Rafael H. Bordini, Louise A. Dennis and Yves Lespérance (eds)
Pre-proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019)
13th-14th May 2019, Montreal, Canada
`cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_27.pdf`

The "Why did you do that?" Button: Answering Why-questions for end users of Robotic Systems

Vincent J. Koeman[1], Louise A. Dennis[2], Matt Webster[2], Michael Fisher[2], and Koen Hindriks[3]

[1] Delft University of Technology, The Netherlands
`v.j.koeman@tudelft.nl`
[2] University of Liverpool, UK
`{L.A.Dennis, M.Webster, MFisher}@liverpool.ac.uk`
[3] Vrije Universiteit Amsterdam, The Netherlands
`k.v.hindriks@vu.nl`

Abstract. The issue of explainability for autonomous systems is becoming increasingly prominent. Several researchers and organisations have advocated the provision of a "why did you do that?" button which allows a user to interrogate a robot about its choices and actions.
We take previous work on debugging cognitive agent programs and apply it to the question of supplying explanations to end users in the form of answers to *why-questions*. These previous approaches are based on the generation of a trace of events in the execution of the program and then answering why-questions using the trace. We implemented this framework in the *agent infrastructure layer* and, in particular, the Gwendolen programming language it supports – extending it in the process to handle the generation of applicable plans and multiple intentions.
In order to make the answers to why-questions comprehensible to end users we abstract away from some events in the trace and employ application specific *predicate dictionaries* in order to translate the first-order logic presentation of concepts within the cognitive agent program in natural language. A prototype implementation of these ideas is provided.

1 Introduction

As Autonomous Systems become more prevalent in society, issues related to the ways in which humans interact with such systems become more important. Among these issues is the question of transparency and, in particular, explainability. Wortham and Theodorou [37], and Sheh [25] (among others) have argued that the ability for a robot (and by extension any autonomous system) to provide explanations of its behaviour helps users develop accurate mental models of the robot's reasoning and so interact better with the robot and develop trust. Charisi et al. [6], Turner [27] and The IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems [26] in particular advocate the provision of a "why did you do that?" button to help the user understand a robot's behaviour.

We take as our focus autonomous systems which employ a cognitive agent to make high level decisions such as [30, 29, 39]. One of the reasons often put forward for the employment of cognitive agents in this role is their in principle ability to explain their decisions to end users. However, in practice, little research has been performed in actually providing such explanations of reasoning.

There are a number of key problems in the provision of explanations. Firstly they require a backward view of the program execution (in contrast to common debugging practice in which a breakpoint is set and the program is then run forwards from the breakpoint) Secondly log files, which are the obvious solution to the first problem tend to be verbose and their production can cause significant performance overheads. These problems are exacerbated when all the information needed to understand why something is taking place must be captured.

In this paper we combine work on the debugging of cognitive agent programs in the Beliefs-Desires-Intentions (BDI) paradigm [18] with work on the provision of explanations for programmers in GOAL [16] and AgentSpeak [33]. Koeman et al. [18] generate an omniscient trace of key events that take place during program execution in a manner which limits the overhead cost of producing the trace. Each event stores enough information about the agent's mental state to reconstruct the state of the program execution at that point. This trace is supported by tools allowing it to be viewed at a high-level of abstraction hiding extraneous information unless a user wants to see it.

We have implemented omniscient tracing in the *Agent Infrastructure Layer* (AIL) [8, 10], a prototyping tool for verifiable interpreters for cognitive agent programming languages, with particular attention to the GWENDOLEN programming language [9] but also consider the extent to which this framework is generic. In applying this framework to the AIL we extended the events considered beyond changes to the agent's mental state to include a number of events involved in the generation of plans and the handling of intentions.

The development of omniscient debugging was driven, in part, by a desire to support programmers in answering why-questions. Programmers can interrogate the high level trace at specific points in the program execution and ask "why did you do that" (as outlined in [16]). Winikoff [33] reports on a similar system constructing why and why-not explanations over traces for AgentSpeak.

We implemented this idea in our AIL-based omniscient debugging framework. We developed an explanation generation framework for end users that is specific to GWENDOLEN, providing explanations at a higher level of abstraction than previously considered, and using *predicate dictionaries* to provide natural language substitutes for application specific logical predicates. This implementation generates explanations when multiple intentions are being executed in an interleaved fashion (something omitted from [33]).

2 Background and Related Work

2.1 Cognitive Agent Programming

At its most general, an *agent* is an abstract concept that represents an *autonomous* computational entity that makes its own decisions [35]. A general agent is simply the encapsulation of some distributed computational component within a larger system. However, in many settings, something more is needed. Rather than just having a system which makes its own decisions in an opaque way, it is increasingly important for the agent to have explicit *reasons* (that it could explain, if necessary) for making one choice over another.

Cognitive agents [3, 22, 36] enable the representation of this kind of reasoning. Such an agent has explicit reasons for making the choices it does. We often describe a cognitive agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. Such agents make decisions about what action to perform, given their current beliefs, goals and intentions. This view of cognitive agents is encapsulated within the Beliefs-Desires-Intentions (BDI) model [21, 22, 23]. Beliefs represent the agent's (possibly incomplete, possibly incorrect) information about itself, other agents, and its environment, desires represent the agent's long-term goals while intentions represent the goals that the agent is actively pursuing (the representation of intentions often includes partially instantiated and/or executed plans and so combines the goal with its intended means).

There are *many* different agent programming languages and agent platforms based, at least in part, on the BDI approach [24, 1, 7, 20, 15]. Agents programmed in these languages commonly contain a set of *beliefs*, a set of *goals*, and a set of *plans*. Plans determine how an agent acts based on its beliefs and goals and form the basis for *practical reasoning* (i.e., reasoning about actions) in such agents. As a result of executing a plan, the beliefs and goals of an agent may change and actions may be executed.

It is generally recognised that debugging BDI agent programs is hard [31, 32] (and by extension that agent behaviour can be difficult to understand even when performing as desired). In particular agents react to events in dynamic environments; events which may combine in unexpected ways and which may be handled by the agent "in parallel" with each other. Furthermore many cognitive agent languages have provision for failure handling which, again, may interact in complex ways with the behaviour of the rest of the program.

2.2 Explanations in Cognitive Agent Systems and "Why" Questions

Ko and Myers [17] created the WHY-LINE tool, which allows developers to pose "why did" or "why didn't" questions about the output of Java programs. A trace is generated in memory through bytecode instrumentation, containing everything necessary for reproducing a specific execution. From this trace, a set of questions and associated answers is generated. The authors note that their approach is not suited for executions that span more than a few minutes or executions that process or produce substantial amounts of data. However, their results do show that the approach enables developers to debug failures substantially faster.

Hindriks [16] and Winikoff [33] both consider a similar model applied to the debugging of cognitive agent programs in GOAL and AgentSpeak respectively. Of these [16] is the earlier and has a more informal treatment than [33] which sought to extend, formalise and implement the proposal. The two approaches are therefore similar in their underlying conception and we use them as the basis for our work. The key idea is that a trace of events is stored as a log. Each event in the trace can be interrogated and an explanation constructed in a systematic way using information either stored in that event and/or by referring to a previous event in the log. For instance the explanation for why some action was executed might be that "the action's preconditions held and a plan was previously selected which contained the action". Explanations can then also be given for why the preconditions succeeded and/or why the plan was selected.

Koeman et al. [18] propose a trace based mechanism for debugging cognitive agent programs. Although concerned with many of the same issues as [16] and [33] (and indeed, intended as support for the mechanisms proposed in [16]) the authors focus on more foundational questions of what information needs to be stored in a trace in order to reconstruct the state of an agent at that point, and the performance overhead of storing such traces for a program. They conclude that if a trace stored the key events in agent execution, namely the changes to the agent's mental state, then the program run could be reconstructed without the significant performance impact associated with storing the full state of an agent at each step in execution. They develop a *space-time visualiser* for these traces which allows a programmer to inspect the trace and query the state of the underlying program at any point.

Hindriks [16] and Koeman et al. [18] consider primarily changes to an agent's mental state (i.e., beliefs and goals) in their tracing and debugging frameworks. Winikoff [33] extends this to include traces and explanations for the selection of plans but assumes that the entirety of a plan is executed before anything else happens. The AIL allows interleaved execution of plans by manipulating intentions. In our work therefore, we integrated the approach in [33] with that of [18] and then extended it to the handling of multiple intentions[4].

A few systems have considered the question of providing explanations specifically for end users of cognitive agent systems. In Harbers [14] explanations of agent behaviour are generated based on the beliefs and goals of the agents using a goal hierarchy paired with a behaviour log. Winikoff et al. [34] presents a similar system but adds the concept of preferences (or *valuings*) to the explanations presented to end users. This was evaluated using hand-crafted explanations generated according to the methodology outlined in the paper.

2.3 The Agent Infrastructure Layer and Gwendolen

The Agent Infrastructure Layer (AIL) [8] is a set of Java classes intended to assist in the development of BDI-style programming languages. Gwendolen [9] is the most mature language in this framework.

The AIL provides default data structures for agents, beliefs, goals, plans and intentions. Individual languages implemented in the AIL define custom reasoning

[4] Though it should be noted that the implementation of omniscient debugging in GOAL also handles GOAL's module mechanism (although this is not reported in depth in [18]) which is not entirely dissimilar to the concept of intention in the AIL.

cycles for agent deliberation. However the toolkit has an underlying assumption that such reasoning cycles will typically involve the following steps in some order:

– Perception which creates sets of new beliefs and removes beliefs that no longer hold.
– Posting events (either as new intentions, or added to existing intentions) when beliefs are acquired or removed and goals are acquired or removed.
– Selecting plans to react to events.
– Selecting among intentions which represent partially processed plans or unhandled events.
– Processing one (or more) steps in an intention which include adding and removing beliefs and goals and executing actions.

These default steps therefore form the core events supported by our implementation of omniscient debugging within the AIL.

Gwendolen Operational Semantics We use GWENDOLEN as our key implementation language. We present here a simplified version of the GWENDOLEN operational semantics which is presented in full in [9]. The semantics presented here assumes all terms are ground (so ignores issues surrounding the handling of unifiers), and ignores a number of language features such as locking and suspending intentions, dropping goals, agent sleeping and waking behaviour, message handling and special cases such as transitions for handling goals that can't be planned. The intention is to present enough information to allow our framework to be understood. This operational semantics is shown in Figure 1. Following [33] we annotate the transitions (expressions above the arrow) with the events that are stored by the omniscient debugger. These are discussed further in Section 3.

The GWENDOLEN reasoning cycle shown here has five stages $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ and $\mathbf{E}$[5]. One transition in each stage is executed in turn. In the semantics we show the stage that a transition applies to with a letter to the right of the rule. A GWENDOLEN agent starts in stage **A** and so (1) is the first rule to apply, followed by (2) and so on. In stage **D** whichever rule applies to the top of the current intention is applied and then the reasoning cycle moves on to stage **E**.

BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some form of *deed stack* (deeds include actions, belief updates, and the commitment to goals). In GWENDOLEN intention structures[6] also maintain information about the event that triggered them (the addition or removal of a belief or the posting of a (sub-)goal). GWENDOLEN aggregates this information: an intention becomes a stack of tuples of an event and a deed. Each tuple associates a particular deed with the event that caused the deed to be placed on the intention. New events are associated with an empty deed, ϵ.

We represent an agent state as a tuple $\langle i, I, B, A \rangle$ where: i is the current intention; I is a queue of intentions $\{i_1, i_2, ..\}$; B is a set of the agent's beliefs; and A is a set of currently applicable plans for the current intention i.

[5] The implementation of GWENDOLEN contains a sixth stage for message handling.
[6] A refinement of the AIL's intention structure which is more general.

$$\frac{\mathcal{S}_{\text{int}}(I \cup \{i\}) = (i', I')}{\langle i, I, B, \emptyset\rangle \xrightarrow{seli(i)} \langle i', I', B, \emptyset\rangle}\,\mathbf{A} \tag{1}$$

$$\frac{\mathcal{G}(i, I, B) = A}{\langle i, I, B, \emptyset\rangle \xrightarrow{genp(A,i);\Gamma} \langle i, I, B, A\rangle}\,\mathbf{B} \tag{2}$$

$$\frac{(e, ds) = \mathcal{S}_{\text{plan}}(A)}{\langle i, I, B, A\rangle \xrightarrow{selp(e,ds,i)} \langle (e, ds)\ \texttt{@}\ \texttt{tl}_i(i), I, B, \emptyset\rangle}\,\mathbf{C} \tag{3}$$

$$\frac{B \models g}{\langle (e, +!g);_i i, I, B, \emptyset\rangle \rightarrow \langle i, I, B, \emptyset\rangle}\,\mathbf{D} \tag{4}$$

$$\frac{B \not\models g}{\langle (e, +!g);_i i, I, B, \emptyset\rangle \xrightarrow{add((e,+!g),i)} \langle (+!g, \epsilon);_i (e, +!g);_i i, I, B, \emptyset\rangle}\,\mathbf{D} \tag{5}$$

$$\frac{}{\langle (e, +b);_i i, I, B, \emptyset\rangle \xrightarrow{crei((+b,\epsilon))} \langle i, I \cup (+b, \epsilon), B \cup \{b\}, \emptyset\rangle}\,\mathbf{D} \tag{6}$$

$$\frac{}{\langle (e, -b);_i i, I, B, \emptyset \ldots\rangle \xrightarrow{crei((-b,\epsilon))} \langle i, I \cup (-b, \epsilon), B \backslash \{b\}, \emptyset\rangle}\,\mathbf{D} \tag{7}$$

$$\frac{\mathbf{do}(a)}{\langle (e, a);_i i, I, B, \emptyset\rangle \xrightarrow{act(a,i)} \langle i, I, B, \emptyset\rangle}\,\mathbf{D} \tag{8}$$

$$\frac{P = \mathbf{Percepts} \quad OP = \{b \mid b \in B \backslash P \wedge percept(b)\}}{\begin{array}{c}\langle i, I, A, B\rangle \xrightarrow{\Pi} \\ \langle i, I \cup \{(\texttt{percept}, +b) \mid b \in P \backslash B\} \cup \{(\texttt{percept}, -b) \mid b \in OP\}, B, A\rangle\end{array}}\,\mathbf{E} \tag{9}$$

Fig. 1. Simplified GWENDOLEN Semantics

A GWENDOLEN program consists of a set of plans, Δ, of the form, $e : \{g\} \leftarrow ds$ (where ds is a sequence of deeds to be executed if event, e is posted and guard, g, follows from the agent's beliefs and goals), a set of initial beliefs, $\mathcal{B}$, and a set of initial goals, Gs. In an agent's initial state the current intention is *null*, the intention set consists of one intention for each of the initial goals provided by the programmer of the form $(\texttt{start}, +!g)$. The belief base is $\mathcal{B}$ and the applicable plans are empty.

(1) governs the selection of intentions. $\mathcal{S}_{\text{int}}$ is an application specific function that selects one intention out of a set of intentions and returns a tuple of the selected intention and the set without that intention in it. By default $\mathcal{S}_{\text{int}}$ operates on a queue data structure and so in general the current intention is placed at the end of the queue and the intention at the top of the queue is selected.

(2) represents the process of inspecting the plan library and finding plans that match the current intention. These are transformed into *applicable plans* and returned by the function $\mathcal{G}$. A plan, $e : \{g\} \leftarrow ds$ matches an intention if e matches the event in the top tuple of the intention, g is a logical consequence of the agent's beliefs and goals (goals are inferred from the events posted in all intentions) and the deed in the top tuple of the intention is ϵ. Applicable plans are an interim data structure that describe how the plan changes the current

intention. An applicable plan describes new tuples to be placed on the top of the intention stack (replacing the existing top tuple). A tuple is created for each deed in ds and associated with e. Where the top deed on the intention is not ϵ ("no plan yet") then $\mathcal{G}$ simply returns the existing top tuple so there is no change to the intentin and it continues to be processed as normal[7].

(3) uses the application specific function $\mathcal{S}_{\text{plan}}$ to pick an applicable plan to be applied. By default, this treats the set as a list and picks the first plan based on the order they appear in the Gwendolen program. We use the syntax (e, ds) @ $\texttt{tl}_i(i)$ to represent the replacement of the top tuple in the intention by the tuples in the applicable plan.

(4), (5), (6), (7) and (8) process the top deed in the intention handling the instruction to add a goal (depending upon whether the goal is already achieved or not), add a belief, drop a belief and execute an action respectively. $(e, d);_i i$ represents the addition of the tuple (e, d) to the top of the intention i. $\mathbf{do}(a)$ represents the execution of an action in some external environment. These rules make a check on the top deed in the intention to see what type it is (e.g. the addition of a belief, the deletion of a goal). We represent these checks implicitly using the notation: a for an action; $+b$ for a belief addition; $-b$ for a belief removal; and $+!g$ for a goal addition.

(9) handles perception. A set of **Percepts** are gathered from the environment. New percepts are added as intentions to add a belief. Out of date percepts (i.e., percepts in the belief base that can no longer be perceived) are handled by creating a new intention to remove them.

3 An AIL-based Framework for Omniscient Debugging Driven Explanations for Cognitive Agents

As noted above, omniscient debugging was developed with the intention of supporting explanations in the form of answering why- and why-not-questions as outlined in [16] and [33]. The implementation of omniscient debugging in GOAL had already identified a number of key events that were needed in an omniscient trace. Since both GOAL and the AIL were implemented in Java it was possible to port much of the framework for efficient storage of event traces directly from GOAL to the AIL. The only necessary step was identifying additional events and agent states that needed storing.

Omniscient debugging for GOAL focused on the changes in agent goals and beliefs as the key events underpinning a trace. We used the analysis from section 2.3 to extend this to [8]:

1. Creation of intention, i: $crei(i)$.
2. Selection of intention, i: $seli(i)$.

[7] This somewhat baroque mechanism has its roots in Gwendolen's origin as an intermediate language into which all BDI languages could be translated [11].

[8] Note this is not the complete set of events shown in figure 1. This is elaborated further in section 3.1

3. Successful evaluation of guard, g for (applicable) plan π, with unifier, θ in intention, i: $bel(\pi, g, \theta, i)$.
4. Selection of an applicable plan, (e, ds) in intention, i: $selp(e, ds, i)$.
5. Execution of action, a, by intention i: $act(a, i)$.
6. Adding or removing goal, g, by intention i: $addg(g, i), delg(g, i)$.
7. Adding or removing belief, b by intention i: $addb(b, i), delb(b, i)$.
8. Modification of intention, i, by adding or removing tuples, ts: $add(ts, i)$, $del(ts, i)$.

Intentions are stored in events in their current state. In order to track the evolution of intentions in traces more easily, we extended intentions with an ID number, k, and will use the notation i_k to represent that intention, i, has ID number, k. Mostly we will omit ID numbers except where they are necessary to understand the construction of explanations.

3.1 Adaptation to Gwendolen

Commands to log these events were embedded in relevant parts of the AIL toolkit, primarily in classes used to implement transition rules in reasoning cycles. This is why in Figure 1 we were able to annotate the transitions in the Gwendolen semantics with the associated events. Given Gwendolen always creates an intention when beliefs are changed and modifies the current intention when a goal is posted we do not use $addg(g, i), delg(g, i)$, or $delb(b, i)$ in Gwendolen traces and only use $addb(b, \texttt{start})$ for the addition of initial beliefs. We rely instead on $crei(i)$ and $add(ts, i)$.

We added a further event just for Gwendolen programs which was the generation of a set of applicable plans, A for some intention, i: $genp(A, i)$.

In Figure 1 we reference two further constructs, Γ and Π, these represent situations where one transition in the semantics generates several events in the trace. Γ logs each successful guard evaluation for plans in Δ as an event in the trace and associates them with the relevant applicable plan. Π logs the creation of the intentions caused by the addition and removal of beliefs following perception.

3.2 Trace Construction and Visualisation

The way in which these events are presented to an agent programmer, i.e. as a log of the execution, is often overlooked. However, an effective and efficient execution log is vital to the debugging process [38]. Effective here means a log contains sufficient information for an agent programmer to get insight into the behaviour of an agent program. Efficient here means that only the information that potentially provides such insights is provided in a log in a concise manner. Arguably, a trace for an agent program contains at least the information needed to construct such a log.

Recording events (i.e., tracing) is an important step that facilitates later visualisation, explanation, and other forms of processing of the execution of an agent program. Moreover, in order to not significantly influence the execution of a traced program, this has to be done as efficiently as possible.

We were able to use the work of Koeman et al. [18] more or less directly to implement tracing in the AIL with these desirable properties.

3.3 From Traces to Explanations

For answering a why-question, a list of recorded events (i.e, a trace) is mapped to a chain of reasons (i.e., an explanation). Reasons thus represent a selection of directly connected events that might span over large parts of the trace. For example, the event of adopting a goal can be directly connected to the event of evaluating the guard of the plan in which it was adopted, in between which many other events could be present in the trace (e.g., the evaluations of other guards). Based on the known ordering of (types of) events, an efficient algorithm can be constructed for deducing reasons from them.

3.4 Why-Questions in Gwendolen

In our initial prototype we consider only the answering of why-questions, leaving consideration of why-not-questions (and their interaction with why-questions) to further work.

In order to generate explanations we need to link each of the traced events to a local explanation as outlined particularly in [33] but also implied in [16]. To do this the explanation had to be grounded in the specific language, Gwendolen, under consideration but nevertheless could be fitted into a general framework.

We consider some event e occurring at step, N, in a trace t and assume, following [33], the existence of a language specific function *why* such that $\mathit{why}(e_N, t)$ returns some representation of an explanation. This representation may recursively refer to some previous event(s) which can also be expanded and so on, if desired, all the way back to the start of program execution. However, our focus on end users means that explanations should have a default cut-off point and are not unwrapped further unless requested by the user. These cut-offs are the reasons why a plan was selected and the reason why an intention was created.

Figure 2 shows the algorithm for constructing *why* for Gwendolen.

As noted, $\mathit{why}(e_N, T)$ can be read as "why did e occur at step N in trace T" where e is one of our traced events. We can also ask why some formula is believed at step N, $\mathit{why}(b_N, T)$, and why some formula is a goal at step N, $\mathit{why}(!g_N, T)$. The left hand side of each equation then constructs an explanation presented as an expression – reasons joined by `because` or `and`. We use $\vee$ to represent situations where there is a choice of possible explanations. So, for instance in (10) the reason an applicable plan is selected is because the guard g was believed and *either* the event e created an intention (as happens when beliefs are posted in Gwendolen) *or* e was added to an intention (as happens when goals are posted in Gwendolen). In constructing an explanation we always select the most recent event that satisfies the formula so, again in (10),

$$
\begin{aligned}
why(selp((e,ds),i_k)_N,T) =& bel((e,ds),g,\theta,{i'}_k)_{N'} \\
& \quad \texttt{and } (crei((e,\epsilon)_k)_{N''} \vee add((e,\epsilon),{i''}_k)_{N''}) && (10)\\
why(b_N,T) =& crei((e,+b))_{N'} \texttt{ because } why(crei((e,+b))_{N'},T) \\
& \vee addb(b,\texttt{start}) && (11)\\
why(!g_N,T) =& add((+!g,\epsilon),i)_{N'} \texttt{ because } why(add((+!g,\epsilon),i)_{N'},T) \\
& \vee crei((e,+!g),i)_{N'} \texttt{ because } why(crei((e,+!g),i)_{N'},T) \\
& && (12)\\
why(add((+!g,\epsilon),i_k)_N,T) =& selp((e,ds),{i'}_k)_{N'} \texttt{ because} \\
& \quad why(selp((e,ds),{i'}_k)_{N'},T) \wedge +!g \in ds \\
& \vee crei((e,+!g)_k)_{N'} \texttt{ because } why(crei((e,+!g)_k)_{N'},T) \\
& && (13)\\
why(act(a,i_k)_N,T) =& selp((e,ds),{i'}_k)_{N'} \texttt{ because} \\
& \quad why(selp((e,ds),{i'}_k)_{N'},T) \wedge a \in ds && (14)\\
why(crei((\texttt{start},d))_N,T) =& \texttt{start} && (15)\\
why(crei((\texttt{percept},d))_N,T) =& \texttt{percept} && (16)\\
why(crei((e,\epsilon)_k)_N,T) =& selp((e,ds),i_k)_{N'} \texttt{ because} \\
& \quad why(selp((e,ds),i_k)_{N'},T) \wedge e \in ds && (17)
\end{aligned}
$$

Fig. 2. Why questions in Gwendolen

$bel((e,ds),g,\theta,i'_k)_{N'}$ means that N' is the most recent step in the trace before N where a matching event occurs. Moreover we need the previous events to have created or manipulated the same intention (and Gwendolen's intention selection mechanism means that other intentions may have been manipulated in between selecting a plan and creating the intention) so we track the intention ID number, k, to ensure we are considering the events occurring in the correct intention.

(11) and (12) ask why something is believed at step N or is a goal at step N and the answer in the first case is that at some previous step an intention was created to add the belief or it was an initial belief, and in the second case that either the goal event was posted to an intention or an intention was created to add the goal (as happens with the initial goals).

(13) asks why some goal was posted to the top of an intention, this is either because a plan was selected previously in the intention which included the posting of the goal as a deed ($+!g \in ds$) or because an intention was created to post the goal.

An action (14) is taken because some plan was selected that included the action in its deeds.

There are three reasons why an intention may have been created: because it is an initial goal or belief (15), because something was perceived (16), or because a plan was selected which included posting the event in its deed stack.

Our framework is similar in conception and presentation to that in [33]. We have in general introduced a pattern of e `because` $why(e, T)$ where [33] normally only has $why(e, T)$. This is motivated in part by our focus on end users instead of programmers: we think it important that explanations mention that "a plan was selected which contained the deed" (which is how we render $selp((e, ds), i) \wedge (d \in ds)$ into natural language) and then state why that plan was selected. Winikoff [33] leaves this implicit and the explanation is only for why the plan was selected. Winikoff [33] is also concerned with a number of other events in a trace – for instance where some deed is not the first to be executed in the body of a plan, then part of the explanation for the its execution includes that the previous deeds were successful. We have taken the view that end users will not generally consider "and the parts of the plan before this succeeded" as part of an explanation, though we may well need to incorporate aspects of this when we look at why-not-questions (i.e., something may not have happened because a previous deed failed).

Other differences arise from differences between the semantics of Gwendolen and AgentSpeak.

3.5 Natural Language Presentation

To make the printing of reasons (and thus also events) suitable for end-users, additional steps are needed. First, we propose a *predicate dictionary* that provides a mapping of predicates to strings (e.g., 'state(X)' to 'the robot is in state X'). Second, an internal translation of specific programming symbols is required (e.g., '+!' to 'added the goal').

3.6 Implementation

The AIL is implemented in Java. Therefore we were able to create an abstract class for events and a framework for storing and presenting visualisations based on the work in [18]. We were then able to create specific event types for the events of interest. We extended the visualiser with an interface to allow why-questions to be asked – specifically "why did you perform this action?", "why did you hold this belief?" and "why did you have this goal?".

We then constructed a specific explanation mechanism for the Gwendolen language based on the algorithm in Figure 2. The details of this required some additional events to be tracked – for instance in order to correctly identify the plan which caused some deed to be placed on an intention it was necessary to extract information both from the generation of plans in stage **B** and the selection of a plan from the set in stage **C** of the Gwendolen reasoning cycle [9].

The AIL comes with a flexible configuration mechanism based on that in JavaPathFinder [28]. We adapted the underlying AIL code to store events in a trace if tracing was enabled.

[9] We omit the gory details here, instructions for accessing the source code can be found in section 6.

We also implemented a pretty printing mechanism in the AIL which allowed languages to customise a pretty printer for the presentation of traces. These pretty printers could be further customised by the use of application specific dictionaries specifying natural language substitutions for predicates appearing both traces and explanations.

This gave us a flexible and extensible framework for implementing omniscient debugging in order to enable why-questions in AIL languages.

4 Evaluation

Our current implementation is a prototype only so a full evaluation has yet to be undertaken. However, it is possible to present initial results.

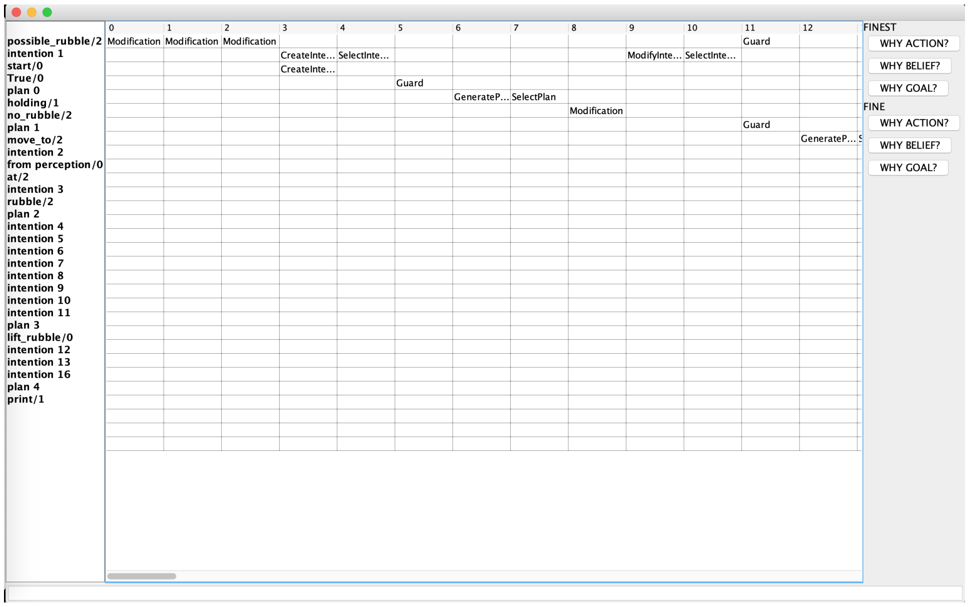
4.1 Traces in Gwendolen for Tutorial examples

The AIL comes with an extensive set of examples based on tutorials for the framework, the Gwendolen language, and the AJPF model-checker [10]. We used these as an ongoing driver for development of our framework – in particular to help settle on appropriate pretty printing conventions. Figure 3 shows part of a pretty printed version of the event trace for one of these examples as it is constructed [10]. Our visualiser for traces is shown in Figure 4.

```
selected Intention 1: add the goal achieve "the robot is holding rubble".
confirmed Intention 1: add the goal achieve "the robot is holding rubble"
can still be processed.
generated 1 applicable plan(s): continue processing: add the goal achieve
"the robot is holding rubble" for an event.
selected continue processing: add the goal achieve "the robot is holding
rubble".
added achieve "the robot is holding rubble" to the agent's goals.
modified intention by posting an event to become Intention 1: respond to
the event start which has no plan yet AND add the goal achieve "the robot
is holding rubble".
selected Intention 1: respond to the event start which has no plan yet
AND add the goal achieve "the robot is holding rubble".
```

Fig. 3. A Pretty Printed Event Trace for a Gwendolen program

[10] NB. It is generally accepted that end users prefer natural language presentations while developers often prefer something more compact so this log presents the events with end users in mind, though it remains much more verbose than is required for an explanation.



	0	1	2	3	4	5	6	7	8	9	10	11	12
possible_rubble/2	Modification	Modification	Modification									Guard	
intention 1				CreateInte...	SelectInte...					ModifyInte...	SelectInte...		
start/0				CreateInte...									
True/0						Guard							
plan 0							GenerateP...	SelectPlan					
holding/1									Modification				
no_rubble/2												Guard	
plan 1													GenerateP... S
move_to/2													
intention 2													
from perception/0													
at/2													
intention 3													
rubble/2													
plan 2													
intention 4													
intention 5													
intention 6													
intention 7													
intention 8													
intention 9													
intention 10													
intention 11													
plan 3													
lift_rubble/0													
intention 12													
intention 13													
intention 16													
plan 4													
print/1													

Fig. 4. Trace Visualiser

4.2 Explanations for Gwendolen for Tutorial examples

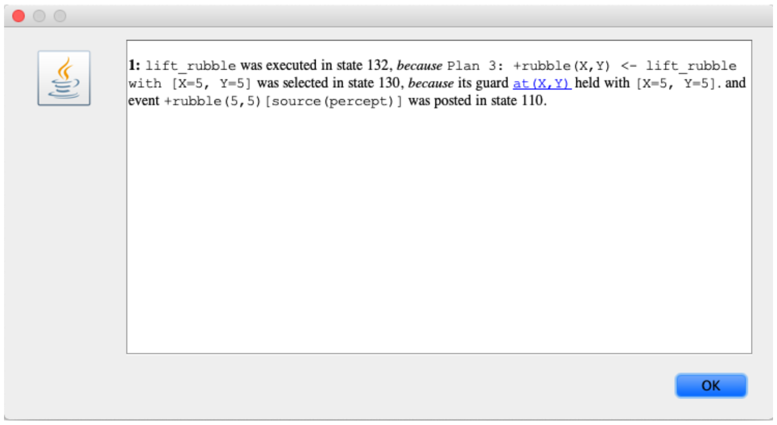
Figure 5 shows an example explanation (for why the robot performed the action `lift_rubble`). Sections of the explanation underlined in blue can be clicked on to generate further explanations – in this case for "why do you believe at(5, 5)".

4.3 Potential Use Case: Self-certifying Offshore Assets

In order to validate our intuitions about appropriate explanations for end users we investigated a prototype agent program for surveying offshore assets such as oil rigs and wind farms [12, 13]. This agent guides an unmanned aircraft that must select a suitable path between the legs of an oil rig based on wind speed, wind direction, and perceived tolerance to risks. Guided by the developers, we produced a predicate dictionary for the application which converted the program's internal representation into natural language – e.g., `enactRoute(route2, t2)` becomes `enact route2 with target t2` and so on.

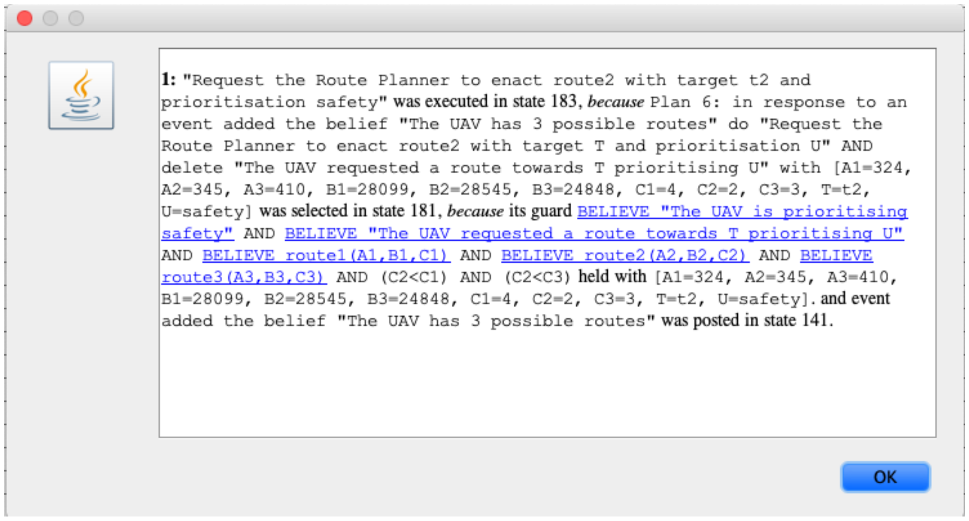
A sample explanation is shown in Figure 6

These explanations were shown to the developers who confirmed that they provided explanations of use to their end users (considered to be experts in unmanned aircraft operation and offshore asset inspection), though obviously further work is needed on the presentation (e.g., performing unifications rather than showing the unifier).



1: `lift_rubble` was executed in state 132, ***because*** `Plan 3: +rubble(X,Y) <- lift_rubble with [X=5, Y=5]` was selected in state 130, ***because*** its guard `at(X,Y)` held with `[X=5, Y=5]`. and event `+rubble(5,5)[source(percept)]` was posted in state 110.

Fig. 5. Generating an Explanation



1: `"Request the Route Planner to enact route2 with target t2 and prioritisation safety"` was executed in state 183, ***because*** `Plan 6: in response to an event added the belief "The UAV has 3 possible routes" do "Request the Route Planner to enact route2 with target T and prioritisation U" AND delete "The UAV requested a route towards T prioritising U" with [A1=324, A2=345, A3=410, B1=28099, B2=28545, B3=24848, C1=4, C2=2, C3=3, T=t2, U=safety]` was selected in state 181, ***because*** its guard `BELIEVE "The UAV is prioritising safety" AND BELIEVE "The UAV requested a route towards T prioritising U" AND BELIEVE route1(A1,B1,C1) AND BELIEVE route2(A2,B2,C2) AND BELIEVE route3(A3,B3,C3) AND (C2<C1) AND (C2<C3)` held with `[A1=324, A2=345, A3=410, B1=28099, B2=28545, B3=24848, C1=4, C2=2, C3=3, T=t2, U=safety]`. and event `added the belief "The UAV has 3 possible routes"` was posted in state 141.

Fig. 6. Explanation of Route Selection

4.4 Traces and Explanations for Other Languages

To evaluate our claim that tracing in the AIL is generic we enabled tracing for another language implemented in the AIL, without any further customisation for the language. The language selected was `pbdi` [4], a reimplementation of a BDI library for Python[11] intended to allow agents written using the library to be verified. We generated an omniscient trace for a simple program in this language (one which stops the operation of a small Pi2Go robot using a command done when the switch on the side of the robot is pressed). A sample trace is shown in Figure 7.

```
added obstacle_left to the agent’s beliefs.
added switch_pressed to the agent’s beliefs.
evaluating the guard of 1 :: +!_aAny||True||done||[]
1 :: +!_aAny||switch_pressed||print("Stopping Agent")||[]
 resulted in True.
evaluating the guard of 2 :: +!_aAny||True||print("Obstacle: ",agent.sensor_value("obstacle_centre"))
 resulted in True.
generated 2 applicable plan(s): 1 :: +!_aAny||True||done||[]
1 :: +!_aAny||switch_pressed||print("Stopping Agent")||[]
, 2 :: +!_aAny||True||print("Obstacle: ",agent.sensor_value("obstacle_centre"))||[]
 for an event.
selected 1 :: +!_aAny||True||done||[]
1 :: +!_aAny||switch_pressed||print("Stopping Agent")||[]
.
created [empty]::
   *  +state||True||print("Stopping Agent")||[]
      +state||True||done||[]
.
selected [empty]::
   *  +state||True||print("Stopping Agent")||[]
      +state||True||done||[]
.
Stopping Agent
performed print("Stopping Agent").

performed done.
removed obstacle_left from the agent’s beliefs.
removed switch_pressed from the agent’s beliefs.
```

Fig. 7. A Sample Trace for *BDIPython*

As can be seen, the lack of language specific pretty printing for plans renders this less readable, but nevertheless a clear trace has been generated of the key events in the execution of the program.

[11] `https://github.com/VerifiableAutonomy/BDIPython`

5 Conclusion

We sought to combine omniscient debugging and answering why-questions for cognitive agent programs in order to generate explanations for end users. To do this we ported omniscient debugging to the AIL toolkit and thus demonstrated its general applicability beyond the GOAL language for which it was developed.

On top of the traces generated by the omniscient debugger we were able to construct explanations for programs in the Gwendolen language. To do this we extended work by [16] and [33] aimed at answering why-questions for developers. This extension involved adding the capability to handle multiple intentions via the tracking of intention IDs and the use of pretty printing and application specific dictionaries to render explanations into natural language.

While this prototype system has yet to be formally evaluated, informal feedback suggests that the end user explanations are appropriate for the intended purpose. A major piece of further work will involve integration of the framework into an application being developed for offshore inspection of oil rigs and wind farms [12, 13] and the evaluation of the generated explanations by the application's users. Work is also needed to integrate the answering of why-not-questions into the framework in order to provide *constrastive explanations* as discussed in [19] which argues that why-questions answer counter-factuals. Minor work focused on improving the presentation of traces and explanations is also required. We would also like to investigate the use of tracing/explanation levels analogous to the logging levels used by Java in order to increase the flexibility of the provided explanations allowing users to "drill down" into more detail if the provided explanation does not meet their needs.

6 Open Data Statement

The source code for the AIL is available from `http://mcapl.sourceforge.net` where the work in this paper can be found in the `omniscient` branch of the git repository. The version discussed here will be archived in the University of Liverpool data catalog in due course.

Acknowledgments

This research was partially funded by EPSRC grants Verifiable Autonomy (EP/LO24845/1) and the Offshore Robotics for Certification of Assets (EP/RO26173) Robotics and Artificial Intelligence Hub.

References

1. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
2. Bordini, R.H., Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Springer (2005)
3. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
4. Bremner, P., Dennis, L.A., Fisher, M., Winfiled, A.F.: On proactive, transparent and verifiable ethical reasoning for robots. Proceedings of the IEEE special issue on Machine Ethics: The Design and Governance of Ethical AI and Autonomous Systems (2019), to Appear
5. Chakraborti, T., Sreedharan, S., Zhang, Y., Kambhampati, S.: Plan explanations as model reconciliation: Moving beyond *Explanation as Soliloquy*. In: subbarao Kambhampati (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence. pp. 156–163 (2016)
6. Charisi, V., Dennis, L.A., Fisher, M., Lieck, R., Matthias, A., Slavkovik, M., Sombetzki, J., Winfield, A.F.T., Yampolskiy, R.: Towards moral autonomous systems. CoRR abs/1703.04741 (2017), `http://arxiv.org/abs/1703.04741`
7. Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming multi-agent systems in 3APL. In: [2], chap. 2, pp. 39–67
8. Dennis, L., Fisher, M., Webster, M., Bordini, R.: Model checking agent programming languages. Automated Software Engineering pp. 1–59 (2011), `http://dx.doi.org/10.1007/s10515-011-0088-x`, 10.1007/s10515-011-0088-x
9. Dennis, L.A.: Gwendolen semantics: 2017. Tech. Rep. ULCS-17-001, University of Liverpool, Department of Computer Science (2017)
10. Dennis, L.A.: The MCAPL Framework including the Agent Infrastructure Layer and Agent Java Pathfinder. The Journal of Open Source Software 3(24) (2018)
11. Dennis, L.A., Bordini, R.H., Farwer, B., Fisher, M., Wooldridge, M.: A common semantic basis for BDI languages. In: Fifth International Workshop on Programming in Multi-Agent Systems (ProMAS'07). Lecture Notes in Artificial Intelligence, vol. 4908, pp. 124–139. Springer (2008), `http://dx.doi.org/10.1007/978-3-540-79043-3_8`
12. Dinmohammadi, F., Page, V., Flynn, D., Robu, V., Fisher, M., Patchett, C., Jump, M., Tang, W., Webster, M.: Certification of safe and trusted robotic inspection of assets. In: 2018 Prognostics and System Health Management Conference (PHM-Chongqing). pp. 276–284 (Oct 2018)
13. Fisher, M., Collins, E., Dennis, L., Luckcuck, M., Webster, M., Jump, M., Page, V., Patchett, C., Dinmohammadi, F., Flynn, D., Robu, V., Zhao, X.: Verifiable self-certifying autonomous systems. In: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 341–348 (Oct 2018)

14. Harbers, M.: Explaining Agent Behaviour in Virtual Training. Ph.D. thesis, SIKS Dissertation Series (2011), no. 2011-35
15. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming: Languages, Tools and Applications. pp. 119–157. Springer US, Boston, MA (2009)
16. Hindriks, K.V.: Debugging is explaining. In: Rahwan, I., Wobcke, W., Sen, S., Sugawara, T. (eds.) PRIMA 2012: Principles and Practice of Multi-Agent Systems. pp. 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
17. Ko, A.J., Myers, B.A.: Extracting and answering why and why not questions about Java program output. ACM Transactions on Sotware Engineering and Methodology 20(2), 4:1–4:36 (2010)
18. Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Omniscient debugging for cognitive agent programs. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. pp. 265–272. IJCAI'17, AAAI Press (2017)
19. Miller, T.: Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence 267, 1–38 (Feb 2017)
20. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: [2], pp. 149–174
21. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In: Proc. 2nd Int. Conf. Principles of Knowledge Representation and Reasoning (KR&R). pp. 473–484. Morgan Kaufmann (1991)
22. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. In: Proc. Int. Conf. Knowledge Representation and Reasoning (KR&R). pp. 439–449. Morgan Kaufmann (1992)
23. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proc. 1st Int. Conf. Multi-Agent Systems (ICMAS). pp. 312–319. San Francisco, USA (1995)
24. Rao, A.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away: Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World. LNCS, vol. 1038, pp. 42–55. Springer (1996)
25. Sheh, R.K.: "Why did you do that?" explainable intelligent robots. In: AAAI-17 Workshop on Human-Aware Artificial Intelligence (2017)
26. The IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems: Ethically aligned design: A vision for prioritizing human well-being with autonomous and intelligent systems. version 2. Report, IEEE (2017)
27. Turner, J.: Robot Rules: Regulating Artificial Intelligence. Palgrave Macmillan (2019)
28. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)
29. Webster, M.P., Fisher, M., Cameron, N., Jump, M.: Formal methods for the certification of autonomous unmanned aircraft systems. In: Proc. 30th Int. Conf. Computer Safety, Reliability and Security (SAFECOMP). LNCS, vol. 6894, pp. 228–242. Springer (2011)

30. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Programming Multi-Agent Systems, LNCS, vol. 7837, pp. 54–71. Springer (2013)
31. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. Journal of Artificial Intelligence Research 51 (2015)
32. Winikoff, M.: BDI agent testability revisited. Autonomous Agents and Multi-agent Systems 31(1094) (2017)
33. Winikoff, M.: Debugging agent programs with Why? questions. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. pp. 251–259. AAMAS '17, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2017)
34. Winikoff, M., Dignum, V., Dignum, F.: Why bad coffee? explaining agent plans with valuings. In: Skavhaug, A., Guiochet, J., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP. LNCS, vol. 9923, pp. 521–534. Spinger (2016)
35. Wooldridge, M.: An Introduction to Multiagent Systems. John Wiley & Sons (2002)
36. Wooldridge, M., Rao, A. (eds.): Foundations of Rational Agency. Applied Logic Series, Kluwer Academic Publishers (1999)
37. Wortham, R.H., Theodorou, A.: Robot transparency, trust and utility. Connection Science 29(3), 24200247 (2017)
38. Zeller, A.: Why Programs Fail, Second Edition: A Guide to Systematic Debugging. Morgan Kaufmann Publisheres Inc., San Francisco, CA, USA (2009)
39. Ziafati, P., Dastani, M., Meyer, J.J., van der Torre, L.: Agent programming languages requirements for programming autonomous robots. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) Programming Multi-Agent Systems. pp. 35–53. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)