AgentSpeaX: Explain, Actually*

Michael Winikoff[1 [0000−0002−5545−7003]], Zahra Daoui[2 [0009−0000−7217−2010]], Andrea Gatti[2 [0009−0003−0992−4058]], Viviana Mascardi[2 [0000−0002−2261−9926]], and Angelo Ferrando[3 [0000−0002−8711−4670]]

[1] Victoria University of Wellington, New Zealand, michael.winikoff@vuw.ac.nz
[2] University of Genova, Italy, zahra.daoui@edu.unige.it, andrea.gatti@edu.unige.it, viviana.mascardi@unige.it
[3] University of Modena and Reggio Emilia, Italy, angelo.ferrando@unimore.it

Abstract. Explainability is a key requirement for autonomous agents operating in interactive and human-facing environments. In Belief-Desire-Intention (BDI) agent architectures, the explicit representation of beliefs, goals, and plans suggests that explanations can be grounded in an agent's actual execution. In this paper, we present *AgentSpeaX*, a conceptual framework for execution-level explainability in AgentSpeak(L) agents, along with a working prototype. AgentSpeaX defines a formal explanation model based on instantiated plan executions, ensuring that explanations are faithful to what the agent *actually* did rather than to abstract plan schemas. The framework supports contrastive explanations and explanation exchange through dedicated communicative acts. We instantiate the formalism for the Jason language and provide a Prolog-based explainer as a reference realization that generates explanations based on Jason agent execution.

Keywords: eXplainable AI · Belief-Desire-Intention · AgentSpeak(L) · KQML

1 Introduction and Motivation

Autonomous software agents are increasingly deployed in interactive, open, and human-facing environments, where their decisions may have significant consequences. In such settings, explainability is essential for trust [23,37,31], accountability [10], and effective interaction: users and developers must understand not only what an agent did, but also how and why its behaviour emerged [34,36,18,16]. Belief–Desire–Intention (BDI) [5,30] architectures are often regarded as particularly amenable to explainability, as they explicitly represent beliefs, goals, and intentions underlying agent behaviour, allowing explanations to be generated in human-meaningful terms [32,33,27,41].

In this paper, we argue that explanations for BDI agents should be *execution-grounded*, *i.e.*, derived directly from the agent's deliberation and execution traces so that they truthfully and faithfully reflect the actual decision-making process. In other words, explanations should account for what the agent *actually did* at runtime, rather than what its abstract plans or design-time models suggest it ought to have done. We introduce *AgentSpeaX*, a conceptual framework for execution-level explainability in AgentSpeak(L) agents. AgentSpeaX defines a formal explanation model based on instantiated

* While we pay homage to "Love Actually", the 2003 romantic comedy directed by Richard Curtis, we emphasize that AgentSpeaX explanations are driven by what agents *actually* did.

plan executions, capturing the contribution of goals, plans, actions, and belief updates to observed behaviour, and ensuring faithful explanations even in the presence of recursion, repeated execution, and failure.

Beyond individual explanations, AgentSpeaX treats explanations as first-class communicative artefacts. Building on established speech-act–based communication semantics, such as those underlying agent communication languages like KQML (e.g., [35,21]), we extend agent communication with explanation-oriented performatives, enabling agents to request and provide explanations about past actions.

We instantiate the formalism for the Jason [4] agent programming language, identifying the semantic adaptations and technical assumptions required to preserve soundness.

We evaluate our work by providing an implementation that demonstrates feasibility, showing that the approach works. The implementation is a Prolog-based explainer that is an executable specification of the explanation rules. Other work has evaluated the effectiveness of explanations [6,15,40,1,20,19,14,2], with the most relevant being recent work evaluating using Beliefs, Desires, and Valuings to explain BDI agents [39,40,41]. These evaluations found that the concepts used for explanation are preferred [40,41], especially valuings, and that contrastive explanations are more effective than full explanations in supporting trust and understanding [39].

In summary, this paper makes the following novel contributions:

i. an explanation-oriented communication mechanism based on explicit performatives, treating explanations as first-class communicative artefacts with operational semantics (Section 4);
ii. an execution-level explanation model for AgentSpeak(L) agents, grounded in time-stamped instantiated plan executions (Section 5);
iii. a disciplined instantiation of the explanation formalism for the Jason agent programming language, clarifying semantic assumptions and limitations (Section 6);
iv. a reference Prolog explainer that demonstrates the practical computation of execution-grounded explanations from Jason execution traces (Section 7).

2 Related Work

Explainability for BDI agents has attracted substantial attention, driven by the intuition that the explicit representation of beliefs, goals, and intentions should facilitate human-comprehensible explanations. Early work leveraged this symbolic structure by exposing internal mental states and plan structures to users (e.g. [15,6]).

Subsequent research introduced interactive mechanisms for explanation generation. Dennis and Oren [11] proposed a dialogue-based approach in which agents with differing views of an execution incrementally reveal traces to focus explanations on user concerns. Panisson *et al.* [29] developed an argumentation-based framework that translates internal reasoning into natural-language explanations using argumentation schemes. While effective in interactive settings, these approaches rely on auxiliary explanatory machinery that operates alongside, rather than directly from, the agent's execution semantics.

Miller's influential review [27] established that human explanations are predominantly contrastive, answering "why P rather than Q?". Building on this insight, Winikoff *et al.* [41] proposed a formal explanation framework grounded in folk-psychological concepts, including preferences (*valuings*), and demonstrated that preference-based explanations are often preferred by users. Recent work has extended this line of research to contrastive explanations for goal selection [26] and to explanations that anticipate human misconceptions [7]. Our work extends the Winikoff *et al.* framework by grounding explanatory factors in *time-stamped execution traces*, enabling faithful explanations under recursion and repeated execution—limitations of their original atemporal model over abstract goal–plan trees.

Recent research has emphasised tailoring explanations to different audiences. Yan *et al.* [42] proposed a multi-level framework generating explanations at implementation, design, and domain levels from Jason execution logs. Mauri and Mascardi [25] similarly advocated abstracting runtime traces into narratives as a basis for explanation. Rodriguez and Thangarajah [33] introduced the XAg paradigm, promoting explainability by design via reusable explanation patterns. While these approaches recognise the importance of execution traces, they typically treat trace extraction and explanation generation as engineering concerns, without providing formal semantics for the explanation mechanism.

The emergence of generative and neuro-symbolic BDI agents introduces new explainability challenges. Approaches using large language models for planning or reasoning (e.g. [8,17]) increase adaptability but may weaken traceability, as generated plans lack stable symbolic structure. ChatBDI [12,13] exemplifies this trend by integrating LLM-generated language into BDI agents via KQML-based communication. Nevertheless, explanations ultimately require grounding in concrete symbolic execution.

Unlike prior work that derives explanations from abstract plan schemas or auxiliary explanatory modules, AgentSpeaX computes explanations directly from *instantiated plan executions* recorded in time-stamped traces. This ensures faithful accounts of what agents *actually did*, even in the presence of recursion, repeated execution, and dynamic variable instantiation. Moreover, by extending KQML with dedicated AskWhy and TellWhy performatives, we treat explanations as first-class communicative artefacts with formal operational semantics (an aspect not addressed by existing BDI explainability frameworks).

3 Preliminaries

This section recalls the minimal AgentSpeak(L) and KQML concepts required to define execution-level explanations, following the operational semantics in [35].

3.1 AgentSpeak(L)

AgentSpeak(L) is a logic-based language for programming BDI agents [30]. Beliefs are first-order atomic formulae $b ::= P(t_1, \ldots, t_n)$, where P is a predicate symbol and t_i are first-order terms.

Achievement goals have the form $!at$, where at is an atomic formula. Actions are written similarly to predicates, $a ::= A(t_1, \ldots, t_n)$, but denote operations affecting the environment or the agent state.

Plans define how agents react to events and are written as

$$p ::= te : ctxt \leftarrow h$$

where te is a triggering event, $ctxt$ is a belief context, and h is the plan body. Triggering events are generated by belief or goal addition and deletion:

$$te ::= +b \mid -b \mid +g \mid -g.$$

Plan bodies consist of actions, belief updates, test goals, and achievement goals:

$$h ::= a \mid g \mid ?c \mid +b \mid -b \mid h; h'.$$

AgentSpeak(L) operational semantics is based on a transition system over configurations $\langle ag, C, M, T, s\rangle$ [35]. The agent program ag consists of a belief base and a plan library. The agent's circumstance is $C = \langle I, E, A\rangle$, where I is the set of intentions (each an intention stack of partially instantiated plans), E is the set of events of the form (te, i), with i either an intention or the empty intention $\top$ for external events, and A is the set of actions to be executed in the environment. Communication is modelled by $M = \langle In, Out, SI\rangle$, where In and Out are the agent's inbox and outbox, respectively, and SI records suspended intentions awaiting replies to communication acts. Temporary information used within a reasoning cycle is stored in $T = \langle R, Ap, \iota, \epsilon, \rho\rangle$, where R and Ap denote the sets of relevant and applicable plans, and ι, ϵ, and ρ record the intention, event, and plan currently under consideration. Finally, s denotes the current step of the reasoning cycle (e.g., message processing, event selection, plan retrieval, intention execution, or intention clearing).

3.2 KQML Communication

One of the earliest formal accounts of KQML semantics was given by Labrou and Finin [21], building on the action-theoretic treatment of speech acts by Cohen and Perrault [9]. In this view, communicative acts are modelled as actions with pre- and post-conditions over agents' mental states. In particular, `Tell` updates the receiver's beliefs, while `Achieve` requests the receiver to attempt to make a given condition true in the environment [22].

In AgentSpeak(L), KQML messages have the form $\langle mid, id, ilf, cnt\rangle$, where mid is a message identifier, id denotes sender or receiver, ilf is the performative, and cnt the content. Communication is often asynchronous: incoming messages are stored in an inbox M_{In}, outgoing messages in M_{Out}, and one message is processed per reasoning cycle. Messages of type ask (`AskIf`, `AskAll`, `AskHow`), however, are synchronous and suspend the current intention until a reply is received. Accordingly, the operational semantics of the `.send` action distinguishes two cases. In the transition rules below, primed components (e.g., M'_{Out} or C'_I) denote the updated version of that component after the transition, while all other components remain unchanged.

$$\text{(ExecActSndAsk)}\ \frac{T_\iota = i[head \leftarrow .\text{send}(id, ilf, cnt); h] \quad ilf \in AskSet}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M', T, ProcMsg\rangle}$$

$$
\begin{array}{ll}
where & AskSet = \{AskIf, AskAll, AskHow\} \\
& M'_{Out} = M_{Out} \cup \{\langle mid, id, ilf, cnt\rangle\} \\
& M'_{SI} = M_{SI} \cup \{(mid, i[head \leftarrow h])\} \\
& \qquad \text{with } mid \text{ a new message identifier;} \\
& C'_I = (C_I \setminus \{T_\iota\})
\end{array}
$$

$$
\text{(ExecActSnd-noAsk)}\ \frac{T_\iota = i[head \leftarrow .\text{send}(id, ilf, cnt); h] \quad ilf \notin AskSet}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M', T, ClrInt\rangle}
$$

$$
\begin{array}{ll}
where & AskSet = \{AskIf, AskAll, AskHow\} \\
& M'_{Out} = M_{Out} \cup \{\langle mid, id, ilf, cnt\rangle\} \\
& \qquad \text{with } mid \text{ a new message identifier;} \\
& C'_I = (C_I \setminus \{T_\iota\}) \cup \{i[head \leftarrow h]\}
\end{array}
$$

The rule *ExecActSndAsk* defines the semantics of sending an ask-type message. A well-formed message is added to the outgoing mailbox, and the current intention is suspended and stored among suspended intentions until a reply is processed. The rule *ExecActSnd-noAsk* defines the semantics of sending a non-ask message. In this case, the message is added to the outgoing mailbox, but the intention is not suspended. The `.send` action is removed from the intention, which is then returned to the intention set and proceeds to the clearing phase of the reasoning cycle. We also have a rule *MsgExchg* (not shown due to space) which provides a semantics to KQML communication at the MAS level: when agent id_i sends a message to agent id_j, the outgoing message $\langle mid, id_j, ilf, cnt\rangle$ is removed from the mailbox of id_i and added to the mailbox of id_j, while the environment env remains unaffected. The processing of received messages at the agent level is defined separately by the local message-handling rules introduced in Section 4.2.

In Jason, message sending is realised by the internal action `.send(Receiver, Performative, Cnt)` (with additional optional parameters for synchronous ask-type messages). Invoking `.send` results in the transmission of a KQML message with the given performative and content to the specified receiver(s).

4 Formalization of KQML-X

KQML-X extends KQML with message types related to explanation: AskWhy and TellWhy. The semantics of these are defined in terms of the explanatory concepts and the agent's mental model in order to ensure that a TellWhy message includes all (and only) the relevant explanatory factors. Section 4.1 therefore provides a full definition of the explanatory factors as a function $Explain$. We next introduce the execution-level assumptions underlying explanation generation, and then (§4.1) formalize the notion of explanation and its grounding in the agent's goals, beliefs, and their evolution over time. Finally, we use this to define the semantics of the new KQML-X message types (§4.2).

Overview of technical assumptions. The formalisation presented in this section relies on a number of execution-level assumptions that clarify how explanations are extracted from AgentSpeak(L) and Jason executions. In particular, we assume access to a time-stamped trace recording the execution state (grounded plan instances, beliefs) and executed *changers*. We define a "changer", in the AgentSpeak(L) context, to be any element that may appear in the body of a plan: internal and user-defined actions, actions on the environment, update of beliefs, test and achievement goals. Action preconditions and postconditions are used only when explicitly available; when they are unknown or underspecified, the explainer proceeds conservatively. Notably, internal actions whose effects and preconditions are not explicitly modelled are ignored for explanatory purposes, ensuring that explanations are derived only from observable and semantically grounded execution information. These assumptions are summarised here to guide the reader, and discussed in detail in Section 6.

4.1 Explanation

An explanation is given in response to a query of the form "why did you do action X?" or, more generally, to a *contrastive* query such as "why did you do X rather than F?". Following Winikoff *et al.* [41], we model an explanation as a *set of explanatory factors*, each capturing a distinct contribution to the agent's deliberation and involving a *desire* D, a *belief* B, or a *valuing* V preference [24].

In contrast to the original formulation, which is atemporal, we extend this model to account explicitly for the temporal structure of BDI deliberation and execution. Accordingly, explanatory factors are annotated with the deliberation or execution time t at which they contributed. Formally:

$$F ::= \mathsf{D}{:}(G,t) \mid \mathsf{B}{:}(C,t) \mid \mathsf{V}{:}(P_1{<}P_2,t),$$

where G is a goal, C is a condition, $P_1{<}P_2$ indicates that plan P_2 was preferred to plan P_1, and t denotes the deliberation or execution point at which the factor contributed.

For example, a cleaning robot that is asked why it picked up garbage at a certain time point might respond with explanatory factors such as: *I believed there was garbage at my location at the time point*, *I desired to take the garbage to the rubbish bin*, and *I preferred to take the garbage to the bin over continuing to explore*.

Conditions and action preconditions. In AgentSpeaX, changers are treated abstractly as execution steps that may be associated with preconditions and postconditions. While such conditions are not part of the core AgentSpeak(L) syntax, we assume that they are available when changers correspond to internal actions, environmental actions with an explicit model, or test goals (`?c`). In Section 6 we show how, in Jason, test goals and belief updates are systematically used to bridge this abstraction.

Formally, we use the term *changer precondition* to denote any condition C such that the execution trace records that C was checked and found to hold at the deliberation or execution step in which the corresponding changer was performed. This notion is trace-based and independent of the concrete AgentSpeak(L) syntax used to realise the check. When preconditions or postconditions are unavailable, the explanation mechanism proceeds conservatively.

Execution traces. To connect explanatory factors with the temporal structure of BDI reasoning, we assume that each agent ag produces an execution trace $tr(ag)$ consisting of a sequence of time-stamped *states* and *changers*. Each state records, at least, the belief base and the instantiated plans that were available in the plan library at that time. We write $tr(ag,t)_{|B}$ (resp. $tr(ag,t)_{|Pl}$) for the belief (resp. instantiated plan) component of the trace at time t. We use $tr(ag,t)_{|Ch}$ to denote the changer executed at time t. We introduce a base explanation function: $Explain^*(ag, tr(ag), x, t)$, which returns the set of *all* explanatory factors that justify the agent's decision to perform or achieve x at time t. The contrastive explanation function $Explain(ag, tr(ag), x, t, f)$ is defined as a projection of $Explain^*(ag, tr(ag), x, t)$ guided by the contrastive alternative f. The optional argument f denotes a contrastive alternative and is used to guide the selection of explanatory factors relevant to a "why x rather than f?" query. Intuitively, contrastive explanation focuses on those factors that distinguish the execution of x from that of f. Formally, $Explain(ag, tr(ag), x, t, f)$ is obtained from $Explain^*(ag, tr(ag), x, t)$ by: (i) filtering out explanatory factors that would equally support the execution of f, and (ii) retaining only contrastive belief and valuing factors that refer to alternatives relevant to f. See Winikoff [39] for details. When no explicit contrast is specified, we write $Explain(ag, tr(ag), x, t)$ as shorthand for $Explain^*(ag, tr(ag), x, t)$. Time is represented abstractly as a discrete execution index reflecting the order of deliberation and execution steps recorded in the trace, without assuming any particular temporal logic. For all of the rules below we assume that $x = tr(Ag,t)_{|Ch}$, i.e. that when asking "why did you do x at time stamp t?" the trace actually includes x being done at t (if this is not the case, then the rules below do not apply, and the answer is some form of "actually, I didn't do x at t").

Desire-based factors. Let $P = (e : C \leftarrow B)$ be a plan in the agent's plan library, where $trigger(P) = e$ denotes its triggering event and $body(P) = B$ denotes its body. If a changer x occurs at time t within the execution of a plan P adopted (*i.e.*, instantiated) to pursue a goal G at time $t_s \leq t$, then the pursuit of G constitutes an explanatory factor:

$$\frac{P \in tr(ag,t_s)_{|Pl} \qquad x \overset{*}{\in} body(P) \qquad trigger(P) = G}{\mathsf{D}{:}(G,t_s) \in Explain^*(ag, tr(ag), x, t)}.$$

Here, $\overset{*}{\in}$ denotes *recursive membership* in a plan body, defined over a plan library Π as: $x \overset{*}{\in} B \;\equiv\; (x \in B) \;\vee\; \big(y \in B \;\wedge\; (y : C \leftarrow B') \in \Pi \;\wedge\; x \overset{*}{\in} B'\big)$.

This definition assumes that explanation is performed over a grounded execution trace, in which plan bodies and triggering events are instantiated to ground (propositional) actions and goals. Accordingly, $\overset{*}{\in}$ is defined over ground plan bodies and does not account for first-order unification or variable propagation across recursive plan invocations. Extending $\overset{*}{\in}$ to operate over first-order plan schemas would require explicit unification mechanisms, which we deliberately leave outside the scope of this work to keep presentation simple.

Belief-based factors. Belief factors arise from changer preconditions and plan context conditions.

Changer preconditions. If changer x was executed at time t and had a precondition C that was believed at that time, then the belief in the residual of C with respect to previously established effects contributes to the explanation:

$$\frac{tr(ag,t)_{|B} \models pre(x) \qquad C' = Residual(pre(x), PostBefore(x)) \qquad C' \not\equiv \top}{\mathsf{B}{:}(C',t) \in Explain^*(ag, tr(ag), x, t)}.$$

Here, $PostBefore(x)$ denotes the set of propositional facts established by the post-conditions of changers that are guaranteed to have occurred before x in the execution trace. The function $Residual(C,S)$ returns a propositional formula C' such that $S \wedge C' \models C$, capturing a sufficient part of the condition C that is not already entailed by S. Residual computation serves an explanatory filtering role: it removes from a precondition only those literals already ensured by prior execution, without aiming at logical minimisation. As a result, the returned residual is guaranteed to be sound but not necessarily minimal.

Algorithm 1 (right) gives a simple, sound procedure for computing $Residual(C,S)$ at the level of propositional literals, under the assumption that C is given in disjunctive normal form and that S consists of atomic propositions derived from the execution of changers in the trace. The algorithm processes disjunctive conditions by handling each conjunctive clause independently and removing only those literals already entailed by S.

Algorithm 1 $Residual(C,S)$

```
R ← ∅
for all conjunctive clauses D of C do
    D' ← { l ∈ D | l ∉ S }
    if D' ≠ ∅ then
        R ← R ∪ {⋀ D'}
    end if
end for
if R = ∅ then
    return ⊤
else
    return ⋁ R
end if
```

Plan context conditions. If plan P was selected (*i.e.*, instantiated) at time t_s and its context condition C held at that time, then for any changer x occurring within the execution of P at time $t \geq t_s$, the belief in C contributes to the explanation:

$$\frac{P \in tr(ag,t_s)_{|Pl} \qquad x \overset{*}{\in} body(P) \qquad tr(ag,t_s)_{|B} \models context(P)}{\mathsf{B}{:}(context(P),t_s) \in Explain^*(ag, tr(ag), x, t)}.$$

Alternative context conditions. Consider a triggering event for which multiple plans were available. If plan P was selected at time t_s but an alternative plan Q was not selected because its context condition C did not hold at that time, then the failure to establish C constitutes a contrastive explanatory factor:

$$\frac{x \overset{*}{\in} body(P) \qquad trigger(P) = trigger(Q) \qquad tr(ag,t_s)_{|B} \not\models context(Q)}{\mathsf{B}{:}(\mathsf{naf}(context(Q)),t_s) \in Explain^*(ag, tr(ag), x, t)}.$$

Here, $\mathsf{naf}(C)$ denotes negation-as-failure, reflecting the operational interpretation of failed condition checking in Prolog and Jason.

Valuing (preference) factors. We say that a plan is *applicable* at time t_s if its triggering event matches the current changer and its context condition is entailed by the belief base recorded in the execution trace at time t_s. Suppose that multiple plans were applicable

for the same triggering event at time t_s, and that the agent selected plan P_2 over another applicable plan P_1 due to a preference relation. In this case, the application of this preference yields a valuing factor:

$$\frac{\begin{array}{ccc} x \overset{*}{\in} \mathit{body}(P_2) & tr(ag,t_s)_{|B} \models \mathit{context}(P_1) & tr(ag,t_s)_{|B} \models \mathit{context}(P_2) \\ & \mathit{trigger}(P_1) = \mathit{trigger}(P_2) & P_2 \succ P_1 \end{array}}{\mathsf{V}{:}(P_1{<}P_2, t_s) \in \mathit{Explain}^*(ag, tr(ag), x, t)}.$$

Here, $P_2 \succ P_1$ denotes that the agent preferred plan P_2 over plan P_1 at time t_s when both were applicable for the same triggering event. This preference relation is left abstract: it may arise from an explicit plan ordering in the plan library, from designer-specified priorities, from deliberative plan-evaluation criteria, or from other architecture-specific mechanisms for resolving plan selection. The rule assumes that the execution trace records sufficient information about plan applicability and preference application to support contrastive explanations among alternative plans.

4.2 KQML-X Message Receipt Rules

We extend the local message-handling semantics with two new performatives, AskWhy and TellWhy, which support explanation-oriented queries and responses. These additions follow the same pattern used for other ask-type performatives such as AskIf, AskAll, and AskHow. On the sending side, no new operational rule is required: the standard ExecActSndAsk rule (see Section 3.2) already handles all ask-type communicative acts, and we simply extend its domain so that

$$\mathit{ilf} \in \{\mathit{AskIf},\ \mathit{AskAll},\ \mathit{AskHow},\ \mathsf{AskWhy}\}.$$

As a consequence, sending an AskWhy message suspends the current intention and enqueues the message in the outgoing mailbox, as for other ask-type illocutionary forces. On the receiving side, AskWhy and TellWhy require additional processing rules. An incoming AskWhy request specifies a changer x and an execution time t. Upon reception, the agent computes an explanation using $\mathit{Explain}(ag, tr(ag), x, t, f)$ (Section 4.1). If x does not occur at time t in the execution trace, the function returns the empty set, indicating that no execution-grounded explanation is available. The explanation is returned as a TellWhy message. When received as a reply to a suspended AskWhy request, the corresponding intention is resumed; when received unsolicited, the explanation is simply made available. In both cases, the message provides a structured set of explanatory factors that can be inspected, rendered, or further queried. Here, $S_M(M_{In})$ denotes the selection function that returns the message chosen for processing from the inbox M_{In} in the current reasoning cycle.

$$\text{(ProcMsgAskWhy)}\ \frac{\begin{array}{c} S_M(M_{In}) = \langle mid, id,\ \mathsf{AskWhy},\ (x,t,f)\rangle \\ E = \mathit{Explain}(ag, tr(ag), x, t, f) \end{array}}{\begin{array}{c} \langle ag, C, M, T, \mathit{ProcMsg}\rangle \rightarrow \\ \langle ag, C, M', T, \mathit{ProcMsg}\rangle \end{array}}$$

$$\text{where}\quad M'_{In} = M_{In} \setminus \{\langle mid, id,\ \mathsf{AskWhy},\ (x,t,f)\rangle\}$$
$$M'_{Out} = M_{Out} \cup \{\langle mid, id,\ \mathsf{TellWhy},\ E\rangle\}$$

$$\text{(ProcMsgTellWhy)}\ \frac{\begin{array}{c} S_M(M_{In}) = \langle mid, id,\ \mathsf{TellWhy},\ E\rangle \\ (mid, i) \notin M_{SI} \end{array}}{\begin{array}{c} \langle ag, C, M, T, ProcMsg\rangle \rightarrow \\ \langle ag, C, M', T', SelEv\rangle \end{array}}$$

where $M'_{In} = M_{In} \setminus \{\langle mid, id,\ \mathsf{TellWhy},\ E\rangle\}$
$T' = \langle R, Ap, \iota, \varepsilon, \rho, E\rangle$

$$\text{(ProcMsgTellWhyRepl)}\ \frac{\begin{array}{c} S_M(M_{In}) = \langle mid, id,\ \mathsf{TellWhy},\ E\rangle \\ (mid, i) \in M_{SI} \end{array}}{\begin{array}{c} \langle ag, C, M, T, ProcMsg\rangle \rightarrow \\ \langle ag, C', M', T', SelEv\rangle \end{array}}$$

where $M'_{In} = M_{In} \setminus \{\langle mid, id,\ \mathsf{TellWhy},\ E\rangle\}$
$M'_{SI} = M_{SI} \setminus \{(mid, i)\}$
$C'_I = C_I \cup \{i\}$
$T' = \langle R, Ap, \iota, \varepsilon, \rho, E\rangle$

The temporary structure T follows the standard AgentSpeak(L) definition and is preserved across transitions, except for an auxiliary component used to store reply-related information. Accordingly, T' is identical to T except that this auxiliary component is populated with the received explanation E. When a TellWhy message is received as a reply to a suspended AskWhy request, the corresponding intention is resumed, following the standard AgentSpeak(L) reply-handling mechanism. When TellWhy is received unsolicited, the explanation is stored without affecting the intention structure.

This design deliberately leaves open how explanations are subsequently exploited. An agent may render them for a human user, inspect them to support follow-up queries, or reason over them to guide further deliberation or interaction. Different usages correspond to different application scenarios, ranging from human–agent interaction to agent–agent explanation exchange (e.g. [28]).

Together, the rules for AskWhy and TellWhy integrate explanation-oriented communication into the agent's operational semantics in a manner consistent with other ask-based interactions. Because the sending side leverages the existing ExecActSndAsk rule, no structural changes to the reasoning cycle are required: there are no new components in the agent's state, and the execution of existing constructs is not affected. The only change is to define the semantics of the new KQML-X message types. The receiver-side rules provide a computationally grounded semantics for explanation exchange, enabling agents to request and obtain explanations grounded in their internal goals, beliefs, and value-based decision traces.

5 AgentSpeaX: Extending AgentSpeak(L) with Explanations

This section describes how the explanation framework introduced in Section 4 can be realized at the level of AgentSpeak(L) agents. Our focus is on the architectural and representational aspects required to support explanation, independently of any particular AgentSpeak(L) implementation.

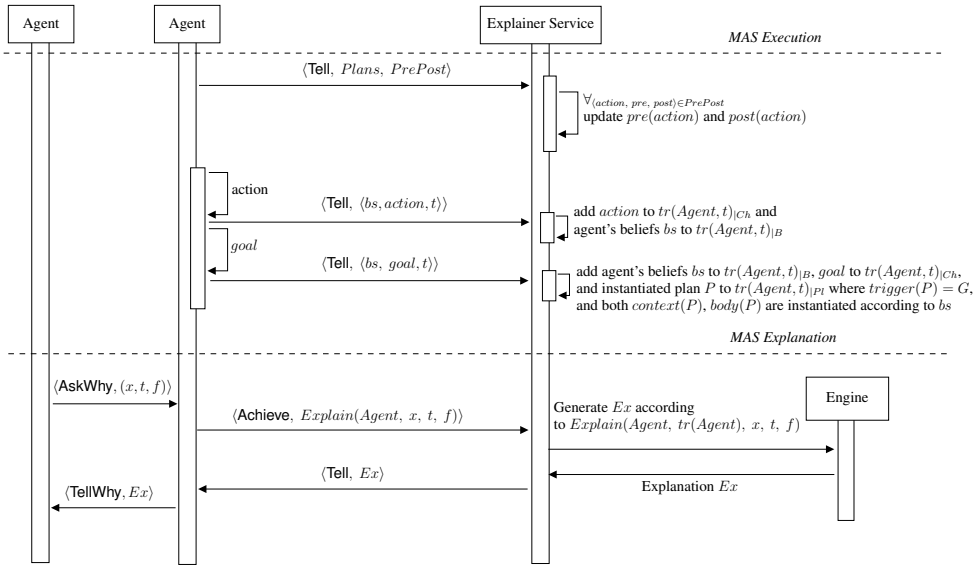

Fig. 1. Sequence diagram of the interaction between an agent and the explainer to answer a AskWhy performative message.

Following the architecture proposed by Winikoff [38], we treat explanation as a separate reasoning component that operates alongside an AgentSpeak(L) agent. Rather than embedding explanation logic into the agent interpreter, AgentSpeaX relies on an external *explainer* that reasons over a symbolic representation of the agent's execution. This separation preserves the autonomy and semantics of the original agent while enabling faithful, execution-grounded explanations.

Figure 1 illustrates the interaction between an AgentSpeak(L) agent and the explainer component. During normal execution, the agent incrementally reports relevant execution information, such as executed actions, pursued goals, belief updates, and instantiated plans, to the explainer. This information is used to maintain a structured, time-indexed representation of the agent's deliberation and execution history, namely the trace $tr(ag)$. When the agent later receives an AskWhy query, the explainer uses this accumulated representation to compute an explanation according to the formal rules defined in Section 4.1, and returns the result as a TellWhy message.

This architectural perspective abstracts away from the internal details of the AgentSpeak(L) interpreter and provides a clean interface between agent execution and explanation. In particular, it makes explicit which execution-level information must be exposed by the agent, without constraining how that information is produced or collected. In the next subsection, we discuss how this approach can be applied to Jason, and identify the additional adaptations and assumptions required to align the explanation framework with Jason's concrete semantics.

6 Instantiating the Explanation Formalism for Jason

The explanation principles introduced in Section 4.1 are defined over an abstract AgentSpeak(L) semantics. Here, we instantiate these principles for *Jason*, identifying the semantic adaptations and assumptions required to derive execution-grounded explanations from concrete executions. The discussion is conceptual rather than implementation-oriented. Nevertheless, most components of the picture have been implemented, and the implementation of the others is under way. As discussed in [3, §4], the instrumentation has been implemented by extending the Jason `Agent` and `AgArch` classes. This means that the programme does not need to modify their code, other than selecting these extended classes. What we have not implemented yet - just for time constraints, since all the needed information is already there in the extracted trace - is the filtering operation that extracts the components needed by the Prolog explainer from the trace, in the format needed by the explainer.

Instantiating the formalism for Jason requires accommodating practical language features not captured by the idealised semantics, including negation in context conditions, internal actions, and Jason's deterministic execution model. The remainder of this section presents a disciplined instantiation that preserves the conceptual structure of explanations while enabling their sound extraction from real Jason executions.

Core Formal Extensions

(E1) Negation and Add/Delete Postconditions. We allow negated literals in plan context conditions and action preconditions, and distinguish between positive ("add") and negative ("delete") postconditions when such information is available. This enables more precise belief-based explanations, in particular when filtering preconditions using the effects of earlier execution steps. In AgentSpeak(L) and Jason, negation is treated operationally rather than declaratively: negated conditions are checked via negation-as-failure in context conditions and test goals, while belief deletion is explicitly recorded in the execution trace. Accordingly, negative conditions and delete effects are handled by the explainer only when they are manifested as observed condition checks or belief updates in the trace, and not by assuming an explicit action model. The explainer does not depend on the availability of negated preconditions or delete postconditions. When such information is unavailable, as is common in standard Jason programs, the explainer proceeds conservatively, possibly yielding less specific but always sound explanations.

(E2) Addition of Jason Step Types. In addition to actions, Jason plan bodies may include belief updates and test goals. We normalize these constructs by treating them as special actions:

$$?c \;\mapsto\; a_{?c} \text{ with precondition } c \text{ and no postconditions,}$$

$$+b \;\mapsto\; a_{+b} \text{ with postcondition } b, \qquad -b \;\mapsto\; a_{-b} \text{ with postcondition } \neg b.$$

Belief replacement, written in Jason as $-+b$, is treated as the sequential execution of belief deletion followed by belief addition, that is:

$$-+b \;\mapsto\; a_{-b}\,;\, a_{+b}.$$

In Jason, belief deletion is often non-ground: the action a_{-b} removes all beliefs that unify with b in the belief base. This normalization preserves the operational effect of belief updates while allowing them to be handled uniformly within the explanation formalism. This normalization allows all steps in a plan body to be handled uniformly by the explanation rules defined in Section 4.1, while preserving the operational semantics of belief update in Jason.

(E3) Sequential OR Semantics for Plan Selection. Jason selects plans using sequential OR semantics: plans are considered in the order in which they appear, and the first applicable plan is chosen. Accordingly, we adapt the explanation rules so that contrastive belief factors arise from earlier plans in the sequence whose context conditions did not hold. Explicit valuing (preference) factors are only generated when additional preference information is available.

Technical Assumptions for Sound Explanation To ensure that explanations can be extracted unambiguously from execution traces, we adopt the following assumptions. These assumptions do not compromise soundness: they may result in explanations that include additional factors, but not in incorrect ones.

(A1) Action Disambiguation. Each changer symbol appears in the body of at most one plan instance. When this is not the case, changers must be syntactically disambiguated.

(A2) Partial Observability of Preconditions, Effects, and Internal Actions. Only explicitly available preconditions and postconditions are used in explanation generation. When changer preconditions or postconditions are unknown or unavailable (as is common in standard AgentSpeak(L) programs) no corresponding action-precondition or postcondition-based belief factors are generated. Similarly, internal actions whose pre- or postconditions are not explicitly modelled are ignored for explanatory purposes, except where their effects are directly observable in the execution trace. In all these cases, the explainer proceeds conservatively, relying only on observed execution information. This may result in less detailed explanations, but never in unsound ones.

(A3) Restricted Belief Rule Expansion. For explanation purposes, we allow the syntactic expansion of simple belief rules whose conclusions can be directly grounded in the current belief base. In particular, rules of the form $b_1 \wedge \cdots \wedge b_n \rightarrow b$ may be unfolded when all premises b_i are explicitly present in the belief base, yielding b as an inferred belief that can be referenced in an explanation. For example, in the cleaning robot scenario, a rule stating that a room is *dirty* if it contains visible debris can be expanded when the belief `debris(Room)` is present, allowing the explanation to refer directly to the inferred belief `dirty(Room)`. More complex forms of rule-based inference are not currently traced. This includes recursive rules, rules involving negation as failure, or rules whose application depends on multiple alternative derivations. This restriction keeps explanation extraction tractable and ensures that generated explanations remain closely aligned with the agent's observable execution, at the cost of not exposing deeper inference chains as explicit explanatory factors.

Limitations and Future Extensions Some features of Jason are only partially addressed and remain topics for future work. These include plans triggered by belief addition or removal events, which require explicit causal links between belief updates and earlier actions to support faithful explanation. Further limitations concern a full semantic treatment of internal actions such as `.send`, the computation (rather than explicit recording) of action postconditions, and the handling of probabilistic or nondeterministic actions. We return to these issues in the discussion section.

7 Implementation and Case Study

In this section, we describe the implementation of the explanation mechanism and illustrate its use through a concrete case study. The focus is on explanation computation, rather than on the engineering aspects of integrating an explainer into the Jason runtime.

The explanation mechanism is implemented as a standalone Prolog *explainer*, which takes as input a representation of an agent's execution and produces explanations according to the rules defined in Section 4.1 and instantiated for Jason in Section 6. For the case study, the translation from Jason executions to the Prolog representation is performed manually. This translation mirrors information obtainable through automatic instrumentation [3] and does not affect the correctness of explanation computation.

7.1 Explainer Engine

The core of the implementation is the explainer engine, implemented in Prolog (full code publicly available[4]). Explanations are computed by querying the engine with a predicate of the form `explain(X,TS,F)`, where `X` denotes a changer, `TS` the execution time at which it occurred, and `F` an optional contrastive alternative. The explainer explicitly renders the corresponding "why" question and returns the set of explanatory factors associated with the execution of `X` at time `TS`.

Internally, the explainer operates over a single conceptual execution trace, as defined in Section 4.1. In the implementation, this trace is represented in a decomposed form by a set of time-indexed relations capturing executed changers, instantiated plan instances, changers preconditions and postconditions, and belief states.

Execution Trace. Changers executed by the agent are represented as a sequence of time-stamped events:

```
1 tr([(1,next(slot)), (2,pick(garb)), (3,move_towards(1,1)), (4,drop(garb)), ...]).
```

Time stamps allow the explainer to distinguish repeated executions of the same changer and to associate explanatory factors with the correct deliberation or execution point. Note that, the predicate `tr/1` represents the sequence of executed changers (this corresponds to $tr(ag,t)_{|Ch}$ as denoted in Section 4.1). Additional execution information, such as belief states and plan instances, is indexed by the same time stamps and jointly constitutes the execution trace used by the explainer.

[4] https://github.com/AngeloFerrando/BDIExplainer

Plan Instances. Plans are represented as *plan instances*, rather than abstract plan schemas (e.g., as $p(t, label, trigger, context, body)$), where within $body$ actions and sub-goals have a time stamp. Plan instances are not maintained as a separate execution trace, but as time-indexed components of the same execution trace, stored in a separate relation for modularity (this corresponds to $tr(ag, t)_{|Pl}$ component used in Section 4.1). Each instance is associated with the time at which it was selected:

```
p(TS, checkslot_rec, goal(check(slots)), not(garbage(r1)),
  [(TS, act(next(slot))), (TS1, goal(check(slots)))]).
```

This representation supports recursion and ensures that explanations are grounded in the actual execution structure of the agent, rather than in static plan definitions.

Jason selects plans using sequential OR semantics, considering applicable plans in the order in which they appear. This is captured explicitly by assigning an order to plans associated with the same triggering event:

```
plan_num(checkslot_rec, 1).
plan_num(checkslot_base, 2).
```

The explainer exploits this ordering to generate contrastive explanations, identifying earlier plans in the sequence whose context conditions did not hold at the time of selection.

Changer Preconditions and Postconditions. Changer preconditions and postconditions are represented explicitly when available. As with plan instances, this information is not maintained as a separate execution trace, but as a time-indexed component of the same conceptual execution trace, stored in a dedicated relation for modularity (this corresponds to pre and $post$ functions used in Section 4.1):

```
prepost(TS, act(next(slot)), [], []).
```

Belief states. These are likewise represented as time-indexed facts and are queried by the explainer to evaluate plan context conditions, negation-as-failure, and belief-based explanatory factors.

7.2 Case Study: Cleaning Robot

To illustrate the approach, we consider a cleaning robot adapted from the Jason distribution (available in the public repository). The robot repeatedly scans a set of locations, picks up garbage when present, and carries it to a disposal point. The relevant behaviour is encoded by a small set of recursive plans: one plan iterates over slots when no garbage is present, while an alternative plan is selected when garbage is detected, triggering a nested sequence of subgoals that ultimately lead to the execution of `pick(garb)`.

The agent's behaviour involves: (i) recursive goal achievement, (ii) conditional plan selection based on percepts, (iii) repeated execution of the same actions, and (iv) negated context conditions. This makes it a suitable test case for explanation generation, since these features are part of the contribution of this paper over prior work.

We consider the example query: *Why did the agent perform* `pick(garb)` *at time 2?* This corresponds to querying the explainer with `explain(act(pick(garb)), 2, null)`. The explainer explicitly renders the "why" question and returns a set of time-stamped explanatory factors derived from the execution trace and the instantiated plan structure.

An excerpt of the output produced by the explainer is shown to the right. The explanation can be read as follows. The action `pick(garb)` was executed because the agent was engaged in a recursive slot-checking goal (`checkslot_rec`), which at time 2 selected a plan whose context condition `garbage(r1)` held. This triggered a cascade of subgoals (e.g., `take`, `ensure_pick_rec`, `carry_to`), each of which contributes a desire-based explanatory factor. At the same time, the explainer reports that an alternative plan, applicable when `not(garbage(r1))` holds, was not selected at time 2, yielding a contrastive factor. Overall, the explanation reflects both the hierarchical goal structure of the agent and the temporal evolution of context conditions that drove plan selection in this execution.

Why act(pick(garb))@2 ?
- ccond garbage(r1)@2
- ccond **not**(garbage(r1))@1
- desire carry_to@2
- desire checkslot_base@2
- desire checkslot_rec@1
- desire ensure_pick_rec@2
- desire take@2
- ncond **not**(garbage(r1))@2

8 Conclusions and Future Work

This paper presented *AgentSpeaX*, a conceptual framework for execution-grounded explainability in BDI agent systems. By grounding explanations in instantiated AgentSpeak(L) plan executions, AgentSpeaX ensures that explanations faithfully reflect an agent's deliberation and execution, even in the presence of recursion and repeated plan execution. Dealing with recursion and considering real AgentSpeak(L) plans rather than abstract and-or trees is a new, powerful feature of our AgentSpeaX w.r.t. Winikoff *et al.* [41]. The framework also treats explanations as first-class communicative artefacts, enabling explanation exchange through dedicated performatives.

We instantiated AgentSpeaX for the Jason agent programming language and provided a Prolog-based explainer as an executable specification of the explanation rules. The contribution is intentionally conceptual: the explainer demonstrates how execution-grounded explanations can be computed, while instrumentation and trace extraction are treated as orthogonal engineering concerns.

Future work includes relaxing the assumptions made for realizing a first working prototype of the explainer, and integrating the explainer *ChatBDI* [12,13], where AgentSpeaX can provide structured execution-grounded content for natural-language explanations. There is also scope for further evaluation of efficiency and effectiveness. Overall, AgentSpeaX lays a foundation for BDI agents that are both autonomous and accountable.

Acknowledgments. Michael Winikoff would like to thank the University of Ljubljana for hosting him on sabbatical while this paper was written.

References

1. Abdulrahman, A., Richards, D., Bilgin, A.A.: Exploring the influence of a user-specific explainable virtual advisor on health behaviour change intentions. Auton. Agents Multi Agent Syst. **36**(1), 25 (2022). https://doi.org/10.1007/s10458-022-09553-x
2. Amitai, Y., Septon, Y., Amir, O.: Explaining Reinforcement Learning Agents through Counterfactual Action Outcomes. In: AAAI. pp. 10003–10011. AAAI Press (2024). https://doi.org/10.1609/AAAI.V38I9.28863
3. Ancona, D., Daoui, Z., Ferrando, A., Gatti, A., Winikoff, M., Mascardi, V.: The secret life of traces: a MAS engineering perspective. In: EMAS (2026)
4. Bordini, R., Hübner, J., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason, vol. 8. John Wiley & Sons, Ltd, United Kingdom (10 2007). https://doi.org/10.1002/9780470061848
5. Bratman, M.: Intention, Plans, and Practical Reason. Cambridge, MA: Harvard University Press, Cambridge (1987)
6. Broekens, J., Harbers, M., Hindriks, K.V., van den Bosch, K., Jonker, C.M., Meyer, J.C.: Do you get it? user-evaluated explainable BDI agents. In: Dix, J., Witteveen, C. (eds.) Multiagent System Technologies, 8th German Conference, MATES 2010, Leipzig, Germany, September 27-29, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6251, pp. 28–39. Springer (2010). https://doi.org/10.1007/978-3-642-16178-0_5
7. Buçinca, Z., Swaroop, S., Paluch, A.E., Doshi-Velez, F., Gajos, K.Z.: Contrastive explanations that anticipate human misconceptions can improve human decision-making skills. In: Yamashita, N., Evers, V., Yatani, K., Ding, S.X., Lee, B., Chetty, M., Dugas, P.O.T. (eds.) Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, CHI 2025, YokohamaJapan, 26 April 2025- 1 May 2025. pp. 1024:1–1024:25. ACM (2025). https://doi.org/10.1145/3706598.3713229
8. Ciatto, G., Aguzzi, G., Battistini, R., Baiardi, M., Burattini, S., Ricci, A.: Exploiting GenAI for plan generation in BDI agents. In: ECAI (10 2025). https://doi.org/10.3233/FAIA251223
9. Cohen, P.R., Perrault, C.R.: Elements of a plan-based theory of speech acts. Cognitive Science **3**(3), 177–212 (1979). https://doi.org/10.1016/S0364-0213(79)80006-3
10. Cranefield, S., Oren, N., Vasconcelos, W.W.: Accountability for practical reasoning agents. In: Lujak, M. (ed.) Agreement Technologies (AT). LNCS, vol. 11327, pp. 33–48. Springer (2018). https://doi.org/10.1007/978-3-030-17294-7_3
11. Dennis, L.A., Oren, N.: Explaining BDI agent behaviour through dialogue. Auton. Agents Multi Agent Syst. **36**(1), 29 (2022). https://doi.org/10.1007/S10458-022-09556-8
12. Gatti, A., Mascardi, V., Ferrando, A.: ChatBDI: Think BDI, Talk LLM. In: Das, S., Nowé, A., Vorobeychik, Y. (eds.) Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Detroit, MI, USA, May 19-23, 2025. pp. 2541–2543. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2025). https://dl.acm.org/doi/10.5555/3709347.3743930
13. Gatti, A., Mascardi, V., Ferrando, A.: Let Me Talk to You! Natural Language Interaction Between Humans and BDI Agents via ChatBDI. In: ECAI 2025, Frontiers in Artificial Intelligence and Applications, vol. 413, pp. 3646–3654. IOS Press (2026). https://doi.org/10.3233/FAIA251242
14. Gyevnar, B., Droop, S., Quillien, T., Cohen, S.B., Bramley, N.R., Lucas, C.G., Albrecht, S.V.: People Attribute Purpose to Autonomous Vehicles When Explaining Their Behavior: Insights from Cognitive Science for Explainable AI. In: Conference on Human Factors in Computing (CHI). pp. 86:1–86:18. ACM (2025). https://doi.org/10.1145/3706598.3713509

15. Harbers, M., van den Bosch, K., Meyer, J.C.: Design and evaluation of explainable BDI agents. In: Huang, J.X., Ghorbani, A.A., Hacid, M., Yamaguchi, T. (eds.) Proceedings of the 2010 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2010, Toronto, Canada, August 31 - September 3, 2010. pp. 125–132. IEEE Computer Society Press (2010). https://doi.org/10.1109/WI-IAT.2010.115
16. High-Level Expert Group on Artificial Intelligence: The assessment list for trustworthy artificial intelligence. https://digital-strategy.ec.europa.eu/en/library/assessment-list-trustworthy-artificial-intelligence-altai-self-assessment (2020)
17. Ichida, A.Y., Meneguzzi, F., Bordini, R.H.: BDI agents in natural language environments. In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 880–889. IFAAMAS (2024)
18. IEEE: IEEE Standard for transparency of autonomous systems. IEEE Std 7001-2021 (2022). https://doi.org/10.1109/IEEESTD.2022.9726144
19. Kaptein, F., Broekens, J., Hindriks, K.V., Neerincx, M.A.: Personalised self-explanation by robots: The role of goals versus beliefs in robot-action explanation for children and adults. In: 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN). pp. 676–682. IEEE (2017). https://doi.org/10.1109/ROMAN.2017.8172376
20. Kaptein, F., Broekens, J., Hindriks, K.V., Neerincx, M.A.: Evaluating Cognitive and Affective Intelligent Agent Explanations in a Long-Term Health-Support Application for Children with Type 1 Diabetes. In: 8th International Conference on Affective Computing and Intelligent Interaction (ACII). pp. 1–7. IEEE (2019). https://doi.org/10.1109/ACII.2019.8925526
21. Labrou, Y., Finin, T.: A semantics approach for KQML – a general purpose communication language for software agents. In: Proceedings of the Third International Conference on Information and Knowledge Management. p. 447–455. CIKM '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/191246.191320
22. Labrou, Y., Finin, T.: A proposal for a new KQML specification. Tech. rep., TR CS-97-03 from UMBC (1997)
23. Langley, P., Meadows, B., Sridharan, M., Choi, D.: Explainable agency for intelligent autonomous systems. In: Singh, S., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 4762–4764. AAAI Press (2017). https://doi.org/10.1609/AAAI.V31I1.19108
24. Malle, B.F.: How the Mind Explains Behavior. MIT Press (2004), ISBN: 9780262134453
25. Mauri, M., Minor, M.: Towards Explainable BDI Agents for End-Users. In: Engineering Multi-Agent Systems (EMAS 2025). Lecture Notes in Computer Science, Springer (2025)
26. Melo, V.S., Panisson, A.R., Bordini, R.H.: Towards Generating P-Contrastive Explanations for Goal Selection in Extended-BDI Agents. In: PRIMA 2023: Principles and Practice of Multi-Agent Systems. Lecture Notes in Computer Science, vol. 14368, pp. 355–371. Springer (2023). https://doi.org/10.1007/978-3-031-45368-7_23
27. Miller, T.: Explanation in artificial intelligence: Insights from the social sciences. Artif. Intell. **267**, 1–38 (2019). https://doi.org/10.1016/J.ARTINT.2018.07.007
28. Omicini, A.: Not Just for Humans: Explanation for Agent-to-Agent Communication. In: Vizzari, G., Palmonari, M., Orlandini, A. (eds.) Proceedings of the AIxIA 2020 Discussion Papers Workshop co-located with the the 19th International Conference of the Italian Association for Artificial Intelligence (AIxIA2020), Anywhere, November 27th, 2020. CEUR Workshop Proceedings, vol. 2776, pp. 1–11. CEUR-WS.org (2020). https://ceur-ws.org/Vol-2776/paper-1.pdf
29. Panisson, A.R., Engelmann, D.C., Bordini, R.H.: Engineering explainable agents: An argumentation-based approach. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems - 9th International Workshop, EMAS 2021, Virtual Event, May

3-4, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13190, pp. 273–291. Springer (2021). https://doi.org/10.1007/978-3-030-97457-2_16
30. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer (1996). https://doi.org/10.1007/BFb0031845
31. Robinette, P., Li, W., Allen, R., Howard, A.M., Wagner, A.R.: Overtrust of robots in emergency evacuation scenarios. In: Human Robot Interaction (HRI). pp. 101–108. IEEE/ACM (2016). https://doi.org/10.1109/HRI.2016.7451740
32. Rodriguez, S., Thangarajah, J.: Explainable Agents (XAg) by Design. In: Dastani, M., Sichman, J.S., Alechina, N., Dignum, V. (eds.) Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024, Auckland, New Zealand, May 6-10, 2024. pp. 2712–2716. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2024). https://dl.acm.org/doi/10.5555/3635637.3663263
33. Rodriguez, S., Thangarajah, J., Davey, A.: Design patterns for explainable agents (XAg). In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1621–1629. IFAAMAS (2024)
34. Verhagen, R.S., Neerincx, M.A., Tielman, M.L.: A two-dimensional explanation framework to classify AI as incomprehensible, interpretable, or understandable. In: Explainable and Transparent AI and Multi-Agent Systems (EXTRAAMAS). LNCS, vol. 12688, pp. 119–138. Springer (2021). https://doi.org/10.1007/978-3-030-82017-6_8
35. Vieira, R., Moreira, Á.F., Wooldridge, M.J., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. J. Artif. Intell. Res. **29**, 221–267 (2007). https://doi.org/10.1613/jair.2221
36. Winfield, A.F.T., Booth, S., Dennis, L.A., Egawa, T., Hastie, H.F., Jacobs, N., Muttram, R.I., Olszewska, J.I., Rajabiyazdi, F., Theodorou, A., Underwood, M.A., Wortham, R.H., Watson, E.N.: IEEE P7001: A proposed standard on transparency. Frontiers Robotics AI **8**, 665729 (2021). https://doi.org/10.3389/frobt.2021.665729
37. Winikoff, M.: Towards trusting autonomous systems. In: Engineering Multi-Agent Systems (EMAS). LNCS, vol. 10738, pp. 3–20. Springer (2017). https://doi.org/10.1007/978-3-319-91899-0_1
38. Winikoff, M.: Towards engineering explainable autonomous systems. In: Briola, D., Cardoso, R.C., Logan, B. (eds.) Engineering Multi-Agent Systems - 12th International Workshop, EMAS 2024, Auckland, New Zealand, May 6-7, 2024, Revised Selected Papers. Lecture Notes in Computer Science, vol. 15152, pp. 144–155. Springer (2024). https://doi.org/10.1007/978-3-031-71152-7_9
39. Winikoff, M.: Contrastive explanations of BDI agents. In: Amato, C., Dennis, L., Mascardi, V., Thangarajah, J. (eds.) Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Paphos, Cyprus, May 25-29, 2026. International Foundation for Autonomous Agents and Multiagent Systems / ACM (2026)
40. Winikoff, M., Sidorenko, G.: Evaluating a Mechanism for Explaining BDI Agent Behaviour. In: Explainable and Transparent AI and Multi-Agent Systems (EXTRAAMAS), Revised Selected Papers. LNCS, vol. 14127, pp. 18–37. Springer (2023). https://doi.org/10.1007/978-3-031-40878-6_2
41. Winikoff, M., Sidorenko, G., Dignum, V., Dignum, F.: Why bad coffee? Explaining BDI agent behaviour with valuings. Artif. Intell. **300**, 103554 (2021). https://doi.org/10.1016/J.ARTINT.2021.103554
42. Yan, E., Burattini, S., Hübner, J.F., Ricci, A.: A multi-level explainability framework for engineering and understanding BDI agents. Auton. Agents Multi Agent Syst. **39**(1), 9 (2025). https://doi.org/10.1007/S10458-025-09689-6