Annals of Mathematics and Artificial Intelligence (2024) 92:797–814
https://doi.org/10.1007/s10472-023-09843-4

An algorithmic debugging approach for belief-desire-intention agents

Tobias Ahlbrecht[1]

Accepted: 18 March 2023 / Published online: 5 May 2023

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 agents · 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

✉ Tobias Ahlbrecht
ta10@tu-clausthal.de

[1] Department of Informatics, Clausthal University of Technology, Clausthal-Zellerfeld, Germany

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. In the approach proposed in this paper, the agent programmer will have to check if certain agent states are valid with regard to specific question. Therefore, having a concise representation of an agent's state will also play a bigger role in the future of our approach, as it enables the programmer to more quickly assess if any agent state is as expected.

Hindriks [9] and Winikoff [18] 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*.

Algorithmic debugging, sometimes also called declarative debugging, was first proposed by Shapiro in 1982 [16, 17] 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.

Winikoff suggests in [18] to take a look at algorithmic debugging and its possible combination with his 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 [17], has been tried for many languages so far, e.g. Java [6] or Erlang [7], yet not in the context of cognitive agents.

In this paper, we show how algorithmic debugging can be applied to belief-desire-intention (BDI) agent programs, as showcased for the Jason agent framework. A prototype debugger is presented and the advantages and limitations of the approach are discussed.

In Section 2, a brief introduction to the BDI model, AgentSpeak(L), bug types in agents and general algorithmic debugging is given. Section 3 then entirely deals with the application of algorithmic debugging to belief-desire-intention agents. First, the main debugging procedure to locate a buggy goal is outlined and then an example is given in Section 3.1. In Sections 3.2 and 3.3, the process is then expanded to locate a buggy instruction inside a plan of a buggy goal. The connection to the approach based on "Why?" questions is elaborated in Section 3.4 and a debugger prototype is presented in Section 3.5. Finally, advantages and limitations are discussed in Section 3.6 and possible improvements to the application of the approach and the approach itself are considered in Sections 3.7 and 4.

2 Background

2.1 BDI and agentspeak

The belief-desire-intention (BDI) model [5, 14] provides concepts for a reasoning process that selects suitable actions to achieve specific goals. It allows to balance deliberation against means-end-reasoning, i.e. what to achieve and how to achieve it. 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.

The agent follows a *plan* to achieve its goal. Plans are usually predetermined recipes (given by the programmer) describing how to get to a specific state in different circumstances. These generic templates are filled in by the agent according to the specific situation. They could also come from a planner, which is not part of the model, though.

Often, the term *goal* is used for a desire the agent is actively pursuing at the moment.

In this paper, we are mainly concerned with AgentSpeak [13], in particular Jason [3], but the approach is 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 achieve 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. A symptom could have multiple reasons acting together. In that case though, it would generally suffice to find and remedy one of these reasons. On the other hand, such a reason could lead to many symptoms, any of which should lead back to the reason itself. In this work, we will assume the user has noticed a symptom of a bug and intends to locate the underlying reason.

Now we should define what we consider a bug within an agent. Intuitively, it could be anything from causing the agent program to crash, to leading to slightly less than optimal performance. Since the latter could identify almost anything as a bug, our definition will be somewhere in between.

We define a *valid agent* as one that adopts the "right" goals and achieves those, unless it is not possible in the current context. The right goals in this case mean those the user would have expected.

From this, we can define our notion of bug (symptom) as

1. some goal that was but should not have been adopted,
2. some goal that was not but should have been adopted, or
3. some goal that was adopted but not achieved (and the failure was not inevitable).

Please note that the third point somehow contains the case where a goal was technically achieved but had some additional undesirable side effects. Thus, we can read each goal to achieve X as "achieve X using appropriate means".

The approach discussed in this paper will mostly target the first and third point. To reference these, a specific goal that caused an actual computation can be used and analysed.

The second point above deals with some potential thing that needs to be placed in the actual computation first. For this, further techniques will be required for a comprehensive debugging toolkit.

2.3 Algorithmic debugging

Algorithmic debugging is a method that helps comparing intended with actual computations. This comparison is enabled by the debugger, which poses questions to the user, who is able to determine whether a computation has delivered the expected result.

An overview of this process is presented in Fig. 1. Clouds represent questions asked to the user, orange hexagons are the actions of the debugger, and boxes represent questions the debugger answers itself.

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 a different *navigation strategy* (represented by the "Select node" hexagon in Fig. 1), that decides 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

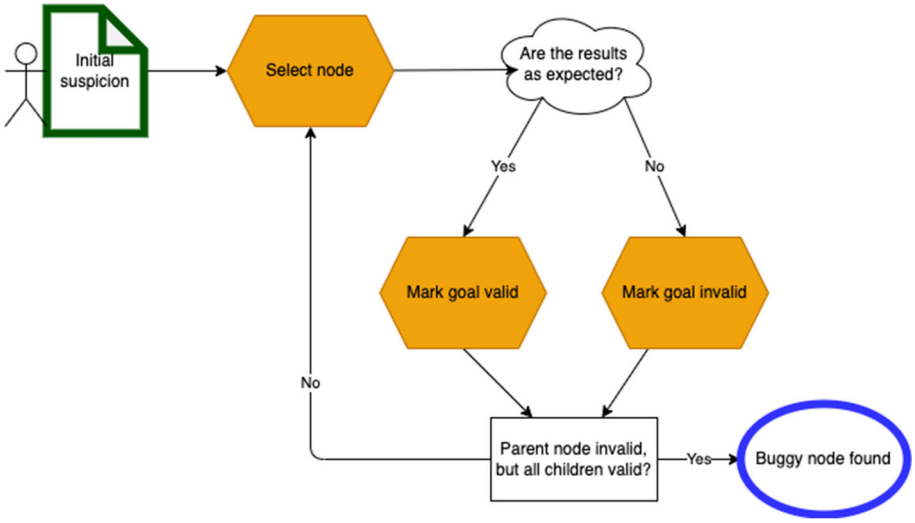

Fig. 1 Algorithmic debugging overview. Clouds represent questions asked by the debugger. Orange hexagons are its actions, boxes are decisions involved in the debugging algorithm

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 subgoals 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 subgoal 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 subgoal 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 achieved as expected by the user (i.e. without undesirable side-effects), we generally do not care for its subgoals (and the whole tree below) anymore.

Conversely, if a goal is not achieved, the bug is either in the plan selected for this goal or one of its subgoals 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 achieved. To allow for this validation, the user is presented with the agent state at the moment the 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.

The process is summarized in Fig. 2. Compared to Fig. 1, nodes are called goals, and the user is asked two consecutive questions to validate a node. There are also two distinct types

[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 subgoal generates an event and the concrete goal including all parameters is known once the event has been processed.

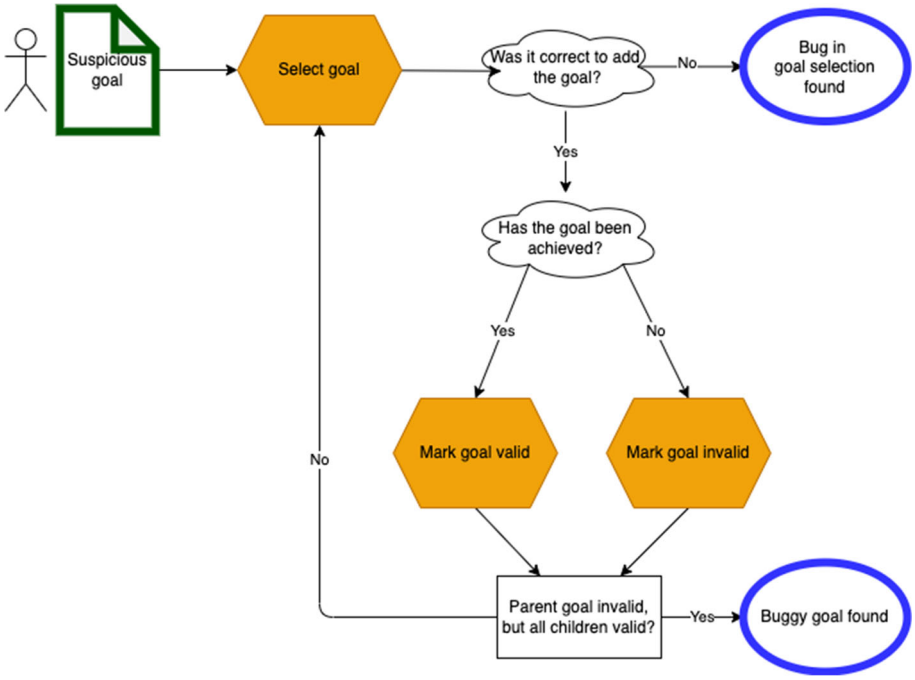

Fig. 2 Algorithmic debugging applied to AgentSpeak

of bugs that are differentiated, i.e. bugs in the selection of goals and bugs in the achievement of goals. When asked if a goal has been achieved, this again includes the possibility for the user to consider any possible undesirable side-effects.

Any goal of the agent can now be the root of a debugging tree. These comprise the initial goals an agent may have, goals that result from communication with other agents, and of course subgoals of other goals. In summary, anything that leads to a plan being instantiated is considered. This also covers plans that are used in response to belief modifications (e.g. due to changes in the environment, or mental notes of the agent itself), which are not considered goals, but may trigger a plan nevertheless.

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 Fig. 3. 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.

Note that there are many ways this agent could be improved. For example, the agent does not memorise the locations of any blocks and always has to explore the rooms again to find a block of the appropriate colour.

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 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 Fig. 4 and starts asking questions.

Since it was correct to *add* all goals that were added, we skip these questions in this description. First, the user has to answer whether the goal `!completeTask(t0, red)` has been achieved. 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 achieved. 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 Fig. 5.

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 subgoals of valid parent goals.

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

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

+!prepared : task(Id, 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. 3 Jason example agent for the BW4T (summary). E.g. (in the full code), there is another plan for handling (i.e. skipping) charging when the agent is already charged. There is a similar plan to skip processing when no packaging is required

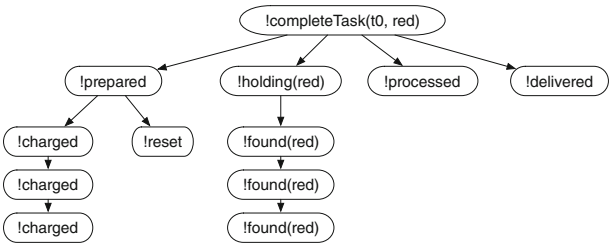

Fig. 4 Debugging tree for the BW4T example

3.2 Plan selection debugging

So far, we have seen how algorithmic debugging can be applied to locate the goal or subgoal that probably directly contains the reason for a bug. A helpful debugging assistant should now continue to help the user pinpoint an exact reason that contributed to the bug. First, possibly a wrong plan could have been chosen to achieve the goal. This can be determined with a simple sequence of questions.

Are the context conditions of the relevant plans correct? If one of the relevant plans (matching the goal trigger) has a wrong context condition, it might have been counted as not applicable, although it would have been, or vice versa. That means

- the plan that was selected is actually not applicable, or
- a previous plan that was not determined applicable should have been selected instead.

Should another context condition have been satisfied? This is slightly related to the first case, however, it points more to an issue prior to plan selection. If the plans' context conditions are correct, but another context condition should have been satisfied, it is implied that either the agent does not have the right beliefs, or something else that was used to determine the truth of the context conditions, e.g. an internal action, did not perform as intended.
Is the order of plans correct? Normally, plans are selected following their order in the plan base. If multiple plans are applicable, the first one is selected. Thus, this question is only necessary in that case.
Is another plan option missing? Maybe none of the plans are actually sufficient to handle the specific case. If so, the selected plan needs to be differentiated into two plans, one of which handles the new case.

The plan selection debugging process is summarised in Fig. 6.

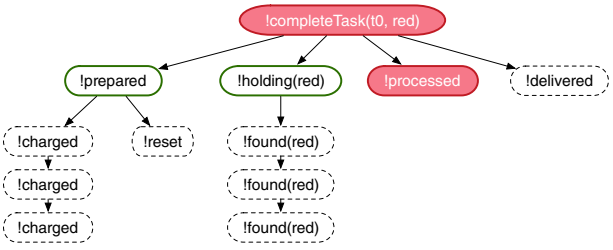

Fig. 5 Marked debugging tree for the BW4T example. Red nodes (filled) are invalid, nodes with a green border are valid. Nodes with a dashed border have not been visited

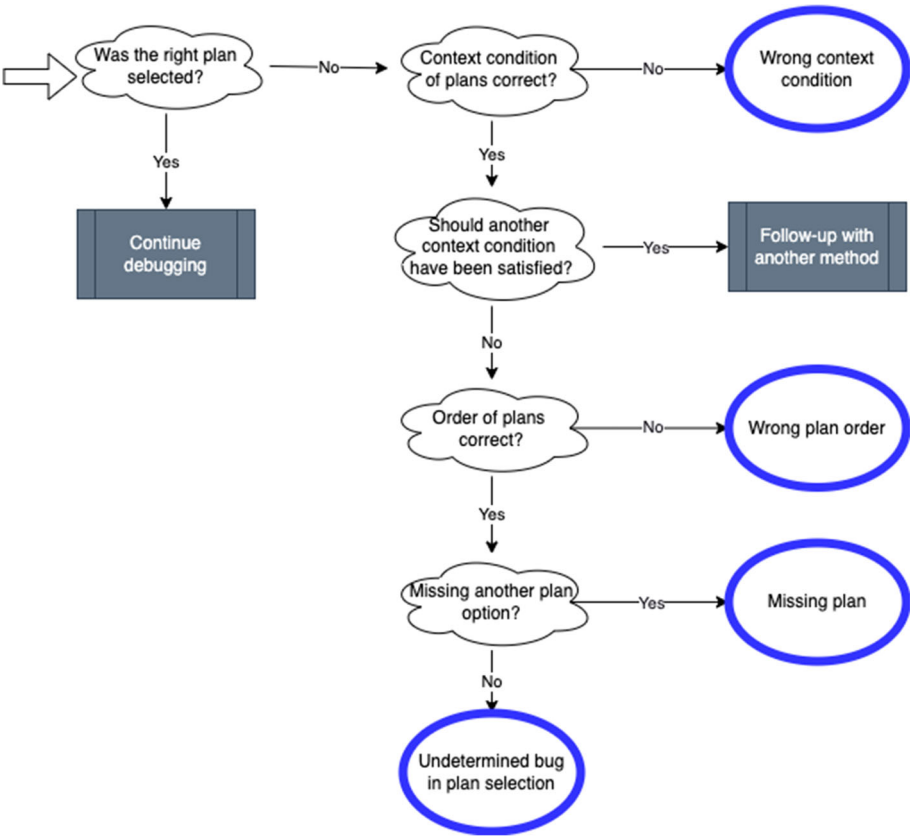

Fig. 6 Debugging the plan selection

Any of these questions can locate a bug. However, if none of them do, an undetermined bug in the plan selection is reported.

3.3 Plan debugging

Now, algorithmic debugging has pointed at the buggy (sub-)goal and we know that the correct plan has been selected to handle it. Therefore, the bug should be somewhere in the plan and we need to locate the wrong part of the plan, i.e. a buggy or missing instruction. To achieve this, we combine the traditional form of debugging, i.e. stepping through the program, with the general activities from algorithmic debugging, i.e. asking the user whether an instruction performed as expected and presenting them with relevant information to answer this question. The key part now is to show the most relevant data for each type of instruction (and allow the user to consult more information if necessary).

In Jason, the most prominent instructions are actions, internal actions, achievement goals, test goals and expressions/constraints.

Actions: An action usually modifies the agent's environment in some way. The modifications are then usually perceived by the agent some time later. In Jason, an action is added to the

set of pending actions and checked in each cycle for completion. To allow the user to verify the action's effects, the plan debugger shows the agent state

- when the action is started, and
- when the action is completed.

As always, the user could review some additional agent states, e.g. if perception takes a while and the action's effects are not immediately visible to the agent.
Internal actions: An internal action can be used to perform some computation in the underlying Java language. This may possibly affect the agent's internal state or deliver a result by unification. For example, the internal action `.gcd(4,8,X)` could calculate the greatest common divisor and "store" the result in `X`. Thus, it is advisable to present the user with

- the agent state after the cycle in which the internal action was executed, and
- the current unifier, especially the newly unified variables.

Achievement goals: The achievement goals have already been checked as part of the algorithmic debugging process and can be skipped.
Test goals: A test goal can be used to

- make sure that something can be derived from the agent's beliefs, and/or
- find some value in the agent's beliefs.

Additionally, if there are relevant plans for the test goal trigger, an entire plan might be instantiated to acquire the information before the belief base is queried. To validate this instruction, the user has to check

- whether the test goal succeeded or failed (and why), and
- how the free variables were unified.

Expressions: An expression, sometimes also called constraint, represents a condition whether the plan can follow its execution. Such an expression could be e.g. `Energy > 50`. For the validation of this instruction type, the user only has to check whether the condition was satisfied, and if not, why it was not.

Using the particular information provided, the user has to say whether the result is as expected or not. If not, the buggy instruction has been found.

The process is summarised again in Fig. 7. If all instructions have been checked by the user, but contrary to expectation no buggy instruction has been found, it has to be considered

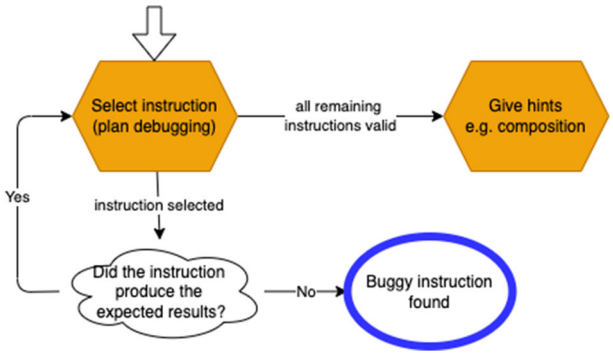

Fig. 7 The plan debugging process

whether the composition or arrangement of instructions is correct, or if maybe some instruction is missing at some point. This could be presented as suggestions for further investigation to the user.

Finally, also for plan debugging a different navigation strategy than selecting instructions chronologically could be employed. However, plans are usually not overly long and jumping back and forth is likely to confuse the user more than it might potentially shorten the process.

3.4 Connection to "Why?" questions

Coming back to the earlier question of how algorithmic debugging relates to the approach based on "Why?" questions presented in [18], the two techniques appear to be complementary.

The complete debugging process so far can be seen in Fig. 8, stitched together from the previously presented fragments. Additionally, the possible connection points to the "Why?" questions approach have been added (as yellow boxes):

1. If a bug in the selection of goals has been found, algorithmic debugging stops. However, the question why the goal was added probably remains. This is a prime target to answer with the "Why?".
2. If some context condition should have been satisfied (or not), the "Why?" can help to find out why the context in question was (not) satisfied.
3. If a buggy instruction has been found, the approach can be used to find out why the instruction did not deliver the expected results. For example, its precondition might not have been satisfied.
4. If a buggy goal was found but all instructions of its plan were correct, the approach could be used to find out what else could be wrong. Maybe some influence external to the plan or even the agent has messed with the plan's execution.

In short, the algorithmic debugging process can either locate a bug or at least lead up to a starting point for the "Why?". question approach

It should also be noted, that algorithmic debugging for BDI agents, as presented in this paper, only works to identify bugs related to an agent's goals. One could imagine a situation where an agent "misbehaves" during the course of achieving some goal without this affecting the goal's achievement. If this is observed by the developer, another technique, like the "Why?" is still necessary to track down the reason for the misbehaviour.

Finally, there is another type of connection between the two approaches. While the process of algorithmic debugging is quite rigid and follows a fixed scheme, an integration of both techniques would allow the user to take control at any moment, following their own intuition. It is not hard to think of situations where the user has an idea of where the bug is before the debugger has had the chance to ask all its questions. Similarly, the "Why?" could be used at any point in the algorithmic debugging process to get further information for validating nodes if necessary.

3.5 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 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

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

Fig. 8 The complete debugging process

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 Fig. 9. 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 achieved 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.

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

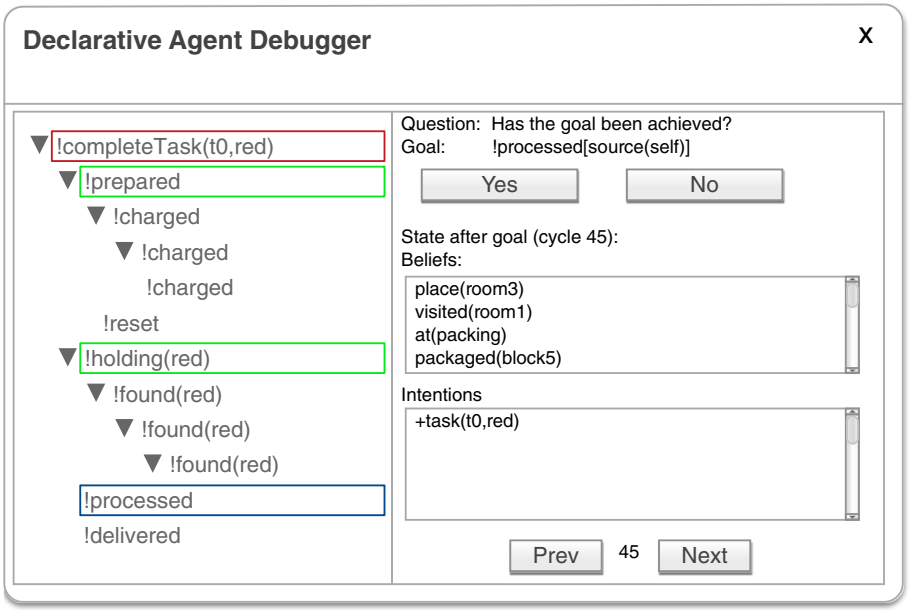

Fig. 9 A wire frame of the debugger

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

3.6 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 achieved. In these kind of looping structures, algorithmic debugging also does not offer much of an advantage. maybe bugs in earlier intentions?

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 subgoals. Some further considerations are more tailored towards AgentSpeak and Jason, but are likely easily adapted to alternatives. 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 subgoal 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.

3.7 Improving conditions for debugging

In order to improve the effectiveness, efficiency or usability of this approach based on algorithmic debugging, one can also aim to alter the conditions in which it is applied.

For one, we mentioned earlier that a conceptual computation might span multiple consecutive intentions, e.g. when one intention ends establishing a belief (through the environment) that itself triggers a follow-up intention. In many cases, it will be prudent to consider both intentions to be part of the same computation. This issue can be easily solved however, if the agent platform supports conceptually linking intentions. For example, AgentSpeak(ER) [15], an extension of AgentSpeak(L), supports combining a plan with additional sub-plans that can only be used in the context of the parent plan. That is, these other sub-plans can only be triggered while their parent plan is still active, i.e. its goal not achieved, either by achievement goals or external event triggers. In this case, the additional behaviours are conceptually (and mechanically) linked to the parent plan and incorporated in its stack, which automatically makes them part of the overarching debugging tree.

The remaining points are related to the (partial but increased) automation of node validation in the debugging tree. For example, if *declarative goals* were to be employed, the debugger could automatically check if all goals succeeded. While many Jason programmers tend to use descriptive goal names, they could as well describe the state that the plan is supposed to achieve as the "name" of the goal. Again in AgentSpeak(ER), support for this type of goal is added. Each AgentSpeak(ER) plan may have an additional *goal condition* that describes when the goal is to be considered achieved. While this would certainly help with validating a node, it would at the same time require the goal conditions to be checked for potential bugs. In addition, each state must still be checked for potential undesired side effects, as we cannot reasonably expect the goal condition to contain all of these (in negated form).

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 subgoal, 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, annotating "everything" with preconditions and postconditions would allow us to further automate locating bugs (at the cost of increased, but maybe also even more deliberate) programming effort.

Having more formally specified environments will also contribute towards better debugging results. For example, it would require fewer annotations, as the effects of an action are known and could automatically be connected to goal conditions or preconditions of later actions, which are also known in that case. A probably much bigger advantage lies in the presentation of results, though. If the effects of actions are known, the programmer can be presented with the precise changes that were caused by any executed plan. This would simplify node validation a great deal. As it is now, only changes between the state before and after the plan can be highlighted, without regard for what actually caused each change.

4 Conclusions and future work

We showed how algorithmic debugging can be applied to BDI agent programs given in AgentSpeak(L). 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.

As always, there is still room for improvement, though. Possible ways to achieve better working conditions for the approach have been discussed in Section 3.7.

Also, *parallel intentions* and the bugs that could arise from their complex interplay need some more consideration. For now, the focus of the approach is on a single intention. Assume that there are two parallel intentions. One of these establishes a subgoal that is required at a later point in time. Now, another intention of the same agent immediately voids that subgoal somehow. The first intention later fails because the subgoal that was needed does not hold anymore. This point in the code can be located with algorithmic debugging, however, the question now is whether the plan should have actually been able to deal with this, or whether the second intention should never have invalidated the subgoal in the first place. Depending on the programmer's view, the bug lies either in the first or in the second intention. In the first case, algorithmic debugging has identified the bug. In the second, it has only identified another symptom and debugging would have to continue in the second intention to find out, why the subgoal had been invalidated in the first place.

One possible approach to deal with parallel intentions would thus be to enable the debugger and/or user to switch between the intention they are debugging at appropriate times.

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 of the workshop paper this work is based on and of this extended version for their very helpful and insightful feedback.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

Code Availability The debugger prototype is available at https://github.com/t-ah/declarative-agent-debugger (ongoing work). The logger and example are available at https://github.com/t-ah/jason-util (also ongoing work).

Declarations

Conflict of interest for all authors None

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. LNAI, vol. 4411, 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. LNAI, vol. 7455, 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. LNCS, vol. 5881, 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. LNAI, vol. 95, pp. 312–319. MIT Press (1995)
15. Ricci, A., Bordini, R.H., Collier, R., Hübner, J.F.: AgentSpeak(ER): An extension of AgentSpeak(L) improving encapsulation and reasoning about goals. Proceedings of 17th AAMAS, 2018, Brasil (2018)
16. 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)
17. Shapiro, E.Y.: Algorithmic program debugging. Yale University (1982)
18. Winikoff, M.: Debugging agent programs with Why? questions. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. pp. 251–259 (2017)