maintain***: for which ψ should a goal never be removed. The approach for identifying ψ above also applies for goals. The instantiated conditions should be associated as in-conditions with the module(s) that are related to the goal.

We illustrate a test for a goal $in(Room)$ that is adopted in a rule with a triggering condition bel($room(Place)$) $\wedge$ not(bel($visited(Place)$)), i.e., indicating that an agent should adopt (multiple) $in(Room)$ goals when it knows about a room that it has not visited before. By simply using this condition for ψ we can instantiate ***G-adopted*** as follows: bel($room(Place)$) $\wedge$ not(bel($visited(Place)$)) `leadsto` goal($in(Place)$). Again, this simple approach can already detect failures, e.g., in case the rule order prevents the rule

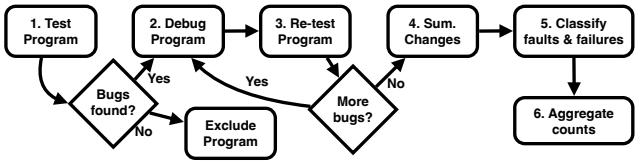

Figure 2: The methodology used in [27]

from ever being applied. Similarly, by simply negating conditions found in a program, we can instantiate ***G-incorrect***. This assumes that the condition must hold for the goal to be adopted, e.g., we never want to go to rooms we have visited before. A similar approach can be used for the ***G-reconsideration*** and the ***G-maintain*** templates.

5. EVALUATION

In this section, we first evaluate whether our automated approach is able to detect all failures in a specific set of agent programs. Next, we evaluate whether we are able to detect all failures in a different set of agents operating in the same environment, but with additional functional requirements. Finally, we evaluate this for a single agent program sample that operates in a different environment.

5.1 Comparing the Approaches

To establish the effectiveness of our automated test framework and test approach, we evaluate here whether the automated test framework, using the test templates introduced above, is able to detect the failures that were detected by manual inspection in [27]. We aim to show that our framework can be used to detect failures to obtain failure-free programs, meaning that functional requirements are met.

Methodology We used the agent program sample and the methodology for testing, debugging, and classifying failures of Fig. 2 that was also used by [27]. Only step 1 and 3 were replaced by tests performed using our test framework instead of manual inspection of logs and traces. We also used the same 'success criteria', i.e., the requirement formulated at the end of Step 1 of our approach. As we want to demonstrate that failures detected in [27] can also be detected automatically, we used a bottom-up approach for writing tests and created test templates for all percepts, actions, and goals in a program to ensure as large a coverage as possible. Upon detecting a failure (bug), we applied the same fixes as [27], which we obtained from the author, and re-ran our tests. We verified that the same bug was found by our tests, e.g., by checking that the code location where the test was terminated corresponds with the code location where a fix was applied. If we could not match a failure found by automated testing with one found originally by manual inspection, this failure was registered separately, the corresponding test conditions were removed, and the evaluation was continued.

Results We analysed and wrote a large set of test conditions for 20 agent programs. The results in terms of number of failures found are summarized in Table 3. Each individual failure found was counted and included in the Table. Note that we used taxonomy of Fig. 1 for classification, and not our refinement, to allow for comparison. We were able to reproduce every failure found by [27] ('Orig.' column in Table 3) and often found more ('Difference' column). In particular, we were able to automatically verify that all programs were failure-free after fixing buggy programs by testing that they met the requirement, indicating success.

We were able to detect 88% of the original failures by performing automated tests. So even though we were able to fully automatically establish that a program was failure-free, detecting some problems that were originally classified as failures required manual inspection (counts between brackets in Table 3). After analysis, it turned out that all these failures only occur when agent's cycles are delayed indefinitely while an environment may continue running. This can happen when a developer manually pauses an agent. As an example, an agent that runs at normal speed and is connected to the environment BW4T [17], which we used in our sample, would never simultaneously receive the percepts not(*in(room(a1))* and *in(room(a2))*. This is because it takes time to move from *a1* to *a2*, and an agent running at normal speed would first receive the first and only later receive the second percept. If an agent is paused, because the environment continues running both percepts may be queued and received simultaneously when the agent is resumed. When we run tests with agents using the automated test framework, we do not detect such 'failures' because agents are never paused. Our finding thus highlights that it is important to take timing issues into account. We also note that no *G5* failures were found (confirmed after discussion with the author of [27]), but we did find such a failure in the sample we discuss in Section 5.

We identified 24% more failures, most of which (70%) were detected by means of the percept test templates (*P1*, *P2*) and the template ***G-duplicate*** (*G4*). Interestingly, these templates are used in Step 2 of our test approach. This suggests that manual inspection is biased more towards a top-down approach in Step 3. As we argued, developing and testing a program is facilitated, however, if confidence is gained first that state updating is free of failures. One likely other reason that explains why we found more failures is that we were able to perform many test runs, as the test framework facilitates performing repeated test runs, and each run may produce different failures because of non-determinism.

A particularly interesting result is that we found that 63 from the 82 (77%) failures detected by automated tests immediately pointed at the code location of the fault. This, of course, makes it easier for a developer to fix a bug, and is a clear indication of the value that automating testing can have for agent programming. Fault locations are hardly ever pointed at, however, when a wrong action is performed (***A-selected***, *A1*), suggesting that locating faults for such failures may require a different approach.

We now show that our approach is also able to detect all failures in sample programs other than those used to reproduce the results of [27]. The test templates that we proposed have been based on the failure taxonomy of [27] which was derived from an analysis of the sample used in the previous section. We aim to provide additional support that shows that our set of test templates is complete and our approach is not biased towards a particular sample by looking at other sample programs.

5.2 Different Agent Program Sample

We selected a new sample of 10 agent programs that were written for the BW4T environment but by different programmers. Moreover, these programs were supposed to satisfy several more functional requirements instead of only one.

Failure	Manual [27]	Automated	Difference
P1	14	17 (3)	3
P2	11	16 (1)	5
A1	29	30 (5)	1
A2	3	4 (0)	1
A3	4	5 (0)	1
A4	1	1 (0)	0
G1	5	5 (0)	0
G2	3	3 (0)	0
G3	7	10 (1)	3
G4	5	11 (0)	6
G5	0	0 (0)	0
Totals	82	102 (10)	20

Table 3: Comparison of results (5.1)

Failure	Count
P-failure (P1,P2)	10
A-selected (A1)	11
A-incorrect (A2,3)	7
G-adopted (G1)	4
G-reconsideration (G2)	1
G-incorrect (G3)	16
G-duplicate (G4)	3
G-maintain (G5)	1
Total	53

Table 4: Results of the performed evaluation (5.2)

Agents were required, for example, to not do any redundant tasks: "An agent should go directly to the drop zone when it is holding a block", formalized as $\texttt{never done}(goTo(Room)) \wedge Room \neq 'DropZone' \wedge \texttt{bel}(holding(Block))$, and "An agent should not go to the drop zone without holding a block", formalized as $\texttt{never done}(goTo('DropZone')) \wedge \texttt{not}(\texttt{bel}(holding(Block)))$. These additional requirements also pose a greater challenge for our test approach, as a program is considered failure-free only if all requirements are satisfied, increasing the difficulty of detecting all failures.

All failures found could be classified using our refined taxonomy; Table 4 presents counts per type. We did not find any *A4* failures but did find a ***G-maintain*** (*G5*) failure. Failures that were detected were fixed using the same approach as [27]. We thus were able to successfully generate failure-free agents that satisfy all functional requirements.

A larger set of functional requirements facilitated the use of a top-down approach (Section 4) rather than a bottom-up approach, as was used for reproduction. An interesting finding is that this lead to a decrease in the amount of test failures that pointed directly at code locations of corresponding faults (25%, 13 out of 53, as opposed to 77% before). More work is needed to explain this finding.

5.3 Different Environment

Finally, we performed an evaluation whether the test framework and approach would detect failures in a single agent program sample that operates in a different environment. We chose an environment called the Vacuum World to this end [8]. In this world, squares in a grid can be either clean, dusty, or contain an obstacle that the robot should move around, thus requiring the agent to also successfully navigate the robot over the grid. A robot is able to move to any neighbouring clean or dusty square and clean up (vacuum) the square it is currently on (removing the dust).

For the evaluation, we used an agent program sample that was developed to meet the requirement that all dusty squares in a grid have been cleaned. This initial agent program did not meet the functional requirement specified, and we thus applied our test approach to detect failures.

Four failures were detected in the agent program, which is similar to the average of about four or five failures that were found for all other sample programs. Two percept failures were detected using the ***P*** templates, and two failures related to action selection were detected using the ***A-selected*** template. After fixes were applied for these failures, the agent did meet its requirement. Consistent with earlier results, percept failures indicated code locations for corresponding faults, whereas the action failures required more debugging effort for fault localization.

6. CONCLUSION

In this paper, we proposed and defined an *automated testing framework* for cognitive agent programs, facilitating automated failure detection and reducing debugging effort that is required from a developer. We argue that modules are a natural unit for testing, and associate test conditions with modules of an agent program. We also introduced a test language that we used to specify test templates for detecting failure types. These test templates refine a failure taxonomy introduced in [27]. A test approach has also been specified that explains how to instantiate test templates and derive test conditions for specific failure types. The main steps of this approach are (i) to define success in terms of functional requirements, (ii) to test cognitive state updating, and (iii) to classify failures that concern actions and goals.

The test language proposed is minimal in the sense that only two temporal operators are provided. We showed by analysing different agent program samples that the language is nevertheless sufficient for detecting all failures in these programs. In particular, we were able to reproduce and detect all failures that were manually identified in [27] using our automated test framework. Interestingly, in about 77% of failures found in this reproduction, the test framework also pointed to the code location of the corresponding fault. We demonstrated that our approach is not biased towards a specific sample of agent programs by applying the framework to other sample programs, and in a different environment. We were able to detect all failures by means of the automated test framework, i.e., all agents eventually met all functional requirements after fixing the detected failures.

Our work points to several issues that need more work. The focus of our work has been on automatically detecting failures. Even though our results are encouraging in that fault localization was facilitated by the test framework, more work is needed for locating faults that correspond with these failures. In particular, we found that faults related to actions that are performed but should not have been performed are difficult to locate. Tools that can explain why these actions were performed might be useful here [13].

Finally, further evaluation of the test framework is needed in more environments. Because of non-deterministic environments and other agents that act in the same environment, failures that occur in one run may not occur in the next. More work is needed to identify how often a test should be repeated in order to find all failures in an agent program. Measures of the variation one should expect in a given environment could be useful here.

Acknowledgments

We would like to thank Michael Winikoff for sharing his experiment data and providing valuable insights on this work.

REFERENCES

[1] R. Bordini, M. Dastani, and M. Winikoff. Current Issues in Multi-Agent Systems Development. In *Engineering Societies in the Agents World VII*, volume 4457, pages 38–61. 2007.

[2] R. H. Bordini, L. Braubach, J. J. Gomez-sanz, G. O. Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

[3] T. Bosse, C. M. Jonker, L. van der Meij, and J. Treur. Leadsto: A language and environment for analysis of dynamics by simulation. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. N. Huhns, editors, *Multiagent System Technologies*, volume 3550 of *Lecture Notes in Computer Science*, pages 165–178. Springer Berlin Heidelberg, 2005.

[4] G. Caire, M. Cossentino, and A. Negri. Multi-agent systems implementation and testing. In *Proc. of the 4th From Agent Theory to Agent Implementation Symposium, AT2AI-4*, 2004.

[5] A. Carrera, C. A. Iglesias, and M. Garijo. Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. *Information Systems Frontiers*, pages 1–14, July 2013.

[6] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems - SELMAS '06*, page 83, New York, New York, USA, 2006. ACM Press.

[7] R. Collier. Debugging agents in agent factory. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 4411 of *Lecture Notes in Computer Science*, pages 229–248. Springer Berlin Heidelberg, 2007.

[8] R. Collier and J. Howell. Vacuum world. `https://github.com/eishub/vacuumworld`. Accessed: 2015-11-17.

[9] M. Dastani. Programming multi-agent systems. *The Knowledge Engineering Review*, 30:394–418, 9 2015.

[10] J. Dix, K. V. Hindriks, B. Logan, and W. Wobcke. Engineering Multi-Agent Systems (Dagstuhl Seminar 12342). *Dagstuhl Reports*, 2(8):74–98, 2012.

[11] E. Ekinci, A. Tiryaki, v. Çetin, and O. Dikenelli. Goal-oriented agent testing revisited. In M. Luck and J. J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *Lecture Notes in Computer Science*, pages 173–186. Springer Berlin Heidelberg, 2009.

[12] J. Gómez-Sanz, J. Botía, E. Serrano, and J. Pavón. Testing and debugging of MAS interactions with INGENIAS. In *Agent-Oriented Software Engineering IX*, pages 199–212, 2009.

[13] K. V. Hindriks. Debugging is explaining. In I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara, editors, *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, volume 7455 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2012.

[14] Z. Houhamdi. Multi-Agent System Testing: A Survey. *International Journal of Advanced Computer Science and Applications*, 2(6):135–141, 2011.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, Dec. 2004.

[16] ISO. ISO/IEC/IEEE 24765:2010 systems and software engineering - vocabulary. Technical report, Institute of Electrical and Electronics Engineers, Inc., 2010.

[17] M. Johnson, C. Jonker, B. van Riemsdijk, P. J. Feltovich, and J. M. Bradshaw. Joint activity testbed: Blocks world for teams (bw4t). In H. Aldewereld, V. Dignum, and G. Picard, editors, *Engineering Societies in the Agents World X*, volume 5881 of *Lecture Notes in Computer Science*, pages 254–256. Springer Berlin Heidelberg, 2009.

[18] M. A. Khamis and K. Nagi. Designing multi-agent unit tests using systematic test design patterns-(extended version). *Engineering Applications of Artificial Intelligence*, 26(9):2128–2142, Oct. 2013.

[19] V. J. Koeman and K. V. Hindriks. Designing a source-level debugger for cognitive agent programs. In Q. Chen, P. Torroni, S. Villata, J. Hsu, and A. Omicini, editors, *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, volume 9387 of *Lecture Notes in Computer Science*, pages 335–350. Springer International Publishing, 2015.

[20] M. Moreno, J. Pavón, and A. Rosete. Testing in agent oriented methodologies. In S. Omatu, M. P. Rocha, J. Bravo, F. Fernández, E. Corchado, A. Bustillo, and J. M. Corchado, editors, *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, volume 5518 of *Lecture Notes in Computer Science*, pages 138–145. Springer Berlin Heidelberg, 2009.

[21] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah. Testing in Multi-Agent Systems. In *Agent-Oriented Software Engineering X*, volume 6038, pages 180–190. Springer Berlin Heidelberg, 2011.

[22] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *Software Engineering, IEEE Transactions on*, 39(9):1230–1244, Sept 2013.

[23] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[24] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *Software, IEEE*, 23(3):82–90, May 2006.

[25] A. M. Tiryaki, S. Öztuna, O. Dikenelli, and R. C. Erdur. SUnit: a unit testing framework for test driven development of multi-agent systems. In *Proceedings of the 7th international conference on Agent-oriented software engineering VII*, pages 156–173, 2007.

[26] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12:67–96, 2012.

[27] M. Winikoff. Novice programmers' faults & failures in goal programs. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 301–308, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.

[28] Z. Zhang, J. Thangarajah, and L. Padgham. Model based testing for agent systems. *Software and Data Technologies*, 22:399–413, 2008.