Debugging Agent Programs with "Why?" Questions

Michael Winikoff
Department of Information Science
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

ABSTRACT

Debugging is hard, and debugging cognitive agent programs is particularly hard, since they involve concurrency, a dynamic environment, and a complex execution model that includes failure handling. Previous work by Ko & Myers has demonstrated that providing Alice and Java programmers with software that can answer "why?" and "why not?" questions can make a dramatic difference to debugging performance. This paper considers how to adapt this approach to cognitive agent programs, specifically AgentSpeak. It develops and formalises definitions for "why?" and "why not?" questions and associated answers, and illustrates their application using a scenario.

Keywords

Programming languages and frameworks for agents and multi-agent systems, Development techniques, tools, and platforms

1. INTRODUCTION

Debugging agent systems is hard. Agent programs are concurrent, distributed, often situated in dynamic environments where things can go wrong, and some cognitive agent languages use an execution model that is complex, and that may include failure handling. All of these factors make debugging agent programs quite challenging.

One exciting recent approach is the use of question-based debugging. Proposed for Alice, and subsequently Java, by Ko & Myers [11, 12], the key idea is to provide programmers with software that allows them to indicate, say, a step in the execution trace, ask "why did this happen?", and get an explanation from the system. Their *Whyline* system also allows the programmer to get answers to questions of the form "why did ⟨something else⟩ *not* happen?".

Ko & Myers' evaluation of the WhyLine demonstrated a dramatic difference: use of the Whyline by Alice programmers reduced debugging time by a factor of 8 [11], and two evaluations with Java programmers found that (i) novice users with the Whyline were around twice as fast as more experienced programmers without the Whyline; and (ii) with a larger system, significantly more of the Whyline users were able to debug successfully [12].

This paper adapts the question-based debugging approach to agents, specifically using AgentSpeak, a well-known cognitive agent programming language. Our key contributions are: articulating a general principle for defining explanations for "why" and "why not" questions; formal definitions of a range of question types; formal definitions of the answers to these question types; and a demonstration of the approach using an example scenario.

Compared with Ko & Myers, one significant difference is that our setting is an agent language, rather than Alice or Java. A second significant difference is that their approach is defined operationally: the tracing of execution is defined at a low level by instrumenting bytecode, and the process of providing answers to questions is defined as an algorithm, but not justified in any formal way. By contrast, our approach is formally defined and well founded: tracing is done in a clear and complete way at a high level, and the principle for defining explanations allows us to define a completeness result.

Interestingly, this paper is not the first to propose using questions to debug agent programs: this was initially proposed by Hindriks back in 2012 [9]. It appears that Hindriks was not aware of the earlier Whyline work. This paper builds on Hindriks' paper in two ways. Firstly, it caters for plans that are triggered by events, and that have plan bodies that are sequences of steps, some of which may be sub-goals. By contrast, Hindriks' work is set in the context of the GOAL programming language where plans are simpler (in essence condition-response pairs, although GOAL does provide modules that provide means of focussing on particular rules). Secondly, Hindriks' paper is informal, and sketches questions and answers informally (in English). By contrast, this paper defines answers formally. Additionally, Hindriks' paper does not consider how traces are captured, which we define in Section 2, and it does not appear that his (informal) definitions had been implemented. By contrast, the definitions in this paper have been implemented.

There are various debugging tools for agent programs (e.g. [1, 3, 5, 8, 13]). They tend to provide the programmer with the ability to trace the program's execution, set breakpoints, and inspect the state of execution. What they do not do is provide the programmer with support to navigate and interpret the often rather large amount of information that results from tracing. There has been some work that has considered analysing and visualising multi-agent system execution (e.g. [2]), which is complementary to our work in that it focusses on visualising the whole of execution at once, rather than on navigating and interpreting parts of the execution. The work of Lam & Barber [15, 16] provides such visualisation, but also supports the programmer in answering "why?" questions about the behaviour of the program. However, this is done by the programmer navigating a graph of concepts, where links are retrospectively inferred from log files.

There is also work that is focussed on testing, rather than on debugging (e.g. [7, 14, 18, 19, 20, 21, 22, 23, 24]), the difference being that the focus is on *finding failures*, as opposed to *locating* the *program fault* that led to the failure.

Appears in: *Proc. of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

The remainder of this paper briefly defines an AgentSpeak-like language (syntax & semantics) including a definition of tracing (Section 2), and then proceeds to the core of the paper: definitions of the questions that can be asked and how to derive answers (Section 3). These are illustrated using a scenario (Section 4), and we then conclude (Section 5).

2. BACKGROUND: AGENTSPEAK

The following grammar defines an AgentSpeak-like language[1]. A program Π is a set (actually ordered list) of plans π_i. A plan π_i is of the form $+!t : G \leftarrow P$. The guard G is a logical formula where textually $\wedge$ and $\neg$ are rendered as `&` and `~` respectively (note that t is a term). A plan body P is either empty (denoted "`true`") or a sequence of steps $S_1; \ldots; S_n$ ($n \geq 1$) where each step can be one of: belief addition $+t$ or deletion $-t$, testing a condition $?G$, an action A or posting an event $!t$, where t is a term.

$$\begin{array}{rcl} \Pi & ::= & \pi^* \\ \pi & ::= & +!t : G \leftarrow P \\ G & ::= & \mathit{term} \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \neg G \\ P & ::= & S_1(; S_i)^* \mid true \\ S & ::= & +\mathit{term} \mid -\mathit{term} \mid ?G \mid !t \mid A \\ A & ::= & \mathit{term} \end{array}$$

We now briefly define the semantics. The semantics (Figure 1) use ✔ and ✘ as semantic-level constructs (i.e. not program constructs) that denote the success or failure of part of an execution. For example, testing a condition resolves to ✔ if that condition holds, and ✘ if it does not. A *plan* transition is defined over a configuration $\langle B, P\rangle$ where B is the belief base, and P is the plan body being executed; we also define *agent* transitions over configurations $\langle B, \Gamma\rangle$ where Γ is a multiset of plan body instances (i.e. intentions). The rules (see Figure 1) are in standard Plotkin structured-operational-semantics style (ignore the text above the arrows for now).

These rules are fairly straightforward: a condition test $?c$ succeeds if it holds with respect to the current belief-base (rule 1) and fails if it doesn't (rule 2). An action A can fail (rule 4) or succeed (rule 3, in which case the belief base is updated with the consequences of its execution, denoted $B' = \mathit{conseq}(A, B)$). Belief changes always succeed and update the belief base B ($+b$ rule 5, and $-b$ rule 6). Rule 7 specifies that a sequence $S_1; S_2$ is executed by executing the first step, and then using the auxiliary function $\widehat{\ldots}$ (explained below) to clean up.

Event posting is a little more complex. An event is handled by collecting all applicable[2] plan body instances Δ, and selecting one of them (by default the first, i.e. Δ is actually a sequence, not a set) using $\mathit{select}(\Delta)$ (defined in Figure 1). Note that the second case in select indicates that if there are no options remaining, then attempting to select an option results in failure.

The definition of select uses an auxiliary construct $\triangleright$ to "attach" a set of backup options to a selected plan body. Note that the definition of $\widehat{\ldots}$ specifies that the options are discarded when a plan succeeds ($\widehat{✔ \triangleright S} = ✔$) and that a failed step that has a backup option should use that backup option ($\widehat{✘ \triangleright \Delta} = \mathit{select}(\Delta)$). We then need to specify that the "attach" construct is simply handled by executing the plan body (rule 8). Then the semantics of posting an event (rule 9) is specified by computing Δ (defined in the bottom left of Figure 1) and selecting an option.

The auxiliary function to simplify $\widehat{\ldots}$ is defined in the last row of Figure 1. In essence it simplifies the program. For instance, $✔; S$ is simplified to S, which captures that if the first statement in a sequence has succeeded, then move on to the second. The fourth case implements failure handling: a failure (✘) with alternative options available is simplified by selecting one of those options.

Having defined transitions over individual intentions, we now define a transition rule over an agent configuration (rule A). We assume an auxilliary function $\widetilde{\{P\}}$ which returns $\{P\}$ except that $\widetilde{\{P\}} = \emptyset$ if $P \in \{✔, ✘\}$. This has the effect of removing completed intentions from Γ.

Although not part of the AgentSpeak language, we need to briefly explain how we handle the environment. We specify the environment in terms of action properties. An AgentSpeak program does not include information about action properties (e.g. pre-conditions). Rather, we use a separate environment specification that, for each action, indicates (i) the conditions under which the action will be successful (its pre-condition $pre(A)$), and (ii) what are the effects of performing the action. The action's effects include not just the direct post-conditions (e.g. that putting a block down means you are no longer holding it), but also the percepts that result from the action (e.g. if the block is being put down in the dropzone, and it's the correct colour, then the agent is told about the new desired colour). Exogenous actions can be handled by making their effects part of the effects of an action.

In order to debug programs we need to not just execute them, but also capture the execution. The semantics in Figure 1 therefore also capture traces: the text above the arrow is a *trace term*. For example, rule 3 indicates that when an action A is successfully executed, it results in a term $act^{✔}_N(A, B, B')$. The grammar below defines the trace terms, where N is a (numerical) identifier for the step (used to disambiguate where, for instance, the same action is performed multiple times during execution; we assume that N increases over time), B and B' are the beliefset (respectively before and after the action's effects, but for other rules we only capture the beliefset resulting from the transition), and ✔ and ✘ indicate whether the transition is a successful or failed one. Finally, in the transition for events Δ is the set of alternatives.

$$\begin{array}{rcl} Tr & ::= & test^{✔}_N(c) \mid test^{✘}_N(c) \mid act^{✔}_N(A, B, B') \mid act^{✘}_N(A, B) \\ & & \mid addb_N(b, B, B') \mid delb_N(b, B, B') \\ & & \mid call_N(\Delta, B, B', Tr') \mid true_N \\ & & \mid Tr_1; Tr_2 \end{array}$$

Now, a trace can be a linear sequence, but it turns out to be useful to capture the hierarchical structure. In order to obtain hierarchical traces that reflect the structure of the calls, rather than a linear sequence, we use a variant rule ($9'$). This performs all the transitions associated with the sub-goal, and packages up the resulting trace into a trace term Tr' that is part of the trace term for the call. The condition $P \in \{✔, ✘\}$ specifies that the execution of the sub-goal has completed. As usual we use $\longrightarrow^*$ to denote the transitive closure of $\longrightarrow$, but where traces are collected into sequence (rule $*$ in Figure 1).

Formally, given an initial beliefset B_0 and a top-level goal $!t$, we have that $\langle B_0, !t\rangle \xrightarrow{Tr}{}^* \langle B, P'\rangle$ where P' is either ✔ or ✘, and Tr is a trace term that captures the execution. We use $\mathcal{T}$ to denote the trace resulting from executing the top-level goal. We overload notation by defining $Tr \in \mathcal{T}$ to be true if Tr is a trace term that appears within $\mathcal{T}$ (either as an element of a sequence, or as

[1]There are a few minor differences compared with Rao's original language [25].

[2]A plan $\pi = +!t{:}G{\leftarrow}P$ is *relevant* for an event e if its trigger t unifies with the event, and it is *applicable* if, in addition, its context condition G is true.

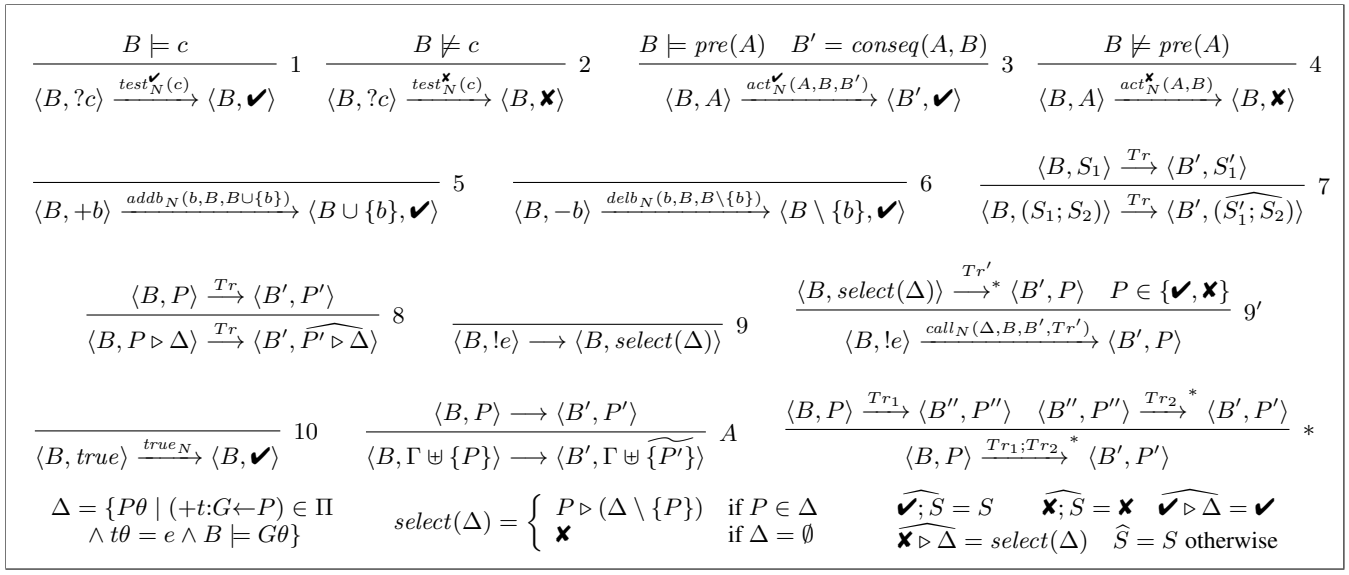

Figure 1: AgentSpeak Semantics

a sub-trace (last argument of $call_N$). We also overload $\in$ to check whether a step S appears in a sequence of steps P (formally: $S \in P$ iff $P = S_1; \ldots; S; \ldots; S_n$), and to check if a step S appears in a plan π (formally: $S \in \pi$ iff $S \in P$ where $\pi = +!t{:}G{\leftarrow}P$).

Given an implementation of AgentSpeak, it could be modified to collect a trace. Alternatively, one could use a modified AgentSpeak meta-interpreter [28] that is extended to capture traces. This is obviously not efficient, but would be adequate for prototyping, and for debugging smaller programs.

3. POSING AND ANSWERING QUESTIONS

The basic principle for answering a "why" question is the following: if we consider the semantics as being non-deterministic, and describing all possible executions, then an explanation for "why did you do $\langle this \rangle$?" is a (prior) point in the execution where there was a choice, and making a different choice would not have led to "$\langle this \rangle$" being done. This basic principle gives us a basis for formulating a completeness result. Suppose that we have defined a particular question type (e.g. "why did you do step S at point N?") and a way of deriving an answer, i.e. an explanation, for a question of that type given a particular trace. Then the way of deriving the answer is *complete* if and only if it is able to provide all possible explanations (i.e. earlier points in the execution where a different choice could have been taken that would have affected the answer to the question).

Clearly, there are many possible answers that could be given, since any action depends on many prior choice points. In this work the overarching goal is to support a programmer to debug their program, so a key consideration is what answer(s) would the programmer find most useful? As a general principle, we answer in terms of the *closest* choice point, and allow the programmer to repeat the "why" question with respect to the answer to find causes that are further away. This process will be familiar to any parent of young children. Also familiar to parents will be the need for a "because" base case. Any query relating to the initial goal will be met with "because that was the initial goal that you specified".

In considering the possible types of questions we follow Hindriks [9] in distinguishing between *doing* and *knowing* (more precisely *believing*): we therefore can ask an agent why it *believes* something, or why it *did* a certain action (more generally, a step). In addition to being able to ask why an agent did or believed something, it can also be useful to explore why it did *not* do or believe something that we expected it to do or believe. This gives us four main types of questions: "why did you do step S at N?" (formally: $why_N(S)$), "why did you believe condition C at point N?" ($why_N(C)$), "why did you *not* do step S at point N?" (formally[3] $\overline{why}_N(S)$), and "why did you *not* believe C at point N?" ($\overline{why}_N(C)$).

In defining how to answer question of these types it turns out (see Section 3.1) that we also need to be able to pose and answer a number of auxiliary question types: "why did step S at point N succeed/fail?" (formally: $why^{\checkmark}(S_N)$ or $why^{\text{✘}}(S_N)$), and "why was plan π selected at N?" (formally: $why^{\triangle}_N(\pi)$, where N is the point at which the event was posted, which is where the set of applicable plans is computed[4]).

Turning to answers, an answer to one of the four main "why" questions is one or more prior steps and/or prior beliefs. For example, an explanation for why a certain action was done at a certain point in time may be in terms of previous steps (e.g. an event being posted) and certain beliefs holding at past time points. Formally, we use S_N to designate the occurrence of step S at point N, and C_N to indicate that the condition C was believed at point N, we also use $C_{\leq N}$ to indicate that condition C held at all execution points prior to and including N. We use NAP_N to indicate "No Applicable Plan at N": this is used to explain why an event failed. We allow answers to be combined with conjunction and disjunction. It is also possible for an answer to be false ("$\bot$") which indicates that the question is wrong, e.g. asking why a step was done when in fact the step was not done. An answer of true ("$\top$") indicates that something was inevitable, for example, a belief update succeeding.

This gives us the following definitions for the possible (formal) forms of questions (Q) and answers (A).

$$\begin{aligned} Q \quad ::= \quad & why_N(S) \mid \overline{why}_N(S) \mid why_N(C) \mid \overline{why}_N(C) \\ & \mid why^{\checkmark}(S_N) \mid why^{\text{✘}}(S_N) \mid why^{\triangle}_N(\pi) \end{aligned}$$

[3] We use an overline to denote negation.

[4] This corresponds to an "eager" evaluation of context conditions, which is done by some, but not all, BDI agent platforms [28].

$$A \quad ::= \quad \top \mid \bot \mid S_N \mid C_N \mid C_{\leq N} \mid \mathsf{NAP}_N \mid A_1 \wedge A_2 \mid A_1 \vee A_2$$

An answer to a question that is of the form S_N or C_N has a corresponding follow-up question $why_N(S)$ or $why_N(C)$.

Each question is asked with respect to a given step in the trace. We assume that we can uniquely identify steps (e.g. that when we say "why did you do putDown?" that we can use an additional ID to distinguish between multiple occurrences of the putDown action). In terms of a user interface, we envisage (following the Whyline), that a programmer would be able to select a step in the trace, and, using a menu, ask "why did you do this?", "why did you not do ⟨*select other possible steps*⟩?", "why did you believe ⟨*specify condition*⟩?", "why did you not believe ⟨*specify condition*⟩?". We also envisage that a part of an answer (C_N or S_N) can be selected, and the corresponding follow-up question posed.

3.1 "Why did you do . . . "

We begin by considering the explanations for why a particular step S was performed at point N (formally: $why_N(S)$).

Following the basic principle outlined earlier, we consider the sequence of steps that result in S, and at each point, what else could have happened. We begin with the posting of the event $!t$ that resulted in the selection of the plan $\pi = +!t{:}G{\leftarrow}P$ where S is one of the steps in P. Figure 2 shows the execution from $!t$, including an indication of where there were alternatives, i.e. where things could have gone differently.

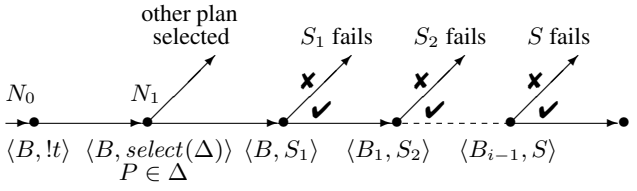

Figure 2: Possible executions from $!t$ resulting in step S

Beginning with posting the event $!t$ (at N_0), a set of applicable plans Δ is computed. In order for S to be done, the plan in question $\pi = +!t{:}G{\leftarrow}P$ (where $S \in P$) must be selected, which requires that it be applicable ($P \in \Delta$, i.e. that its context condition holds at N_0, formally: $B \models G$), and that it is the first applicable plan, which is the case if all earlier relevant plans are not applicable. Once π has been selected, in order for the execution of P to lead to S, each of the steps S_i preceding S in the plan body P must have succeeded. Collecting these conditions we have that in order for posting $!t$ to result in S, we must have that: $!t$ was posted at N_0 (formally: $!t_{N_0}$); and the plan's context condition holds (formally: G_{N_0}, where $\pi = +!t{:}G{\leftarrow}P$, and $S \in P$); and for all relevant plans $\pi_i = +!t : G_i \leftarrow P_i$ that precede π we have that the context condition G_i does *not* hold (formally: $(\neg G_i)_{N_0}$); and for each step S_i that appears before S in P, step S_i succeeds ($why^{\checkmark}(S_i)$, defined in Section 3.1.1).

Formally, the answer to $why_N(S)$, given that the trace contains a trace term $call_{N_0}(\Delta, B, B', Tr')$ where the selected plan is $\pi = +!t{:}G{\leftarrow}P$, $P \in \Delta$, and $P = S_1; \ldots; S_k; S; \ldots$, is:

$$why_N(S) = !t_{N_0} \wedge why^{\Delta}_{N_0}(\pi) \wedge \bigwedge_{i \leq k} why^{\checkmark}(S_i)$$

where $why^{\Delta}_{N_0}(\pi) = G_{N_0} \wedge \bigwedge_{\pi_i \text{ before } \pi} (\neg G_i)_{N_0}$.

Rendered in English, this says "step S was done because event $!t$ was posted at N_0, and plan π's context condition G was true at N_0, and π was selected at N_0 (because all earlier plans π_i, had context conditions that did not hold at N_0), and all of the steps S_i before S succeeded". A programmer could then ask follow-up questions about each of the components of this explanation, e.g. "why was $!t$ posted at N_0?", "why did you believe G at N_0?", "why did you not believe G_i at N_0?", and "why did step S_i succeed?".

However, the scenario in Figure 2, and the discussion so far, are actually incomplete, since they assume that the statement of interest S is in the *first* plan to be selected. But it is also possible that the first selected plan fails, and then an alternative plan is used which leads to S being done. Dealing with this requires generalising the explanation for why the plan π containing S was selected: in order for π to be selected, we require that any relevant plans that occur before π must either not be applicable (as above) or must have already been tried, and failed (new case). This intuition is formalised below in Section 3.1.2, and replaces the earlier definition of $why^{\Delta}_{N_0}$. The definition for $why_N(S)$ is modified by defining π to be not the first plan that is selected, but the plan π_i whose plan body P_i contains S (formally: $S \in \pi_i$). Note that we do not require that π_i succeeds: it could be that it gets as far as S, and then subsequently fails.

Hindriks' informal definition of an explanation for why an action was performed [9] comprises two parts: the condition of the rule that applied, and the pre-condition of the action. By contrast, our definition, which deals with a setting where plans have bodies with sequences of steps, and are triggered by events, is more complex. It also caters for failure recovery.

Before proceeding to consider the question type "why do you believe C?" we first deal with the auxiliary question types: "why did step S succeed/fail?" and "why was plan π selected?"

3.1.1 "Why did step S succeed/fail?"

We now turn to defining why a step succeeds (respectively fails). Intuitively, from the semantics, a belief modification ($+t$ or $-t$) always succeeds. A condition check ($?G$) succeeds iff the condition G holds. An action A succeeds iff its pre-condition $pre(A)$ holds. Finally, an event $!t$ succeeds if there is at least one applicable plan, and if either the selected plan succeeds (all its steps succeed), or some number of plans fail but alternative plans exists and eventually a selected plan succeeds.

The following equations formalise this intuition. It turns out to have some complications. Firstly, in order to explain why an event succeeded (respectively failed) we need to provide explanations for the success (resp. failure) of program fragments (alternatives and sequences). Since program fragments do not have identifiers (N) we need to this this with respect to (compound) trace terms, rather than with respect to a given location. More precisely, the initial query is a single step S with an associated location N. We answer it by looking up the corresponding trace term (see table below). If there is not a corresponding trace term, then the answer is "$\bot$" (i.e. "actually it didn't"). Otherwise we use functions $why^{\checkmark}(P, Tr)$ and $why^{\text{✘}}(P, Tr)$ that are given the program fragment and trace term. For compound P (e.g. $P = S_1; S_2$) the trace term will itself be compound ($Tr_1; Tr_2$). Formally:

$$\begin{aligned} why^{\checkmark}(S_N) &= \text{if } Tr \in \mathcal{T} \text{ then } why^{\checkmark}(S, Tr) \text{ else } \bot \\ why^{\text{✘}}(S_N) &= \text{if } Tr \in \mathcal{T} \text{ then } why^{\text{✘}}(S, Tr) \text{ else } \bot \\ &\quad \text{where } Tr \text{ is the corresponding trace term for } S \end{aligned}$$

The *corresponding trace term* Tr for step S in the context of $why^{\checkmark}$ or $why^{\text{✘}}$ is defined in Table 1.

The equations in Figure 3 define $why^{\checkmark}(S, Tr)$ and $why^{\text{✘}}(S, Tr)$. These functions return an answer formula (following the grammar for A). This is simplified (e.g. $\bot \vee X \Rightarrow X$, $\bot \wedge X \Rightarrow \bot$, $\top \vee X \Rightarrow \top$, $\top \wedge X \Rightarrow X$, etc.) before being reported to the user.

For $true$, $+b$ and $-b$, we have that $why^{\checkmark}(S, Tr)$ is just $\top$, since

Context	Step	Corresponding Trace Term
✔	A_N	$act^{\checkmark}_N(A, B, B')$
✘	A_N	$act^{\times}_N(A, B)$
✔	$?C_N$	$test^{\checkmark}_N(C)$
✘	$?C_N$	$test^{\times}_N(C)$
✔ or ✘	$+b$	$addb_N(b, B, B')$
✔ or ✘	$-b$	$delb_N(b, B, B')$
✔ or ✘	$true_N$	$true_N$
✔ or ✘	$!t_N$	$call_N(\Delta, B, B', Tr')$

Table 1: Corresponding trace term for a given term

these steps always succeed. Similarly, $why^{\times}(S, Tr)$ is $\bot$ for these step types, since they cannot fail, so the answer to "why did it fail?" ($why^{\times}$) is always "actually it didn't" ($\bot$). An action A that succeeds is explained by its pre-condition being true when it is performed, and an action's failure is explained by its pre-condition being false; similarly a condition $?C$'s success is explained by the condition C holding at N, and its failure by the condition not holding at N (note that N appears in Tr, e.g. $Tr = act^{\checkmark}_N(A, B, B')$).

For $why^{\checkmark}(!t, Tr)$ there are two cases (note: in Figure 3 "where" clauses apply to both preceding equations, i.e. to both $why^{\checkmark}$ and $why^{\times}$). Firstly, if the set of applicable plans is empty, then the event cannot succeed, so $why^{\checkmark}(!t, Tr)$ is $\bot$ when $\Delta = \emptyset$. Otherwise, the explanation for the success of the sub-goal t is the combination of an explanation for why the execution of the selected plan $P \triangleright \Delta'$ eventually succeeded, and for why the plan that ended up being successful was selected ($why^{\Delta}_N(\pi_j)$, defined in the next section). We identify π_j by finding the plan that succeeded, i.e. the $\pi_j \in \Pi$ that has an explanation for its success ($why^{\checkmark}(P_j, Tr_j) \neq \bot$, where Tr_j is a subtrace[5] of Tr). Note that this means that the explanation for why $P \triangleright \Delta$ succeeded does not need to include an explanation for why a given plan was selected, since that is provided by $why^{\Delta}_N(\pi_j)$.

Now turning to $why^{\times}(!t, Tr)$ we have two cases: if the set of applicable plans is empty ($\Delta = \emptyset$) then the event's failure is explained by the lack of applicable plans (NAP_N), otherwise, the event's failure is explained by the reasons for why every applicable plan failed. Note that we do not include an explanation for plan selection, since all applicable plans must be tried (and must fail) in order for an event to fail. However, one piece of information that might be of interest to a programmer is why certain plans were not applicable. The definition could be extended to include this if desired by adding a variant of $why^{\Delta}_N(Tr)$ that was of the form *if* $P_i \in \Delta$ *then* $\top$ *else* $(\neg G_i)_N$.

For an alternative $P \triangleright \Delta$ the explanation for why it succeeded is a combination of two cases: either P succeeded, in which case the explanation is why did P succeed; or P failed, in which case the explanation is the combination of an explanation for why P failed, and for why the remainder P_Δ that was selected next from Δ succeeded (recall that P_Δ is actually of the form $P' \triangleright \Delta'$). In the case where Δ is empty, only the first case applies. The explanation for why $P \triangleright \Delta$ failed is a combination of the explanations for why P failed, and why P_Δ failed (unless $\Delta = \emptyset$, in which case we just have $why^{\times}(P, Tr)$).

For a sequence $P_1; P_2$ the explanation for its success is the combination of explanations for why P_1 succeeded and why P_2 succeeded. On the other hand, the explanation for the failure of $P_1; P_2$ has two cases: either P_1 has failed (in which case it is explained), or P_1 succeeded and P_2 failed, in which case the explanation is a combination of these two explanations.

For the various cases where $Tr \equiv Tr_1; Tr_2$ we split Tr in a way that matches the execution, for instance, for $P_1; P_2$, Tr_i is the part of the trace that corresponds to the execution of P_i. In the implementation this is done by passing the whole trace to the first sub-program to execute, and the trace that is unused is returned, and then passed to the next sub-program.

$$
\begin{array}{rcl}
why^{\checkmark}(true, Tr) & = & \top \\
why^{\times}(true, Tr) & = & \bot \\
why^{\checkmark}(+b, Tr) & = & \top \\
why^{\times}(+b, Tr) & = & \bot \\
why^{\checkmark}(-b, Tr) & = & \top \\
why^{\times}(-b, Tr) & = & \bot \\
why^{\checkmark}(A, Tr) & = & (pre(A))_N \\
why^{\times}(A, Tr) & = & (\neg pre(A))_N \\
why^{\checkmark}(?C, Tr) & = & C_N \\
why^{\times}(?C, Tr) & = & (\neg C)_N \\
why^{\checkmark}(!t, Tr) & = & \text{if } \Delta = \emptyset \text{ then } \bot \text{ else} \\
 & & why^{\Delta}_N(\pi_j) \wedge why^{\checkmark}(P \triangleright \Delta', Tr') \\
why^{\times}(!t, Tr) & = & \text{if } \Delta = \emptyset \text{ then } \mathsf{NAP}_N \text{ else} \\
 & & why^{\times}(P \triangleright \Delta', Tr') \\
 & & \text{where } Tr = call_N(\Delta, B, B', Tr') \\
 & & \text{and } select(\Delta) = P \triangleright \Delta' \\
 & & \text{and, for } why^{\checkmark}(!t, Tr) \text{ when } \Delta \neq \emptyset, \\
 & & \pi_j = +!t{:}G_j{\leftarrow}P_j \in \Pi \wedge P_j \in \Delta \\
 & & \text{and } \exists Tr_j \prec Tr' : why^{\checkmark}(P_j, Tr_j) \neq \bot \\
why^{\checkmark}(P{\triangleright}\Delta, Tr) & = & \text{if } \Delta = \emptyset \text{ then } why^{\checkmark}(P, Tr) \text{ else} \\
 & & why^{\checkmark}(P, Tr) \vee \\
 & & (why^{\times}(P, Tr_1) \wedge why^{\checkmark}(P_\Delta, Tr_2)) \\
why^{\times}(P{\triangleright}\Delta, Tr) & = & \text{if } \Delta = \emptyset \text{ then } why^{\times}(P, Tr) \text{ else} \\
 & & why^{\times}(P, Tr_1) \wedge why^{\times}(P_\Delta, Tr_2) \\
 & & \text{where } Tr \equiv Tr_1 \ ; \ Tr_2 \\
 & & \text{and } P_\Delta = select(\Delta) \\
why^{\checkmark}(P_1{;}P_2, Tr) & = & why^{\checkmark}(P_1, Tr_1) \wedge why^{\checkmark}(P_2, Tr_2) \\
why^{\times}(P_1{;}P_2, Tr) & = & why^{\times}(P_1, Tr) \vee \\
 & & (why^{\checkmark}(P_1, Tr_1) \wedge why^{\times}(P_2, Tr_2)) \\
 & & \text{where } Tr \equiv Tr_1 \ ; \ Tr_2
\end{array}
$$

Figure 3: Definition of $why^{\checkmark}$ and $why^{\times}$

3.1.2 *"Why was plan π selected?"*

As discussed earlier, a plan π is selected because: (i) its context condition is true at the point of selection, and (ii) all the relevant plans that appear before it in the plan library Π either are not applicable, or have already been tried, and have failed.

We formalise this as a conjunction that for each π_i that comes before π in the relevant plan list Π_t includes either an explanation that π_i was actually not applicable ($(\neg G_i)_N$), or, for applicable plans (i.e. $P_i \in \Delta$, where $\pi_i = +!t{:}G_i{\leftarrow}P_i$), an explanation for why it failed ($why^{\times}(P_i, Tr_i)$, where Tr_i is the appropriate subtrace).

[5] We define a subtrace as: $T \prec T_0$ iff $\exists T_1, T_2 : T_1; T; T_2 \equiv T_0 \vee T_1; T \equiv T_0 \vee T; T_2 \equiv T_0$.

$$why^{\Delta}_N(\pi_j) \;=\; (G_j)_N \wedge \bigwedge_{i<j} \text{if } P_i \in \Delta \text{ then } why^{\times}(P_i, Tr_i) \text{ else } (\neg G_i)_N$$
where $Tr = call_N(\Delta, B, B', Tr') \in \mathcal{T}$
and $\Pi_t = \{\pi_i \mid \pi_i \in \Pi \wedge \pi_i = +!t{:}G_i{\leftarrow}P_i\}$
and $\pi_j = +!t{:}G_j{\leftarrow}P_j \in \Pi_t \wedge P_j \in \Delta$
and $\pi_i = +!t{:}G_i{\leftarrow}P_i$
and for $i < j$ where $P_i \in \Delta, Tr_i \prec Tr'$
is such that $why^{\times}(P_i, Tr_i) \neq \bot$

3.2 "Why do you believe ..."

In order to answer the question "why do you believe condition C at point N?" (formally: $why_N(C)$) we need to firstly consider where C is true. We define three cases. Firstly, it may be the case that C was true from the start of execution, and remained true until N. In this case we use the answer $C_{\leq N}$. Secondly, it could in fact be the case that C is *not* true at N, in which case the answer to $why_N(C)$ is $\bot$. Finally, the (normal and expected) case is where C became true at some point before N, as a result of a step S at point N_1 (where $N_1 < N$). Specifically, S is such that before S was performed the condition C did not follow from the agent's beliefs ($B \not\models C$), but after S it did ($B' \models C$). Part of the explanation then for why C is believed at N, is that S was performed at N_1 (formally: S_{N_1}). We also require that C remains true between N_1 and N (formally: $\neg\exists Tr \in \mathcal{T}{:}N_1 < Tr.N \leq N \wedge Tr.B \not\models C$, where we use the notation $Tr.X$ to refer to the component of Tr named X, e.g. $Tr.B$, $Tr.N$, $Tr.B'$).

However, there is more to the explanation than just S_{N_1}. The step S may have only contributed *part* of C. For example, if $C = p \wedge q$ and $S = +p$, then in order for S to result in a beliefset $B' \models p \wedge q$ we must have that $B \models q$. In other words, in addition to the explanation of $why_N(C)$ being S_{N_1}, we also need that the agent's beliefs just before S support the "rest" of C. If C includes negations, then this requirement may include that B does *not* contain certain beliefs. Below we simply (informally) define C_0 as the condition that must hold before S (i.e. at N_1) in order for C to hold after S (space precludes a formal definition).

$$why_N(C) \;=\; \begin{cases} C_{\leq N} & \text{if } \forall Tr \in \mathcal{T}{:}Tr.N{\leq}N \Rightarrow Tr.B \models C \\ \bot & \text{if } \forall Tr \in \mathcal{T}{:}Tr.N{=}N \Rightarrow Tr.B \not\models C \\ S_{N_1} \wedge (C_0)_{N_1} & \text{otherwise} \end{cases}$$
where $Tr \in \mathcal{T} \wedge N_1{=}Tr.N \wedge N_1 < N$
$\wedge\, Tr.B \not\models C \wedge Tr.B' \models C$
$\wedge\, \neg\exists Tr \in \mathcal{T}{:}N_1 < Tr.N \leq N \wedge Tr.B \not\models C$
and S is the step at N_1
and C_0 is the condition required at N_1 so that $Tr.B' \models C$

There is one final additional complication. If we consider things from the perspective of all possible executions, then we need to also consider what else might have happened in between N_1 and N[6]. So for each step between N_1 and N, we need to include an explanation for that step, since if it had been done differently (e.g. a condition test had failed instead of succeeding, or a different plan had been selected) then step S might not occur, or C may not have held at N. However, we do not include this in the definition above (thus making it intentionally incomplete[7]) since this additional explanation is arguably not helpful: a programmer asking why C is believed at N really wants to know the information above (where did C become true), and doesn't want this to be cluttered by explanations for why every intervening step occurred.

Hindriks' definition for how to explain why an agent has a certain belief is similar: either the belief is universally true (in his case a consequence of domain knowledge, since he does not allow initial beliefs), or it is the result of a belief change. He does not consider the requirement for "residual" conditions (C_0), i.e. when a change only made a condition true because C_0 already held.

3.3 "Why don't you believe ..."

Hindriks argues, based on Prolog's closed-world assumption, that "why did you not believe C?" can be treated as being equivalent to "why did you believe $\neg C$?". This would provide a simple definition: $\overline{why}_N(C) = why_N(\neg C)$. However, we argue that this is not always an adequate explanation. Suppose that at point N_2 the condition C does not hold, and the programmer was expecting it to hold. It is possible that C does not hold at N_2 because it was made false earlier (in which case $why_N(C)$ will be an explanation in terms of a step S at an earlier point N_1). But it is also possible that C does not hold at N_2 because something that should have been done, was not done. In other words, a step S that would have made C true was not done. This gives an alternative explanation:

$$\overline{why}_N(C) \;=\; why_N(\neg C) \vee \bigvee_{\substack{\pi \in \Pi \wedge S \in \pi \\ S \rightsquigarrow C \wedge N' < N}} \overline{why}_{N'}(S)$$

This finds a step S that appears in a plan π, and includes as a possible explanation for why C was not believed, the explanation for why S was not done at an earlier point N'. The step S is required to be one that contributes towards the condition C (notation: $S \rightsquigarrow C$). Briefly, the step $+t$ contributes towards the condition t, step $-t$ contributes towards $\neg t$, and an action A with postcondition $post(A)$ contributes towards any $p \in post(A)$. We also have that $S \rightsquigarrow C$ implies that $S \rightsquigarrow C \wedge C'$ and $S \rightsquigarrow C \vee C'$.

It is worth emphasising that these alternative explanations are speculative, and that there may be many explanations, since we consider a range of possible plans and possible time points N'. It therefore may be more helpful to the programmer to only provide these explanations if requested, or in the case where $why_N(\neg C) = (\neg C)_{\leq N}$ (i.e. where C has never held up to N).

3.4 "Why didn't you do ..."

This question is used when the programmer was expecting a particular step S' to have been done, but it wasn't. We assume that the programmer poses this question in response to a particular step that was done, but was not expected. However, since, in AgentSpeak, the selection of a step to be done is determined by earlier plan selection, this query is really not about step S' at N, but about the earlier selection of the plan that led to S' being done at N, and whether an alternative plan could have been selected that would have led to S being done. We assume that the programmer is looking for a *direct* explanation, that is, a possible alternative plan that contains S, rather than a plan that results in another plan that results in S.

So, when the question $\overline{why}_N(S)$ is posed, the first step to answering it is to find the event t whose posting led to the execution

[6]Note that we do not need to consider other possible executions prior to N_1, since the explanation includes that S was done at N_1, and if the programmer asks why S was done at N_1, then earlier alternative executions are considered.

[7]In order to make the definition technically complete we would need to add to the explanation a conjunction over all steps S between N_1 and N, an explanation for why the step succeeded (respectively failed).

of the step S' at N. Formally: $call_{N_0}(\Delta, B, B', Tr') \in \mathcal{T}$ where $S' \in P'$ and $P' \in \Delta$ and S' is the step that was done at N (denoted below $S'@N$).

We then consider the desired step S. In the following we focus on the case where there is exactly one plan π that is relevant, and whose body contains S. If no such plan exists, then there is no way that the event could have led directly to S. If there are multiple plans containing S then each is considered separately, and the explanation is the disjunction of the individual explanations (not shown below).

To explain why S was not performed we need to consider three cases. Firstly, it is possible that the plan $\pi = +!t{:}G{\leftarrow}P$ containing S was simply not applicable, in which case the explanation is simply $(\neg G)_{N_0}$ (along with the explanatory text that this is the context condition of the plan that would have done S). Secondly, if π is applicable it may have been reached, and (since it is applicable), selected, but have failed before reaching the desired step S. In this case the explanation is that the desired plan was selected but failed before reaching S due to $why^{\text{✘}}(P, Tr_p)$ (where Tr_p is the part of the trace Tr' corresponding to the execution of P). Thirdly, an applicable π may not have been reached because an earlier plan has succeeded, in which case the explanation is that the plan was not reached because an earlier plan succeeded due to $why^{\text{✔}}(P_j, Tr_j)$ where $P_j \in \Delta$ is the plan body that succeeded, and Tr_j is the part of the trace Tr' corresponding to the execution of P_j.

$$\overline{why}_N(S) \quad = \quad (!t)_{N_0} \wedge \begin{cases} (\neg G)_{N_0} & \text{if } P \notin \Delta \\ why^{\text{✘}}(P, Tr_p) & \text{if } failed(P, Tr') \\ why^{\text{✔}}(P_j, Tr_j) & \text{otherwise} \end{cases}$$

where $call_{N_0}(\Delta, B, B', Tr') \in \mathcal{T}$
and $\exists S'{:}S' \in P' \wedge P' \in \Delta \wedge S'@N$
and $S \in P$ where $\pi = +!t{:}G{\leftarrow}P \wedge \pi \in \Pi$
and $failed(P, Tr') = \exists Tr_p \prec Tr'$ s.t. $why^{\text{✘}}(P, Tr_p) \neq \bot$
and (for 2nd case) $Tr_p \prec Tr'$ such that $why^{\text{✘}}(P, Tr_p) \neq \bot$
and (for 3rd case) $P_j \in \Delta \wedge Tr_j \prec Tr'$ s.t. $why^{\text{✔}}(P_j, Tr_j) \neq \bot$

Finally, as always (but not shown in the definition above), it is possible that the answer to "why did you not do S?" is "actually I did" ("$\bot$"). Note that it is possible that S was done *after* point N. This still yields a response of $\bot$, but we would also point out to the programmer that S had been done, just not yet.

By contrast, in Hindriks' framework he simply considers the point in time where A was done, and then considers the selection of rules at that point. However, we need to consider multiple time points, since we have plans with bodies that can contain sequences of steps.

4. IN ACTION: AN EXAMPLE SCENARIO

We now illustrate the use of the question-based debugging framework using an example scenario. We use a single agent version of the Blocks World for Teams (BW4T) [10]. In brief, this involves an agent that roams an environment of rooms and corridors, seeking to find coloured blocks, and deliver to a dropzone a pre-specified sequence of block colours. An example AgentSpeak program to deliver blocks in the desired order is in Figure 4.

This program runs in an environment that defines the following four actions. (i) **putDown** with pre-condition that a block is being held ($holding(B)$), and consequence deleting $holding(B)$ and $colour(B, C)$. If the block is being $putDown$ at the dropzone, and is of the correct colour, then the consequences also update the $nextColour(C)$ belief, delete the belief $colour(B, C)$, and add that colour C and block B have been delivered ($delivered(B)$, $delivered(C)$); (ii) **pickUp** with pre-condition that the agent is at a block and is not already holding a block ($atBlock(B) \wedge \neg holding(B')$), and has the effect of removing $atBlock(B)$ and adding $holding(B)$, (iii) **goto(P)** with trivial precondition ($true$) which moves to place P (either a room or the dropbox), with the effect of removing any $atBlock(B)$ and $at(B)$ beliefs, adding $at(P)$, and also noting that the agent has $beenthere(P)$, and, if $P{\neq}dropzone$, also adding beliefs $colour(Block, Colour)$ for any blocks that are at P; and (iv) **gotoBlock(B)** which goes to a block B (pre-condition $\neg delivered(B)$), and, like $goto(P)$, deletes existing $atBlock(B')$ and $at(B')$, and adds $atBlock(B)$.

```
1  +!deliver : nextColour(done) ← +done. % If done then stop
2  % select a block of the right colour and go get and deliver it
3  +!deliver : colour(B, C) ∧ nextColour(C) ∧ ¬holding(B) ←
     gotoBlock(B) ; pickUp ; !deliver.
4  +!deliver : holding(B) ∧ colour(B,C) ∧ nextColour(C) ←
     goto(dropzone) ; putDown ; !deliver.
5  % if holding a block that is not the next colour required then
     put it down (this may occur if e.g. someone else delivers a
     block, so the next colour changes)
6  +!deliver : holding(B) ∧ colour(B,C) ∧ ¬nextColour(C) ←
     putDown ; !deliver.
7  % if I know of a place that I've not yet visited then go there
     (explore)
8  +!deliver : place(P) ∧ ¬beenthere(P) ← goto(P) ; !deliver.
9
10 %%% Initial beliefs
11 belief(nextColour(red)). belief(at(corridor)).
12 belief(place(room1)). belief(place(room2)).
13 belief(place(room3)). belief(place(room4)).
```

Figure 4: Example Program for BWT

The program in Figure 4 is run in a scenario where rooms 1-4 contain respectively red, blue, blue, and yellow blocks, and the desired sequence is red, yellow, blue. This results in the following sequence of actions (the numbers 1-19 are the order, not the actual N in the trace; only actions are shown, except for call(deliver) at line 15).

```
1  goto(room1)
2  gotoBlock(b1) % red
3  pickUp
4  goto(dropzone)
5  putDown % now need yellow ...
6  goto(room2) % exploring ...
7  goto(room3)
8  goto(room4)
9  gotoBlock(b4) % yellow
10 pickUp
11 goto(dropzone)
12 putDown % now need blue ...
13 gotoBlock(b3) % blue
14 pickUp
15 call(deliver)
16 gotoBlock(b2) % blue
17 pickUp % FAILED (because already holding b3)
18 goto(dropzone)
19 putDown % now done
```

An obvious problem with this execution is that the pickUp at line 17 failed. The remainder of this section shows how question-based debugging can be used to debug this program. Note that

```
Q1: whys(pickUp,12146,Why).
A1: Why = and(and(posted(deliver,12142),cond(colour(b2,blue)&nextColour(blue)&
  ~holding(b2),12142)), and(notcond(nextColour(done),12142),pre(~delivered(b2),12144)))
Q2: whyns(goto(dropzone),12146, Why).
(the system explains that goto(dropzone) was actually done at 12148 and gives the plan)
A2: Why = bot
Q3: whyc(colour(b2,blue)&nextColour(blue)& ~holding(b2), 12142, Why)
A3: Why = and(done(putDown,12076),bel(and(colour(b2,blue),~holding(b2)),12076))
Q4: whyc(~holding(b2), 12076, Why).
A4: Why = clessthan(~holding(b2),12076)
```

Figure 5: Actual questions and answers from the implementation

all queries and associated responses below are generated by the implementation, and translated into English. Figure 5 shows the actual queries and responses from the implementation.

The point at which the program behaved unexpectedly was the failure of the pickUp action at line 17. The programmer therefore begins by asking why the program did that action Q1: Why did you do $pickUp$ at 17? (formal: $why_{17}(pickUp)$) This results in the answer A1: Because earlier in the trace the event *deliver* was posted at 15; and it was handled by plan with (true) context condition $colour(b2, blue) \wedge nextColour(blue) \wedge \neg holding(b2)$ (since the previous plan's context condition $nextColour(done)$ was false); and the preceding step in the plan body ($gotoBlock(b2)$) had a true pre-condition ($\neg delivered(b2)$).

The programmer considers these reasons. The event *deliver* being posted is correct, but the plan in question (line 3 of Figure 4), should not have been selected: since the agent is already holding block b3[8], it should proceed to deliver it, rather than picking up another block. The programmer therefore asks Q2: Why didn't you do $goto(dropzone)$ instead? (formal: $\overline{why}_{17}(goto(dropzone))$) This results in the answer that A2: actually the program *did* do $goto(dropzone)$, but later in the execution.

That the correct plan did end up being used suggests to the programmer that there is an issue with either the context condition of the second plan (line 3), or in the ordering of the plans. In order to assess this, the programmer asks Q3: why was the context condition of the second plan believed at the point when the event was posted? (Formal: $why_{15}(colour(b2, blue) \wedge nextColour(blue) \wedge \neg holding(b2))$). This yields A3: because $putDown$ was done at line 12 where $colour(b2, blue) \wedge \neg holding(b2))$ was true.

The programmer considers each of these conditions: the putDown at line 12 is correct, and, since room 2 has been visited, the agent should believe $colour(b2, blue)$. The programmer therefore queries the third condition Q4: why did I believe $\neg holding(b2)$ at 12? (Formal: $why_{12}(\neg holding(b2))$) This yields the answer that A4: $\neg holding(b2)$ has been true since the start of execution. That the condition has not changed seems odd, and this prompts the programmer to consider whether it is correct. They then realise that the condition is wrong: the plan should be prevented from being applicable if the agent is already holding *any* block, in other words, the condition should be $\neg holding(B')$, rather than checking for the specific block B.

It is worth emphasising that this is a very simple example. However, our experience is that even for such simple programs, debugging can be very difficult, since it can require the programmer to work backwards through the causal chain that resulted in a particular action, which can be quite difficult, since the cause and effect can be far apart [6]. One of the difficulties is identifying why a particular context condition was true, when it should not have been. Using a traditional debugging tool, the programmer has to work backwards through execution, searching for a state of the agent's beliefs where the context condition becomes false. This is both time consuming and error-prone, and is actually not supported well by existing tools, since it requires going *backwards* through the execution. By contrast, our question-based debugging system is able to answer such questions ("why did you believe C at N?"), pinpointing for the programmer the location at which the condition became true.

[8]This is inferred from the pickUp at line 14. A prudent programmer might check that the agent does actually believe it is holding block b3 using $why_{15}(holding(b3))$.

5. CONCLUSION

We have proposed the use of question-based debugging for (cognitive) agent programs, and provided formal definitions of question types and their associated answers. These were implemented, and illustrated using a scenario.

Future work includes: (i) Designing and developing a user interface for the system, and using it to conduct an empirical evaluation with users. The evaluations done with Alice and Java [11, 12] showed substantial benefits, and we would expect to see substantial benefits for our approach compared with traditional debugging, given the difficulty (noted in the previous section) of following causal chains. (ii) The current implementation is a prototype, and further development is required to make it more scalable, to deal with multiple agents, and to correctly deal with multiple applicable instances of a single plan. (iii) Extending to handle plans that are triggered by belief additions ($+b : G \leftarrow P$), and extending to handle debugging when there are multiple concurrent intentions[9]. (iv) We would also want to formally state and prove a completeness result for our definitions. (v) Considering whether the reasoning could be formulated within an adbuctive reasoning framework. (vi) Finally, another interesting avenue to explore is *declarative debugging* [17, 26, 27]. Declarative debugging can also be seen as a form of question-based debugging, but it differs in two key aspects. Firstly, the sequence of questions is (usually) [4] determined by software, not by the user (although more recent tools do permit free user exploration). Secondly, in declarative debugging the questions being asked (of the user) are of the form "is this correct?". It would be interesting to consider how to adapt declarative debugging to agent systems, and how to integrate it with the "Why?" approach.

Acknowledgements

I would like to thank Martin Purvis, Stephen Cranefield, and Koen Hindriks for discussions related to this work.

[9]The definition of $why_N(C)$ already permits S to be from a parallel intention, so this extension may be relatively straightforward.

REFERENCES

[1] R. Bordini, M. Dastani, and M. Winikoff. Current issues in multi-agent systems development. In *Post-proceedings of the Seventh Annual International Workshop on Engineering Societies in the Agents World.*, volume 4457 of *LNAI*, pages 38–61, 2007.

[2] J. A. Botía, J. M. Hernansáez, and A. F. Gómez-Skarmeta. On the application of clustering techniques to support debugging large-scale multi-agent systems. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *post-proceedings of the 4th International Workshop on Programming Multi-Agent Systems (ProMAS, 2006)*, volume 4411 of *LNCS*, pages 217–227. Springer, 2007.

[3] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI-Agent System Combining Middleware and Reasoning. In R. Unland, M. Calisti, and M. Klusch, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser, 2005.

[4] D. Cheda and J. Silva. State of the practice in algorithmic debugging. *Electr. Notes Theor. Comput. Sci.*, 246:55–70, 2009.

[5] R. Collier. Debugging agents in agent factory. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *post-proceedings of the 4th International Workshop on Programming Multi-Agent Systems (ProMAS, 2006)*, volume 4411 of *LNCS*, pages 229–248. Springer, 2007.

[6] M. Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, Apr. 1997.

[7] E. E. Ekinci, A. M. Tiryaki, Ö. 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 *LNCS*, pages 173–186. Springer, 2009.

[8] J. J. Gomez-Sanz, J. Botía, E. Serrano, and J. Pavón. Testing and debugging of MAS interactions with INGENIAS. In M. Luck and J. J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *LNCS*, pages 199–212. Springer, 2009.

[9] 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 *LNCS*, pages 31–45. Springer, 2012.

[10] M. Johnson, C. M. Jonker, M. B. van Riemsdijk, P. J. Feltovich, and J. M. Bradshaw. Joint Activity Testbed: Blocks World for Teams (BW4T). In *Workshop on Engineering Societies in the Agents World (ESAW)*, volume 5881 of *LNCS*, pages 254–256. Springer, 2009.

[11] A. J. Ko and B. A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE*, pages 301–310. ACM, 2008.

[12] A. J. Ko and B. A. Myers. Extracting and Answering Why and Why Not Questions about Java Program Output. *ACM Trans. Softw. Eng. Methodol.*, 20(2), 2010.

[13] 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*, pages 335–350. Springer, 2015.

[14] V. J. Koeman, K. V. Hindriks, and C. M. Jonker. Automating failure detection in cognitive agent programs. In *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1237–1246, 2016.

[15] D. N. Lam and K. S. Barber. Comprehending agent software. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 586–593. ACM, 2005.

[16] D. N. Lam and K. S. Barber. Debugging agent behavior in an implemented agent system. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *post-proceedings of the 2nd International Workshop on Programming Multi-Agent Systems (ProMAS, 2004)*, pages 104–125. Springer, 2005.

[17] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[18] C. Nguyen, S. Miles, A. Perini, P. Tonella, M. Harman, and M. Luck. Evolutionary testing of autonomous software agents. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 521–528. IFAAMAS, May 2009.

[19] C. D. Nguyen, A. Perini, and P. Tonella. Experimental evaluation of ontology-based test generation for multi-agent systems. In M. Luck and J. J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *LNCS*, pages 187–198. Springer, 2009.

[20] C. D. Nguyen, A. Perini, and P. Tonella. Goal-Oriented Testing for MASs. *International Journal of Agent-Oriented Software Engineering*, 4(1):79–109, 2010.

[21] C. D. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman, and M. Luck. Evolutionary testing of autonomous software agents. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 521–528, 2009.

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

[23] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols. In *Proceedings of the first international joint conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 960–967. ACM, 2002.

[24] D. Poutakidis, M. Winikoff, L. Padgham, and Z. Zhang. Debugging and testing of multi-agent systems using design artefacts. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-agent Programming: Languages, Tools, and Applications*, chapter 7, pages 215–258. Springer, 2009.

[25] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 42–55. Springer, 1996.

[26] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

[27] L. Sterling and E. Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed*. MIT Press, 1994.

[28] M. Winikoff. An AgentSpeak Meta-interpreter and Its Applications. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *post-proceedings of the 3rd International Workshop on Programming Multi-Agent Systems (ProMAS, 2005)*, volume 3862 of *LNCS*, pages 123–138. Springer, 2005.