An algorithmic debugging approach for BDI agents

Tobias Ahlbrecht[0000−0002−4652−901X]

Department of Informatics, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
tobias.ahlbrecht@tu-clausthal.de

Abstract. Debugging agent systems can be rather difficult. It is often noted as one of the most time-consuming tasks during the development of cognitive agents. Algorithmic (or declarative) debugging is a semi-automatic technique, where the developer is asked questions by the debugger in order to locate the source of an error. We present how this can be applied in the context of a BDI agent language, demonstrate how it can speed up or simplify the debugging process and reflect on its advantages and limitations.

Keywords: debugging · bdi · agentspeak · cognitive agents.

1 Introduction

Debugging is a notoriously difficult task. This appears to hold even more for cognitive agent programs and multi-agent systems. Participants of the Multi-Agent Programming Contest regularly note that debugging is one of the most time-consuming tasks involved in building their agent systems [1,2]. Judging from reports and observations, many agent developers resort to basic debugging methods, like simple logging statements, to make sense of what is going on. Existing agent platforms mostly provide support for inspecting an agent's mind state. A number of different debugging techniques have been proposed, though.

Koeman et al. [11] present a debugger for GOAL [8] which only stores the changes to the agent's state in each step, thereby enabling the reconstruction of the state at any point in time with minimal space requirements and negligible impact on performance. Thus, this allows to debug an agent without having to reproduce the circumstances that lead to a bug, which is usually extremely difficult. In our work, we will use a similar logging scheme in order to debug previous agent executions, during which something unexpected has been observed.

Poutakidis et al. [12] show that information from the design process can be leveraged e.g. for generating test cases or general debugging support. For now, we will be working with the information that is always available regardless of the design process used, i.e. the agent code and one of its traces. However, information from the design process may be able to improve the new debugging process later as well.

Others provide debugging assistance by the means of new kinds of visualisations, e.g. Botía et al. [4] employ clustering techniques to give an overview of agent communication. AS we will have the agent programmer check if certain agent states are valid with regard to specific questions, having a concise representation of an agent's state will also play a bigger role in the future of our approach.

Hindriks [9] and Winikoff [17] show that asking "Why?" questions is also debugging and can help with locating the source of a bug. For example, the programmer may ask "why did the agent execute this action?" and depending on the answer, ask more questions, going back through the execution trace until the bug's origin is located. These approaches differ from this paper in that the programmer has to navigate the traces by asking the right questions, while we will try to automate this navigation with *algorithmic debugging*.

Winikoff suggests in [17] to take a look at this *algorithmic debugging* and its possible combination with the approach based on "Why?" questions. Algorithmic debugging is another form of question-based debugging, only the questions are asked *by* the debugger this time, which makes for a semi-automatic process. The debugger is in charge of navigating the trace, while the programmer has to validate results of computations. Algorithmic debugging, originally envisioned in the context of Prolog programs [16], has been tried for many languages so far, e.g. Java [6] or Erlang [7], yet not in the context of cognitive agents.

2 Background

2.1 BDI and AgentSpeak

The belief-desire-intention model [5,14] provides concepts for a reasoning process that selects suitable actions to reach specific goals. A BDI agent has

- *beliefs* - certain things the agent believes to be true about the world,
- *desires* - states the agent wants to bring about, and
- *intentions* - representing the *current* aims of the agent together with actionable steps the agent will be doing in order to achieve them.

Often, the term *goal* is used for a desire the agent is actively pursuing at the moment and *plans* are a common way of specifying recipes for how to achieve a goal in different situations.

In this paper, we are mainly concerned with AgentSpeak [13], in particular Jason[3], but the approach seems general enough to work with any BDI platform based on goals and plans.

In AgentSpeak, each plan consists of a *trigger*, i.e. an event it is meant to handle, a *context* in which it is applicable as determined by the agent's currents beliefs and a *body*, which is a sequence of steps that are supposed to reach the goal. Thus, *events* may trigger a new intention, for which an applicable plan (where the context fits the current situation) is selected among all relevant plans (which match the given event).

2.2 Bugs in BDI agents

When debugging, we typically differentiate between symptoms and reasons or causes. A symptom indicates that a bug is present in the code, while the reason for the bug probably is some code fragment that has been executed at some point (possibly way) before.

For BDI agents, we can try to find a symptom using the abstraction of goals by asking (at least) these four questions that come to mind when thinking about goal adoption and achievement.

1. Was it correct to add a particular goal?
2. Has the goal been achieved?
3. Did (pursuing) the goal only produce desired side effects?
4. Has each expected goal been adopted?

If the answer to one of these questions is "no", we have found at least a *symptom* of the bug, yet probably not the *reason*. If the answer is always "yes", we know that exactly the right goals were adopted and achieved without undesired side effects.

The approach presented later will mostly rely on symptoms related to the first two questions. After the exact sub-goal that contains a bug has been pinpointed, it will be necessary to ask further questions about the selection of plans and the actual actions that were performed to narrow down the problematic statements. Thus, further techniques will be required for a comprehensive debugging toolkit.

2.3 Algorithmic debugging

Algorithmic debugging was first proposed by Shapiro in 1982 [15,16] for logic programming. The idea is to locate the source of errors by comparing the intended behaviour of a program with the results of an actual computation. Once an unexpected or wrong result has been observed, a tree is constructed with this initial computation as root node. The children of each node are its sub-computations, forming the *debugging tree* (sometimes called execution tree) which therefore reflects the structure of the computation. In procedural languages, this is e.g. comparable to a (chronological) run-time call graph. A computation and its actual result are then presented to an oracle (usually the user) in order to validate this node, starting at the root of the tree and continuing according to the user's answers. Since the user is guided through the debugging session by the debugger asking questions, this makes for a semi-automatic technique. If a node is found to be invalid, the source of the bug must be somewhere in the sub-tree below this node, but not necessarily in the node itself. The procedure is thus repeated for the node's children. If a node is valid, the process continues with its sibling instead until an invalid node is found that does not have any invalid children. Then, it can be concluded that the bug must be located in this node.

The process can be optimised by employing different *navigation strategies*, that decide which node to select for validation at which point. In some cases, it might be advantageous to let go of the chronological order of computations

(which is usually easier for the user to follow) in favour of evaluating nodes sooner, which are more likely to contain the bug, e.g. if they have many sub-computations or a large sub-tree.

Other languages that algorithmic debugging has been applied to include Erlang [7] or more common languages like Java [6].

3 Algorithmic debugging and BDI agents

The idea behind algorithmic debugging is to leverage the tree of computations and sub-computations to locate a bug with as few questions as possible. In procedural programming, functions or methods constitute the nodes of the debugging tree. Then, functions called inside a function make the children nodes of that function's node. If the user finds that a node is invalid (i.e. its result is wrong), debugging continues with the node's children. Once an invalid node is found which has only valid children, the reason for the bug must be located in this node.

The question is now of course how to construct the debugging tree for BDI agent executions, i.e. what are the computational units that we can use? In essence, we need a tree of computations that we can navigate and where the results of each computation can be verified by the user.

The entity in the BDI model most closely resembling a function is a *plan*. Plans are sequences of steps, which may include sub-goals that lead to instantiation of further plans. This already gives us a tree structure of computations that we can use for algorithmic debugging: a tree of instantiated plans[1]. The differences are mainly

- the particular plan that is "called" for a sub-goal is dependent on the context at that time, and
- the "expected result", i.e. the goal to be achieved, is already to some degree known when the sub-goal is posted[2]

In procedural languages, if a function has produced the correct result, the bug can not be in one of the function instances it has called. To apply algorithmic debugging to BDI systems, we are now making a similar assumption: If a goal is reached, there is no bug in any of its sub-goals. Conversely, if a goal is not reached, the bug is either in the plan selected for this goal or one of its sub-goals further down in the tree.

Once the tree is established, the user will be asked to validate nodes by confirming (or denying) that the associated goal was actually reached. To allow for this validation, the user is presented with the agent state at the moment the

[1] It is not a tree of intentions, as those usually reflect the current progress. It is also not really a Goal-Plan-Tree, since each goal is only associated with the first plan that has been selected for it.

[2] To be more precise, usually the sub-goal generates an event and the concrete goal including all parameters is known once the event has been processed.

last step of the goal's plan had been processed. Additionally, the user is allowed to navigate back and forth through the agent mind's history.

As per the second point above, we can improve this by adding a second question for node validation that asks if it was correct to add this goal in that situation. If the answer is "no", the bug is already located, namely, adding this goal at that point in time.

In AgentSpeak, triggering events that are not caused by the agent make up the root nodes of possible debugging trees. These can be initial goals of the agent, belief modifications caused by the environment (i.e. percepts) and messages from other agents, though we do not consider multiple agents yet. Then again, algorithmic debugging can start with any goal or corresponding event further down in the tree as well.

Finally, it should be noted that the computation that is to be debugged is usually rerun by a debugger. However, since agent programs seldom lead to deterministic computations, we will instead collect the trace of the agent program, from which the debugging tree can be constructed just the same.

3.1 Example: Simple blocks world

To illustrate the debugging process, a simple but partly extended version of the "BlocksWorld for Teams" (BW4T)[10] scenario was used. The goal here is to find a block of a designated colour and deliver it to a specific room. If requested, the block has to be packaged in another room before delivery. The available actions are `goto`, to move to a room, `gotoBlock` to approach a block inside the room, `pickUp` to pick up a block and `putDown` to put it down again, `activate` to use the packaging room and `recharge` to charge the agent's battery. Especially packaging and recharging are additions to the BW4T scenario. The following (relevant) percepts are available:

delivered(Id) the last delivered task
task(Id, Colour) the current task and the requested colour
packaging means that blocks currently requested need to be packaged
place(R) for each room
energy(E) the agent's current energy level
atBlock(Bid) the agent is currently located at block `Bid`
colour(Bid, Colour) the block `Bid` is of the given colour
holding(Bid) the agent is holding the given block
packaged(Bid) the given block is packaged
at(R) the agent is in room `R`

The agent for this scenario is written in simple Jason code given in Figure 1. Some of the more trivial plans have been left out for readability.

The agent follows the simple plan to prepare itself by putting down a potential block, recharging to an acceptable level, finding and holding the required block, optionally packaging it, and finally delivering it.

The user now runs the agent program in a sample instance of the world. After some agent cycles, the environment reports a failed delivery. The agent program

```
+task(Id, Colour) <-
    !completeTask(Id, Colour).

+!completeTask(Id, Colour) <-
    !prepared;
    !holding(Colour);
    !processed;
    !delivered.

+!prepared : task(C) & holding(B) & not colour(B, C) <-
    putDown;
    !prepared.
+!prepared <-
    !charged;
    !reset.
+!reset <-
    .abolish(visited(_)).

+!charged : energy(MyEnergy) & MyEnergy < 80 <-
    recharge;
    !charged.

+!holding(Colour) : colour(Block,Colour) <-
    gotoBlock(Block);
    pickUp.
+!holding(Colour) <-
    !found(Colour);
    ?colour(Block, Colour);
    gotoBlock(Block);
    pickUp.

+!found(C) : not colour(_, C) & place(P) & not visited(P) <-
    goto(P);
    +visited(P);
    !found(C).

+!processed : packaging <-
    goto(packing);
    putDown;
    activate.

+!delivered <-
    goto(dropzone);
    putDown.
```

Fig. 1. Jason example agent for the BW4T (summary)

shows no error, though. Thus, the user knows the goal `!completeTask(t0, red)` was actually not achieved and starts the debugger on this agent and goal. The debugger presents the user with the debugging tree represented in Figure 2 and starts asking questions.

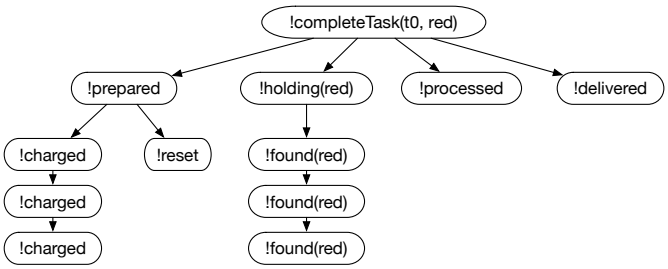

Fig. 2. Debugging tree for the BW4T example.

Since it was correct to *add* all goals that where added, we skip these questions in this description. First, the user has to answer whether the goal `!completeTask (t0, red)` has been reached. The user thus looks for the belief `delivered(t0)` (stemming from a percept) and since it is not there, answers "no". The debugger marks the node invalid and continues with its first child, `!prepared`. The user checks the `energy` belief and ensures the agent has no `holding(_)` belief. Both are satisfactory and the user answers "yes". The node is marked valid and the debugger continues with its sibling. To validate `holding(red)`, the user just checks the beliefs `holding(b5)` and `colour(b5,red)`, deducing that the goal was reached. The debugger marks the goal node as valid and the next sibling, `processed` is considered. The agent has the belief `packaged(block5)`, however, the belief `holding(b5)` has vanished. The user answers "no" and the node is marked invalid. Since this node does not have any children, the bug should be located in the corresponding plan. By inspecting the plan, together with the knowledge that the agent was not holding the block anymore, the user realises a missing `pickUp` action at the end of the plan, which is responsible for the failed delivery in the end. The tree after debugging is given in Figure 3.

This example is rather simple but shows how the debugger can locate the approximate source of a bug by asking simple questions. Compared to simply stepping through the agent's trace from its beginning, quite some effort can be saved by basically skipping over sub-goals of valid parent goals.

3.2 Debugger prototype

To acquire traces that can be fed to the debugger, a logger for Jason has been implemented that incrementally logs all relevant changes to an agent's state as

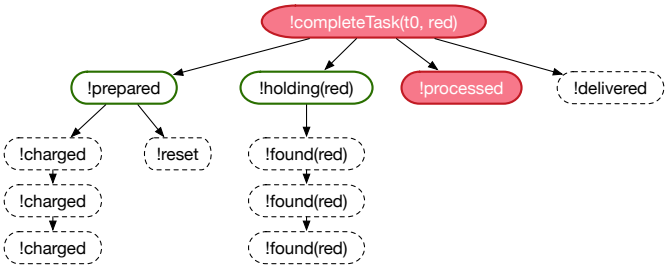

Fig. 3. Marked debugging tree for the BW4T example. Red nodes (filled) are invalid, nodes with a green border are valid. Nodes with a dashed border have not been visited.

inspired by [11]. It has been realised as a custom agent architecture[3], so that the Jason platform can be used as it is. Only a Java file and one line of code in the MAS declaration file have to be included for the logger to work. A prototype of the debugger[4] has been implemented in Python using the PyQT library to create the GUI. A sample wire frame (for the sake of readability) is given in Figure 4. On the left, the debugging tree is visualised with collapsible nodes. In the top right corner, the question about the current goal is asked. Below, the agent's state at the relevant time, i.e. its beliefs and intentions, are shown, allowing the user to check if the goal was reached and the adoption of the goal was correct.

One substantial part of the implementation is surely reading the log file, storing the data in appropriate internal data structures for quick access to plans, intentions, etc. and recreating any agent state on demand. Since the log file only contains changes to the agent's state, all cycles from the start of the execution to the requested step have to be worked through. So far, it has not been a performance concern, but in the future, caching mechanisms will be needed to speed up this process. Additionally, for very long agent runs it would be beneficial to save complete agent states once in a while, so that reconstruction can start at the nearest save point.

The debugger also needs a simple interface asking the programmer which goal they would like to debug. Currently, the prototype allows to select from either all intentions (i.e. only goals that have no other goal as parent), or from the set of all goals the agent adopted, or grouped by the plan that was chosen to achieve the goal.

The next bigger thing to implement is navigating the tree by determining which node has to be validated next and presenting the corresponding questions to the user. For now, the visualisation of the agent state corresponding to the questions is kept simple (a list of beliefs and intentions). Here is probably a lot of potential for helping the user to find the answer. For example, parts of the agent state that have not changed between the times the goal was adopted and

[3] The architecture as well as the BW4T example environment and agent can be found at https://github.com/t-ah/jason-util.

[4] The debugger can be found at https://github.com/t-ah/declarative-agent-debugger.

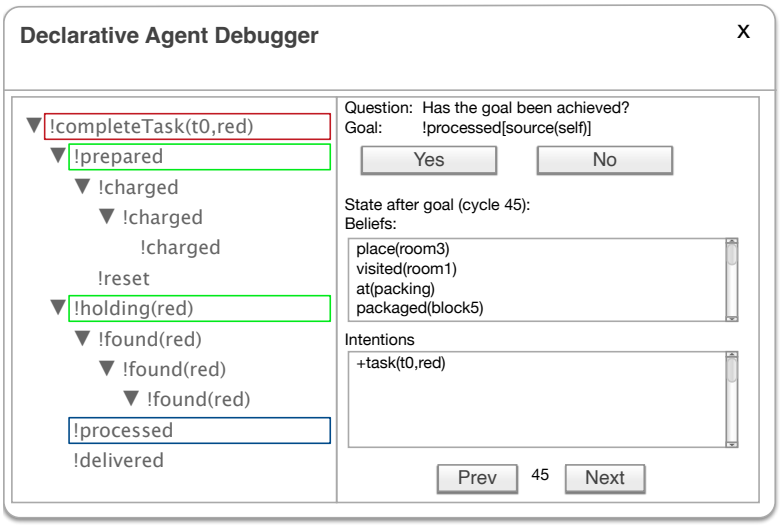

Fig. 4. A wire frame of the debugger

achieved are probably less important for many goals and could be displayed with lower priority.

3.3 Advantages and limitations

An advantage of algorithmic agent debugging is that the symptom can be either concrete or quite vague. E.g. a failed action will lead to a failed goal, which can be a symptom. But also just the user having observed "something wrong" with a goal can be an initial suspicion. In other words, the process does not have to start with a very detailed symptom, like a step of a plan that went wrong.

During implementation and testing of the debugger and trying to come up with examples, also some drawbacks and limitations of the approach were revealed. For one, certain agent programs may lead to the debugging tree just being a linked list. In this case, "algorithmic" debugging would just amount to stepping through the agent execution step by step from the start to the buggy node. Another problem could arise from the tendency to have a goal add itself again recursively until it is actually reached. In these kind of looping structures, algorithmic debugging also does not offer much of an advantage.

Earlier, we noted that the approach seems to generally work for any BDI platform based on goals and plans. This is due to the fact that the debugging tree is created from goals and their relation to their sub-goals. Some additional considerations may be necessary to have the tree span everything related to the same conceptual goal. For example, often the effect of an external action is not immediate. The programmer needs to account for this asynchronicity, e.g. by ending the plan after the action and having another plan ready that is to be triggered by the percept that is expected to result from the action. Without

deeper knowledge about the environment, the debugger cannot link these plans together, although they technically belong to the same course of action and should probably be debugged as such. Annotations may prove useful in bridging this gap in the future. Also, failure handling goes in the same direction. In Jason, plan failure generates a new event that can be handled by additional plans. As such, these are treated the same way as events related to new achievement goals and simply extend the debugging tree. If the failure handling mechanism of the platform instead were to repost the event that led to the failed goal, the question arises whether to treat the resulting goal as a top-level goal, thereby creating a new unrelated debugging tree, or if this new goal is to be treated as a sub-goal of the original one. In situations where the goal often fails and is reposted again and again, this could result in a very "long and narrow" tree, that is rather unhelpful for debugging. Having said that, the final iteration along with all previous attempts to achieve the same goal can be said to belong to the same course of action and therefore the potential bug is probably visible in one or more instances of the failed attempts. It might be a good compromise to shift this decision to the programmer and offer them both options. Depending on the actual structure of both trees, the debugger could even offer a recommendation on a case-by-case basis.

4 Conclusions and future work

We followed a straightforward approach of applying algorithmic debugging to BDI agent programs given in AgentSpeak. As a first result, it is possible to leverage the high-level concepts of goals and plans as a meaningful computational unit that can be used to build the debugging tree and leads to user-verifiable results. We have demonstrated the approach on a simple example and provided a proof-of-concept debugger.

Yet, there is still much to do. The current approach only allows to pinpoint the plan where a bug is located. Further methods are required to continue the bug search inside the plan. For example, debugging with "Why?" questions [17] could follow at this point to track down the exact buggy statement. Once this approach is implemented, it would be reasonable to allow the programmer to switch methods at any point, allowing them to move between more loosely related goals with "Why?" questions and possibly starting another session of algorithmic debugging on another goal that could not be identified with algorithmic debugging alone. Additionally, annotations could be used to allow the programmer to give additional information that would be helpful during debugging. For example, the exact purpose of a statement seems generally less obvious in agent programming than in conventional programming languages. An action could be used to induce some change in the environment, which is later required, maybe in the same plan or in the plan of a sub-goal, to execute another statement. If the programmer had linked these statements by an annotation, the debugger could later use this information, since the former statement can now be understood as a more likely reason for the later statement to fail.

Also, parallel intentions and the bugs that could arise from their complex interplay need more consideration.

Supporting the user validating the results can also be improved by giving better visualisations of the agent state and - if possible - its environment.

We mentioned earlier, that algorithmic debugging can be optimised with different navigation strategies. An interesting question would be how to determine goals and plans that are more likely to contain a bug. For example, if a statement in some plan failed, there is probably something wrong, which could be connected to the bug that the programmer is looking for. Thus, it is probably a good idea to have the programmer validate goals with failed statements earlier to reduce the number of questions.

Finally, we envision a debugger that offers multiple debugging techniques depending on the symptom encountered by the programmer. For example, another method for debugging goals, that were not pursued but expected, is required. This would also allow us to perform a comprehensive analysis with real agent programmers.

Acknowledgements

I thank the anonymous reviewers for their very helpful and insightful feedback.

References

1. Ahlbrecht, T., Dix, J., Fiekas, N., Krausburg, T.: The multi-agent programming contest: a résumé. In: Multi-Agent Progamming Contest. LNCS, vol. 12381, pp. 3–27. Springer (2019)
2. Ahlbrecht, T., Dix, J., Fiekas, N., Krausburg, T.: The 15th multi-agent programming contest. In: Multi-Agent Progamming Contest. LNCS, vol. 12947, pp. 3–20. Springer (2021)
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons (2007)
4. Botía, J.A., Hernansáez, J.M., Gómez-Skarmeta, A.F.: On the application of clustering techniques to support debugging large-scale multi-agent systems. In: International Workshop on Programming Multi-Agent Systems. pp. 217–227. Springer (2006)
5. Bratman, M.: Intention, Plans, and Practical Reason. Cambridge: Cambridge, MA: Harvard University Press (1987)
6. Caballero, R., Hermanns, C., Kuchen, H.: Algorithmic debugging of Java programs. Electronic Notes in Theoretical Computer Science **177**, 75–89 (2007)
7. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: Declarative debugging of concurrent Erlang programs. Journal of logical and algebraic methods in programming **101**, 22–41 (2018)
8. Hindriks, K.V.: Programming rational agents in GOAL. In: Multi-agent programming, pp. 119–157. Springer (2009)
9. Hindriks, K.V.: Debugging is explaining. In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 31–45. Springer (2012)

10. Johnson, M., Jonker, C., Riemsdijk, B.v., Feltovich, P.J., Bradshaw, J.M.: Joint activity testbed: Blocks world for teams (BW4T). In: International Workshop on Engineering Societies in the Agents World. pp. 254–256. Springer (2009)
11. Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Omniscient debugging for cognitive agent programs. In: 26th International Joint Conference on Artificial Intelligence, IJCAI 2017. pp. 265–272. AAAI Press (2017)
12. Poutakidis, D., Winikoff, M., Padgham, L., Zhang, Z.: Debugging and testing of multi-agent systems using design artefacts. In: Multi-agent programming, pp. 215–258. Springer (2009)
13. Rao, A.S.: AgentSpeak (L): BDI agents speak out in a logical computable language. In: European workshop on modelling autonomous agents in a multi-agent world. pp. 42–55. Springer (1996)
14. Rao, A.S., Georgeff, M.P.: BDI agents: from theory to practice. In: Proc. of the First Intl. Conference on Multiagent Systems (ICMAS-95), San Francisco. vol. 95, pp. 312–319. MIT Press (1995)
15. Shapiro, E.Y.: Algorithmic program diagnosis. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 299–308. ACM (1982)
16. Shapiro, E.Y.: Algorithmic program debugging. Yale University (1982)
17. Winikoff, M.: Debugging agent programs with Why? questions. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. pp. 251–259 (2017)