Autonomous Agents and Multi-Agent Systems (2025) 39:9
https://doi.org/10.1007/s10458-025-09689-6

A multi-level explainability framework for engineering and understanding BDI agents

Elena Yan[1] · Samuele Burattini[2] · Jomi Fred Hübner[3] · Alessandro Ricci[2]

Accepted: 16 January 2025

Abstract
As the complexity of software systems rises, explainability - i.e. the ability of systems to provide explanations of their behaviour - becomes a crucial property. This is true for any AI-based systems, including autonomous systems that exhibit decisionmaking capabilities such as multi-agent systems. Although explainabil- ity is generally considered useful to increase the level of trust for end-users, we argue it is also an interesting property for software engineers, developers, and designers to debug and validate the system's behaviour. In this paper, we propose a multi-level explainability framework for BDI agents to generate explanations of a running system from logs at different levels of abstraction, tailored to different users and their needs. We describe the mapping from logs to explanations, and present a prototype tool based on the JaCaMo platform which implements the framework.

Keywords BDI agents · Explainable AI · Debugging

1 Introduction

Explainability emerged in recent years as an important desired property for artificial intelligence (AI) based systems [1]. While the term is more prominently employed to describe the process of extracting human-understandable explanations from Machine Learning models to improve the users' level of trust or allow developers to gain insights into the learned param-

✉ Elena Yan
elena.yan@emse.fr

✉ Samuele Burattini
samuele.burattini@unibo.it

✉ Jomi Fred Hübner
jomi.hubner@ufsc.br

✉ Alessandro Ricci
a.ricci@unibo.it

[1] Mines Saint-Etienne, Univ Clermont Auvergne, INP Clermont Auvergne, CNRS, UMR 6158 LIMOS, F-42023 Saint-Etienne, France

[2] Department of Computer Science and Engineering, Alma Mater Studiorum, University of Bologna - Cesena Campus, Via dell'Università 50, 47521 Cesena, FC, Italy

[3] Department of Automation and Systems Engineering, Federal University of Santa Catarina, PO Box 476, Florianópolis, SC 88040-900, Brazil

Published online: 30 January 2025

eters [2], the field of AI is much broader and includes many other kinds of systems that may benefit from the ability to generate such explanations.

Among these AI-based systems, agent-based systems play a prominent role [3]. In this context, explainability mainly concerns the capability for human users to get clear explanations about the decisions and actions autonomously taken and performed by agents. Such a capability can be considered important not only from an end-user perspective, but also from an engineering one. For instance, developers can exploit explanations to save them time in spotting and understanding problems during agent debugging and fine-tuning.

When adopting an engineering perspective, the agent model and architecture adopted can have a strong impact on explainability. In particular, high-level cognitive models and architectures such as Belief-Desire-Intention (BDI) [4] support explainability *by design* [5, 6], in principle. This has been clearly recognized in literature by different approaches in the last two decades, including works about exploiting explainability for debugging BDI agents [7, 8], and validating BDI agent's behaviour [9]. Existing works based on BDI agent programming languages and platforms typically exploit *traces* generated by the agent at runtime, and build explanations based on the specific operational semantics on which the specific agent programming language is based. This effectively limits the expressiveness of explanations, as the level of abstraction is strictly tied to the adopted agent programming language. Although the explanations can be fairly understandable for developers who are used to work with the agent programming language, they are much harder for end-users to understand them.

In this paper, we recognise that explanations of agent-based systems can be used by different kinds of users and for different purposes, from developers working at the implementation level—i.e., using a specific agent programming language—to designers and engineers who may want to abstract from details of the specific technology and focus more on the architectural level, to finally the domain experts and end-users, who focus on the functional and non-functional requirements of the system as a whole, viewing it as a "black-box".

Following this perspective, in this paper we introduce a *multi-level explainability framework*, which allows for defining explanations at different levels of abstractions, corresponding to different kinds of users. In particular, we identified three main hierarchical levels: an *implementation level* at the bottom, targeting developers; a *design level* in the middle, targeting designers; and a *domain level* at the top, targeting end-users. BDI is taken as main model/architecture reference at the design level, and Jason [10] as programming language at the implementation level—even if the basic principles behind the general idea are independent of the specific agent model/architecture/language adopted.

The framework is based on a process that starts from the execution traces of Jason agents, captured at runtime into logs, generating then *narratives* at the proper level of abstraction from the logs, and finally *explanations* given users' questions about the narratives.

The approach is modular: the generation of a narrative at some upper level is based on the information coming from the lower level, but both abstracting from details that are not relevant at the upper level and enriching them with specific semantic knowledge that characterises such upper level.

In order to evaluate and apply the framework in practice, we developed a first prototype of an *explainer*. The tool can produce explanations at two different levels of abstraction: at the implementation level, targeting developers using concepts that are familiar to Jason programmers, and at the design level, targeting designers that use abstract cognitive concepts that are familiar to any BDI agent expert. The third and final level of explanation, the domain level, is not achieved in this prototype, as it requires further alignment with other methodological

processes to collect and further inject domain knowledge into the explainer component. We discuss this limitation in Sect. 6 and leave further exploration for future works.

The remainder of the paper is organised as follows. First, in Sect. 2 we discuss the related work on explainability for autonomous agent systems. In Sect. 3 we introduce the general idea of generating explanations at multiple levels of abstraction with a running example that will be used in the whole paper. In Sect. 4 we formalise and describe the general process required to build the "multi-level" explainer, taking Jason programming language and the JaCaMo platform as the technology for this prototype. In Sect. 5 we describe the actual prototype implementation and show usage examples. In Sect. 6 we discuss current limitations and perspectives for future works. Finally, in Sect. 7 we summarise the outcomes of our contribution.

2 Related work

In existing research on explainability, there is significant interest in AI systems [1], commonly referred to as eXplainable AI (XAI). While much of XAI research concentrates on machine learning systems and data-driven approaches to explain their black-box algorithms, there is also a substantial body of work dedicated to goal-driven XAI. This area particularly concerns autonomous agents, which aims to explain their decision-making processes and underlying reasoning [11, 12]. Existing research demonstrates that providing explainability features to the systems can be useful for several purposes both from a software engineering perspective and a user perspective. Our focus is specifically on BDI agents [4, 13], where explainability can be gained using folk psychology constructs [14].

In our analysis of the related works in this field, we find that explainability features have been used for different purposes and in different phases of the engineering process of an agent-based system.

In the context of debugging, several approaches use a logging mechanism to record the agent-based system's execution and then elaborate to provide explanations to increase its understanding. The idea is supported by omniscient debugging [15, 16], which allows developers to navigate through the history both forward and backwards without having to re-execute the program to find the root cause of an error. Given the system's trace, a range of works exploits a set of "Why? Questions" following the Whyline [17] approach for debugging Multi-Agent Systems (MAS). The idea is to allow developers to ask questions about the agent such as "why does it (not) *believe* something?" or "why does it (not) *do* a certain action?" and receive explanations for the choices that led to these events [7–9]. A similar work proposes explaining BDI-based agents through user dialogues [18] to help identify disagreements and highlight errors. Complementary approaches present the agent's concepts through visual methods such as concept graph [19, 20], relational graph [21] between the agent's concepts, or tree of goals and sub-goals [22] supporting the debugging scenarios.

Focusing on the testing and validation of the system, explainability is encoded in the high-level description of the expected system's behaviour under the assumption that such descriptions should improve the level of explanation of the non-compliant behaviour. Use cases [23] are great for representing the system's behaviour and scenarios under various conditions as requested by stakeholders. Carrera et al. [24] introduce the BEAST Methodology, an agile testing methodology based on Behaviour Driven Development (BDD), aimed at improving the collaboration between stakeholders and developers and automatically convert behaviour specifications into unit tests to verify system execution with explainable testing

setting. Similar work is presented by Rodriguez et al. [25] who define the description of the system starting from user stories (i.e., stories from the user's perspective) that map to system stories (i.e., stories from the system's perspective) and associate with acceptance criteria. During the development phase, system stories are mapped to agent stories [26] capturing goals, plans, beliefs, and percepts to test whether they meet the requirements through the acceptance criteria [27]. Rodriguez et al. [28] also advocate the need for design patterns to develop explainable by design agents, providing in-built explainability for multi-agent systems based on goal-oriented behaviour. Winikoff [29] augments this concept by stating the need for an explanation from a single component to whole multi-component systems. He proposes an architecture that assigns an explainer agent to each system component requiring explanations, enabling users to ask questions and receive answers from the explainer agents.

Although several research studies have investigated explainable agents for specific purposes, there is a lack of work that adapts the explainability to different classes of users. For instance, experts use the explanations more to resolve anomalies and verify the system's behaviour, focusing on technical details [30], whereas end-users are more interested in understanding the systems' functioning and receiving meaningful information alongside their output [31]. Most research in explainability in MAS focuses on a *single agent* producing a *single explanation* for a *single purpose*. In our research, we aim to integrate the benefits of explainability by providing multiple levels of abstraction tailored to the specific needs of different user levels.

3 A multi-level perspective on explainability

Compared with related works which generally focus on a single level of explanation, our research introduces a different approach by presenting an explainability framework for BDI agents that deals with multiple levels of abstraction. Different levels can be used for different purposes by users playing different roles over the lifecycle of a MAS, from design to programming to final deployment and execution.

3.1 Different roles, different explanations

We start from the idea that explanations in the MAS community have been used either as support for comprehending the behaviour of a running system [19, 21] or for software engineering processes such as collaboration with designers and domain experts [24], debugging [8, 15, 16, 22, 32, 33], as well as testing and validation of agent behaviours [24–27, 34].

All of these use cases require dealing with different levels of abstraction in the generated explanation since they target users playing different roles that are associated with different needs and objectives. Specifically, the roles that we target are:

- **Developers**, for supporting the development of the MAS on a technical level. The explanation needs to be detailed enough to provide a complete view of the inner workings and tailored to the specific technologies that are used to implement the MAS.
- **Software Designers/Architects**, including also designers who need to understand the system's behaviour for effective collaboration with stakeholders, check compliance with requirements and formulate new ones. The level of explanation for this role needs to be more abstract because the technological details are not relevant.

- **End Users**, not familiar with the technological aspects of the system and only interested in what the system is doing. The explanation should be related to system requirements and domain specifics that they can understand.

Identified the different roles requiring an explanation at a specific level of abstraction, we then establish how we can collect information about the running system and present it in a human-friendly way.

3.2 From logs to narratives

A classic, simple and valuable way to collect information about running systems is by using logs to generate traces of the system execution. Logging can be either *explicit* when the developer takes an active part in selecting which information is logged, or *implicit* when the overall execution platform captures and builds traces without explicit intervention—or in many cases allowing simply to select the level of verbosity of the trace. In this case, implicit logging is better suited, as the MAS execution framework can generate uniform traces while executing agent programs and always deliver the same amount of information, without adding any burden to the developer, or unnecessary statements in the agent program itself.

Logs though, tend to remain at a very technical level, as their main functionality is to detail the behaviour of a given system. On the other hand, explainers,—despite having a similar function—use natural language to avoid introducing yet another technical barrier between the explanation and the end users.

In our approach (see Fig. 1), we use the idea of *narrative* to indicate a story or a description of the system's behaviour composed of a sequence of relevant events that are presented as sentences. Such events are extrapolated from the system logs, and translated into natural language so that they can be more easily read by humans.

Following this approach, for each event in the narrative, an *explanation* is a subset of such narrative that is composed only of events that have a *causal link* with the event to explain. We envision that users of the system may read the narrative to understand the system's behaviour, and may question the motivations behind the occurrence of a specific event—e.g. if one questions why their autonomous car took a left turn they would look for the event of turning left in the narrative, then request an explanation for it. The explanation would then contain only previous events that are related to the questioned ones by causal links – e.g. the car may have turned left because of an updated belief about the traffic situation on the main road, as such the narrative would show the event of the new belief being evaluated and the consequent decision-making process that led to the decision of turning. We refrain from defining causality in this paper, with the idea that when defining a level of explanation and identifying the relevant events that could occur at that level it should be possible to also define for each event a causal relationship with another event of that same level. This is further discussed and exemplified in Sect. 4. How trivial or how complex the definition of the causal link determines the level of quality of the produced explanation.

Of course, in the spirit of the multi-level perspective on explainability, users may be able to select a *narrative level* that is better suited for their role and read about the system's behaviour at a specific level of abstraction, as well as receive explanations at that same level.

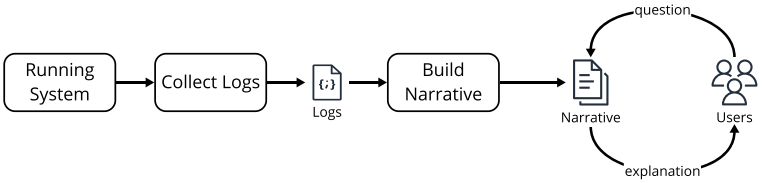


Fig. 1 Process of the multi-level explainability tool

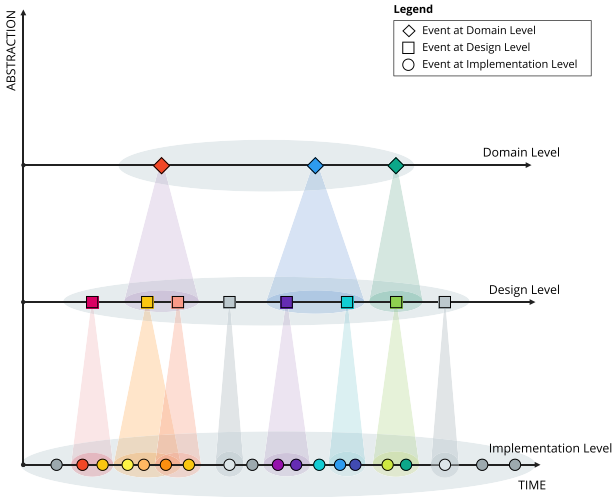


Fig. 2 The idea of multiple levels of abstraction for the explanation of a (computing) system. Explanations in this case are based on relevant events characterising system execution. Explanations and events depend on the level of abstraction considered. The bottom layer involves the technologies used to implement the system (e.g., specific agent-oriented programming languages). The middle layer involves the principles and architectures adopted to design the system (e.g., cognitive architectures). The top level concerns the functionalities as perceived by users and stakeholders of the system

3.3 A multi-level explainability framework

Having identified the different roles users of an explainer may play, and the overall process that allows them to go from logs to narratives to, finally, explanations, we here define the idea of a *multi-level explainability framework*. With this, we intend to conceptualise a generic framework for explainability that can guide MAS developers in supporting the generation of different levels of narratives and explanations, for different user roles.

The level of explanation is more abstract when it deals with less technical users and is hierarchically built on top of the lower level (Fig. 2). We refer to this approach as *multi-level explainability* with different levels of abstraction in the generated narrative. We identify three levels, that correspond to the roles users can play when using the explainer:

- **Implementation Level**. This level is useful for developers in the software engineering phases such as debugging and testing. Thus, we start with a detailed narrative closely related to the MAS at a technical level that facilitates the developers to understand the agent's behaviour. This level includes low-level information concerning the execution engine and details about the implementing technology.

- **Design Level**. This level is useful for architects as well as developers of the system that want to abstract from the low-level implementation details—concerning specific e.g. agent programming languages or technologies—and focus more on the behaviour of the agents at the design level. This level is a good abstraction for designers and architects who want to comprehend the current system's behaviour, interact with other stakeholders and developers or formulate new requirements for improving the system.
- **Domain Level**. This is the top level, abstracting from how the system has been designed and focusing more on what functionalities the system is meant to provide to stakeholders – i.e. functional and non-functional requirements—dealing with domain-specific knowledge and insights. This level is meant to be useful both for end-users, to understand and explain the system's behaviour as a whole while using it, and for software engineers, to validate the system given the specification and requirements defined during the analysis stage of the software engineering process.

These abstraction levels are designed to be flexible, allowing different groups of users to access explanations from different perspectives. Each level of explanation includes all the relevant information necessary for a complete understanding of that specific abstraction. Consequently, levels are considered fully *self-contained*: meaning that all explanations and information necessary for the evaluation of behaviour at a given level are enclosed within it, avoiding considering the details of other levels.

Narratives at higher levels of abstraction are built on top of narratives at lower levels. This modular process involves processing the most relevant events that the system produces at one level and presenting them in a more abstract, higher-level way, at the level of abstraction above. Practically speaking, the implementation level, being strictly tied to the adopted technology, is directly built upon the system logs of the system. The subsequent design level is generated by mapping those implementation events to a more generic set of events, that can abstract from the specific technological details, whereas the domain level uses those abstract events to match the functional requirements of the system and present narratives that can directly refer to the expected system's behaviour.

In the remainder of the paper, we will apply this generic framework to BDI agents, identifying practically how to derive the different levels and implementing a tool that supports the generation of the first two levels, leaving the domain level and consideration on the applicability to other kinds of agents for future works.

3.4 Explaining at different levels of abstractions: an example

Before diving into the practical realisation of the different levels, in this section, we provide an example of how the multi-level explainability framework could look like for a BDI agent. We use the example to further characterise the differences between the levels of abstraction of the narratives in a specific application setting, before explaining how to practically build the different levels in Sect. 4.

We consider the *domestic robot*[1] toy example described in the Jason book [10]. We will use Jason, and the JaCaMo platform as the technological references for this example and the remainder of the paper, although similar considerations should be easily transferred to any other BDI programming platform.

We here introduce the example as we were going to develop it from scratch, first detailing the domain and the high-level requirements of the system, then showing the design of the solu-

[1] The example is available on GitHub at https://github.com/jason-lang/jason/tree/master/examples/domestic-robot.

tion using agents and then briefly describing the implementation. This software engineering process matches the levels of abstraction that we identified for our explanation framework. It follows that the explanations, being tailored to a specific level, will use concepts that come from that same level.

3.4.1 Domain level: narratives for end-users

The narratives for end-users/stakeholders are sequences of *domain events*, i.e. events that are relevant at the domain level. In the software engineering processes, domain events are meant to be identified in the analysis stage, e.g. from use cases [23] or through informal processes typically used in Agile methodologies [35] such as Event Storming or User Stories [36].

Use cases allow for refining this knowledge up to capturing the contract between the stakeholders of a system about its behaviour, describing the system's behaviour under various conditions as the system responds to a request from one of the stakeholders (the primary actor). For instance, given the informal description of the domestic robot example reported in [10]:

> "A domestic robot has the goal of serving beer to its owner. Its mission is quite simple, it just receives some beer requests from the owner, goes to the fridge, takes out a bottle of beer, and brings it back to the owner. However, the robot should also be concerned with the beer stock (and eventually order more beer using the supermarket's home delivery service) and some rules hard-wired into the robot by the Department of Health (in this example this rule defines the limit of daily beer consumption)."

Given this informal description, we may define a `Get a Beer` use case, following the schema suggested in [23]:

```
Name: Get a Beer

Context of Use: Owner wants to get a beer
Scope: the robot (system under design)
Level: user-goal
Primary Actor: Owner
Trigger: The owner asks the robot for a beer to be delivered from the
    fridge

Main Success Scenario:
1. The owner ask the robot for a beer
2. The robot validates the daily limit
3. The robot verifies the availability of at least one beer in the
    fridge (by interacting with the fridge)
4. The robot goes to the fridge and picks the beer
5. The robot delivers the beer to the owner
6. The owner thanks the robot

Extensions:
2a. The robot does not validate the daily limit
2b. The robot informs the owner that the beer cannot be delivered

3a. The robot verifies that no beer are available
3b. The robot informs the owner that no beers are available and ask
    for requesting for a new stock to the supermarket
3c. Owner confirms
3d. The robot sends the request for new stock to the supermarket
```

Use cases capture exactly the narrative that we expect from the system (behaving correctly with respect to the contract). Starting a use case, it is straightforward to identify the domain events to be tracked, forming the narratives: they correspond with the interactions listed in scenarios. Examples of events in the case of the `Get a beer` are:

```
The owner asked the robot for a beer
The robot received a request to bring a beer from the owner
The robot validated the daily limit
The robot verified that the owner's request could not be satisfied,
    given the daily limit
The robot delivered the beer to the owner
The robot refused to deliver the beer with a message: "..."
```

The narrative that we obtain at runtime at the domain level is a sequence of these events, that should correspond to a scenario as described in use cases. As an example, considering the scenario in which the robot refuses to deliver the beer, the behaviour of the system will generate a narrative like the following:

```
1. The robot received a request to bring a beer from the owner
2. The robot verified that the owner's request could not be
    satisfied, given the daily limit
3. The robot refused to deliver the beer with a message: "..."
```

Explanations are sequence of domain events that are related by causal links that are considered relevant to explain the system behaviour, as specified at the domain level (by domain experts). So for instance, the event:

```
The robot did not deliver the beer to the owner
```

can be considered caused by (explained by) the event:

```
The robot verified that the owner's request could not be satisfied,
    given
the daily limit
```

Examples of questions at this level of abstraction concern the motivation that leads to specific domain events, e.g.:

(Q1) **Why the robot refused to deliver the beer?**

Answers can be given exploiting the explanation link defined among domain events. In the example the answer is given by:

```
The robot verified that the owner's request could not be satisfied,
    given
the daily limit
```

It is worth remarking that, at this level, no detail is given about the design or implementation of the system, that is: there is an abstraction barrier here, so every narrative and explanation should refer solely to the vocabulary used to define functional and non functional requirements, from the end-user and domain expert perspective.

3.4.2 Design level: narratives for designers

After the definition of requirements with the stakeholders and domain experts, the development team can plan the architecture of the system, for instance using an agent-oriented

methodology. At this level, software architects and designers are interested in representing, understanding and validating the behaviour of the system abstracting from programming details and from specific technology that will be used to implement it, and featuring an effective level of abstraction to deal with complexity.

In AI literature this problem is the core of the Knowledge Level conceptual framework proposed by Newell [37, 38], strongly related to the Intentional Stance as introduced by Dennett [39]. Following Newell [37], at the Knowledge Level a computing system is modelled as an agent having some knowledge and some goals, and believing it will do whatever is within its power to attain its goals, in so far as its knowledge indicates (principle of rationality). The conceptual framework has been further extended by Jennings [40] for agent-oriented software engineering to consider a social level modelled using agent organisations. The Knowledge Level characterisation provides a way of stating something about the desired behaviour of the system and about what it must incorporate (namely the requisite knowledge and goals), abstracting from any details about actual internal processing and implementation [41]. This is exactly the baseline that we want for defining narratives for designers.

Along this line, the BDI model/architecture [13] allows to further refine the Knowledge Level principle of rationality with concepts that have their roots in practical reasoning [4]. In BDI, beliefs represent the *informative* component of the system state, i.e. the information about the state of the environment, appropriately updated after each sense. Desires represent the *motivational* state of the system, i.e. the information about the objectives to be accomplished by the system. The notion of goal—which is used more frequently than desires in AOSE and AOP—typically refers to those desires that are not in conflict and that the agent is committed to. Intentions capture the *deliberate* component of the system, representing the currently chosen course of actions, being the result of the decision about what specific goal to accomplish and how.

In this paper then we adopt BDI (and the Knowledge Level) as a reference conceptual framework to define narratives at the Design Level. The concrete development of such narratives can be made straightforward by adopting existing agent-oriented methodologies, based on such a level of abstraction. Main examples are Prometheus [42], Tropos [43], and TDF [44].

For instance, Fig. 3 shows the design of the domestic robot system extrapolated from the Jason book, using the Prometheus notation [42].

In that design, the robot is modelled as an agent that can request to pursue the goal `has(owner,beer)` sent by the owner agent. The agent has a plan referred to as `bring_a_beer` to achieve this goal that can be applied when the daily limit for beers is not overcome. Otherwise, the goal cannot be pursued and the agent must use a plan referred to as `too_much_beer` to handle the situation, properly informing the owner.

A simple example of narrative as a sequence of events occurring to the robot agent in the case that the beer can be delivered follows:

```
1. New goal has(owner, beer) adopted from the agent owner
2. New intention int-1 created for goal has(owner, beer) using plan
    "bring_a_beer".
...
7. Action hand_in(beer) executed following intention int-1
8. Intention int-1 accomplished for goal has(owner, beer) using plan
    "bring_a_beer"
9. Goal has(owner, beer) achieved through intention int-1
```

In the case that the goal cannot be accomplished:

```
1. New goal has(owner, beer) adopted from the agent owner
```

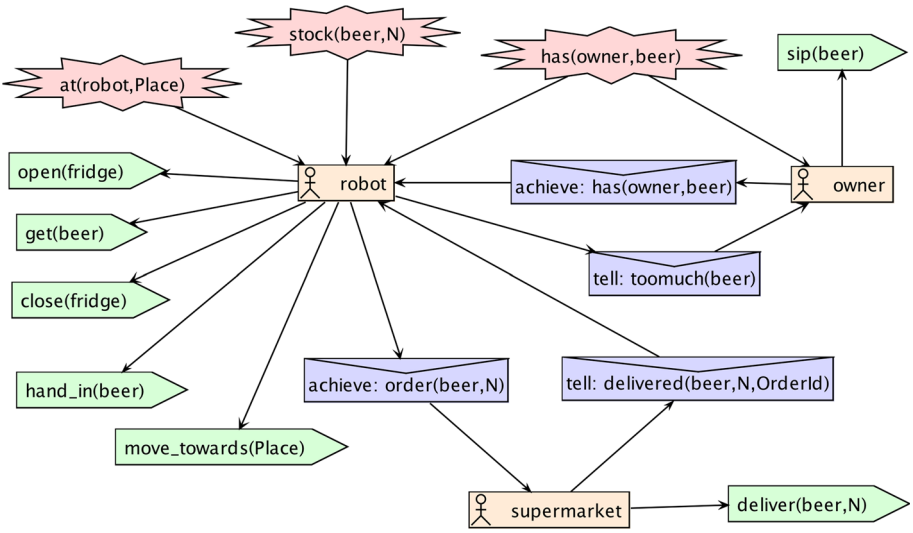


Fig. 3 The design of the multi-agent system to implement the domestic robot example described with the Prometheus notation, from [10]. In the diagram, we can identify the agents, their possible perceptions and actions, and exchanged messages. The agent can perceive: `at(robot,Place)`—the robot place, `fridge` (when the robot is in front of the fridge) or `owner` (when the robot is next to the owner); `stock(beer,N)`—how many beers are stored in the fridge; and `has(owner,beer)`—the owner has a bottle of beer. Actions that can be performed by the agent include `open(fridge)`, `get(beer)`. Regarding the messages (denoted by "envelopes"), the owner sends a message to the robot when he wants another beer. A similar message is sent to the supermarket agent when the robot wants to order more beers. The supermarket informs the robot with a "tell" message when it has delivered the order

```
2. Goal has(owner, beer) cannot be accomplished because there is no
    applicable plan
3. Intention int-62 has to manage a failure in goal has(owner,beer)
    using the plan "too_much_beer"
4. Executed action tell the owner too_much(beer)
...
```

The narratives include relevant events based on a subset of the BDI vocabulary (e.g., goals, plans) used to describe the agent's behaviour, abstracting from implementation/technical detail. In the next section, a more comprehensive account of the possible events will be given.

At this level, architects and designers can use the provided narrative as support for answering for example the following question:

(Q2) **Why did the agent do that action? Why did the agent have that goal?** Architects can gain insights into why certain decisions were taken by the agent and find answers in terms of the mental state of the agent.

Examining the example above, we can track the exchange of messages between the owner and the robot, as well as the choices that lead to performing given actions. An example of a specific question could be:

Why did the agent tell the owner too_much(beer)?

The event:

```
Executed action tell the owner too_much(beer)
```

would be explained with:

```
Intention int-62 has to manage a failure in goal has(owner,beer)
    using the plan "too_much_beer"
```

That is: the robot correctly received the request from the owner, but due to its current beliefs (that encode domain requirements) completes the request by refusing to deliver beer to the owner and informing him accordingly. The explanation for this action, is given by the previous event, clarifying that there is an intention for managing the failure of the goal `has(owner, beer)`. The behaviour is then compliant not only with the expected domain requirements but also with the architectural schema describing agent-to-agent interactions.

3.4.3 Implementation level: narratives for developers

When it comes to the implementation of a problem, the development team chooses specific technologies to effectively build the system and program its behaviour following the schemas that have been defined in the design phase. Programmers are then interested in being able to test and debug their code, they are familiar with the details of the technology and can understand technical language.

In our example the behaviour of the robot agent has two main paths: either taking beer from the fridge and delivering or refusing due to the limit. The implementation of these two cases, in Jason language, is shown in Listing 1. We use comments to explain the code. The goal `has(owner,beer)` is created as a consequence of receiving the achievement message from the owner.

```
available(beer,fridge). // the fridge is initially loaded
limit(beer,10). // the limit is set to 10

// rule that evaluates if the amount of beers consumed is too much
too_much(B) :-
   .date(YY,MM,DD) &
   .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
   limit(B,Limit) &
   QtdB > Limit.

// plan for the successful delivery of beers
+!has(owner,beer)
   :  available(beer,fridge) & not too_much(beer)
   <- !at(robot,fridge);
      open(fridge);
      get(beer);
      close(fridge);
      !at(robot,owner);
      hand_in(beer);
      ?has(owner,beer);
      // remember that another beer has been consumed
      .date(YY,MM,DD);
      .time(HH,NN,SS);
      +consumed(YY,MM,DD,HH,NN,SS,beer).

// plan for refusing
-!has(owner,beer)
   :  too_much(beer) & limit(beer,L)
```

```
  <- .concat("The Department of Health does not allow me to give you
      more than ", L, " beers a day! I am very sorry about that!",M);
     .send(owner,tell,msg(M)).
```

Listing 1 Part of the Jason implementation of the robot agent in the domestic robot example.

As can be seen from the code, there are many technical details that are not relevant at the upper levels, but that may be important to consider when debugging the program. The narrative is then presented in a very close way to the raw log produced with the system execution. It is a detailed, and technical level that follows the operational semantics of the programming language used to code the agents.

Again, an example of the same narrative of the robot not delivering beer to the owner at this level might look like:

```
1. New reasoning cycle started: 11
2. New message from owner: has(owner,beer)
3. Goal has(owner,beer) created
4. Plan options for has(owner,beer) are:
    has(owner,beer) : (too_much(beer) & limit(beer,L))
        <-  .concat("The Department of Health does not allow me to
            give you more than ",L," beers a day! I am very sorry
            about that!",M);
           .send(owner,tell,msg(M)).
5. Plan has(owner,beer) selected
6. Intention 59 has(owner,beer) created, state: undefined
7. Internal action .concat(...) finished
8. New reasoning cycle started: 11
9. Send tell message to owner: msg("The Department of Health does not
    allow me to give you more than 10 beers a day! I am very sorry
    about that!")
10. Internal action .send(owner,tell,msg(M)) finished
11. Intention 59 has(owner,beer) removed, state: undefined
12. Goal has(owner,beer) removed because the goal is achieved
```

The narrative at this level describes the fine-grained agent's behaviour and can be used to explain and answer these developer questions:

(Q3) **Why did a plan fail or get an error?** When a fails or gets an error, developers may be interested in understanding the reason and what is the course of action that led to this failure. The narrative provides details on the motivations and decisions of the system, supporting the process of finding the root cause of the failure.

(Q4) **Why did the agent execute an action?** In the debugging or testing scenario, a program does not necessarily have to fail or encounter errors. It is possible that the system continues its execution, without any interruption, but exhibits a behaviour that is not as expected. Even in these cases, the narrative should support the developer in finding the event that led to that unexpected behaviour.

In this situation, the robot behaved as expected, so no debugging is needed, but the narrative is still useful to verify that all the steps in the plan selection and execution are followed as expected in the test situation. We will show more complex examples in Sect. 5.3.

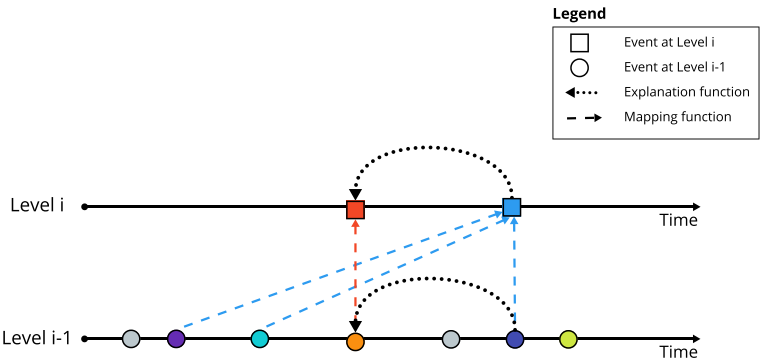


Fig. 4 Example of how events are connected by the explanation function at the same level and by the mapping function between two levels

4 Building narratives at multiple levels of abstraction

In this section, we detail the approach we follow to build narratives at multiple levels following the idea presented in Sect. 3. We adopt a modular process to build narratives at multiple levels above the system logs. At each level of abstraction L^i, we identify formally:

1. the *set of events* that encapsulate the system's behaviour based on the abstraction of the level,
2. the *explanation function* ρ^i that defines formal relations among events for explaining the causal relationship and dependencies of the events, and
3. the *mapping function* b^i, which is defined for the design and domain levels. These functions establish formal relations between events from a lower level to a higher level, building higher-level narratives.

To give an illustrative example, Fig. 4 depicts the process of building narratives and relationships of one upper level and the level below. Each level contains instances of events that belong to its event set. In the figure, events with the same shape are at the same level, while different colours represent different types of events. Certain events are explained by (i.e., caused by) other events at the same level. Each event at the higher level is associated with a mapping function which indicates from which pattern of events is derived from the level below. Notably, events in the mapped sequences: (i) may not necessarily occur sequentially, allowing for interleaving events, (ii) require all events of the sequence to happen in the defined order so that (iii) the last event of the sequence can be interpreted to generate the event at a higher level.

This is the process of building narratives at multiple levels regardless of the multi-agent technology itself. The event set and the explanation functions are derived by examining the desired technology or abstraction. Once defined for each level, the next step involves creating a mapping function between levels, which depends on the abstraction of the level below.

In the following, first, we present a formalisation of the approach on which the narrative and generation are based (Sect. 4.1). Then, for each level (Sect. 4.2 for the implementation level and Sect. 4.3 for the design level), we illustrate the process of constructing narratives using agents programmed with Jason as an example. We describe the adopted process, detailing the set of events, explanation and mapping functions. The narrative at the implementation level delves into details related to Jason technology. Then, grounded on this level, we elevate the abstraction to build narratives at the design level: we have chosen the more general

abstractions of the BDI model for this level, as they provide a relatively general set of concepts capable of representing an agent's behaviour.

4.1 Generating levels of narratives: a formalisation

We here present our proposal in a formal style using sets, functions and logical formulae. We use this formalisation to clearly express the main concepts and properties of the proposal. It is also used to help us properly describe examples in the next sections.

Each level has its own set of events. E^i is the set of events for the level i, i is 0 for the implementation level, $i = 1$ for the design level, and $i = 2$ for the domain level. Based on these events, we can build a narrative. A *narrative* is a sequence S^i of events $e^i \in E^i$ observed at level i. For instance, a sequence of events observed from moment 1 until moment n in the implementation level (level 0) can be represented by:

$$S^0 = \langle e^0_1, e^0_2, \ldots, e^0_n \rangle$$

A narrative in any upper level S^i ($i > 0$) is necessarily built on top of S^{i-1}. The function $b^i : S^i \rightarrow 2^{S^{i-1}}$ defines how a sequence of events at S^i is built on top of events from a sequence at S^{i-1}. For example, $b^1(e^1_1) = \{e^0_1, e^0_3\}$ indicates that the first event of a narrative in the design level (e^1_1) is due to the events e^0_1 and e^0_3 occurring in the implementation level. For any event in the upper levels, there must be at least one event in the level below that justifies its existence:

$$\forall_{i \in \{1,2\}} \forall_{e \in S^i} \quad b^i(e) \neq \{\}$$

For the implementation level, the b^0 function has a particular definition, since this level is not built on top of others:

$$\forall_{e \in S^0} \quad b^0(e) = \{\}$$

An *explanation* is modelled then as a relation between events *in the same level*. The function $\rho^i : S^i \rightarrow 2^{S^i}$ gives the set of events that explains some other event at level i, for instance, $\rho^0(e^0)$ is the set of events that explains e^0 in a narrative at the implementation level. The particular meaning of that explanation is platform or level-dependent. It can be, for instance, the causes of some event, like in "e^1_3 is caused by e^1_1" ($\rho^1(e^1_3) = \{e^1_1\}$). Specific platforms can enrich that explanation with more details. For instance, "the action e^1_3 is motivated by having the goal g (represented by the event e^1_1) and the agent belief in b (e^1_0)" ($\rho^1(e^1_3) = \{e^1_1, e^1_0\}$). Using the function ρ, explanations can be chained. For instance, $\rho^1(e^1_1) = \{e^1_0\}$ may explain that goal g exists because the agent perceived something (represented by the event e^1_0); then $\rho^1(e^1_0)$ explains that perception, and so on. Of course, some events might not require explanation, for example, the event of reading a value from a sensor, and initial goals defined by the developer.

Explanations at different levels should be consistent. For instance, as shown in Fig. 5, if an event e^1_3 is explained by e^1_1 in the design level, and e^0_3 is mapped to e^1_3 and e^0_1 is mapped to e^1_1 (the dashed arrows in the diagram), then e^0_1 should be part of the explanation for e^0_3 (the solid arrow in the diagram).
This requirement can be generally expressed as follows.

$$\forall_{e^1, e'^1, e^0, e'^0} \quad e'^1 \in \rho(e^1) \wedge e'^0 \in b(e'^1) \wedge e^0 \in b(e^1) \implies e'^0 \in \rho(e^0)$$

The explanation function captures just a simple relation between events. What we want to highlight however is that explanations must be intra-level and the relations defined by ρ are

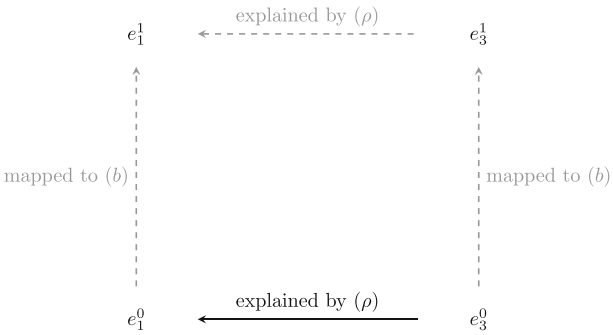


Fig. 5 Relationship of events between different levels

meaningful inside the level. The user does not need to be aware of other levels or functions b to understand the explanation.

Based on the sets and functions introduced in this section, we can define a level L^i as follows:

$$L^i = \langle E^i, b^i, \rho^i \rangle$$

composed of a set of events E^i, a mapping function b^i, and an explanation function ρ^i.

4.2 Building the implementation level

To build the narrative at the implementation level, we have chosen the Jason language [10]. The Jason interpreter operates by continuously executing a reasoning cycle for perceiving the environment and deciding the actions to take. In this section, we describe the process to build this level, following the process described above. Since this level is the level 0, we define only the event set E^0 and the explanation link function ρ^0, the mapping function b^0 is not necessary since the level is built by directly processing the system logs.

4.2.1 Defining the event set for the implementation level: E^0

In this section, we present the set of events identified for Jason as the reference technology. The process of identifying the core events begins by analysing each step in the Jason reasoning cycle. At the beginning of each reasoning cycle, the agent senses the environment and updates its beliefs accordingly. The agent is able to perceive the current state of the environment (i.e., referred to as percept) and communication from other agents (i.e., referred to as speech act messages). These perceived external changes or communications may lead to updates in the agent's belief base. Table 1 illustrates the set of events at the implementation level that the agent can sense, along with the subsequent update of the belief base and the plan library. Events are associated with a type and a narrative template that is used to generate the natural language description of the event. Each template includes placeholders for the specific information that is retrieved when the event is registered.

Another important construct in Jason is *goals*, representing the desired states that the agent aims and intends to achieve by executing actions. Table 2 illustrates the set of events referring to the events for the agent's goal and plan selection. Table 3 contains the events for the agent's intention and action.

Table 1 Event set at the implementation level related to changes in the agent's belief base and plan library, perceptions, and messages

Event type at implementation level	Definition
Reasoning Cycle Started: "New reasoning cycle started: `[num_cycle]`"	Start of a new reasoning cycle numbered `[num_cycle]`
New Percept: "New `[type]` percept `[percept]` from `[source]`"	A new perception of the environment is sensed by the agent. The information of the `[type]` and `[source]` is registered for the `[percept]`
Mailbox Messages: "Messages in mailbox: `[performative]` message from `[agent]`: `[content]`, `[performative]` message from `[agent]`: `[content]`,..."	All unprocessed messages that have arrived in the agent's mailbox. Each message is associated with its `[type]`, the sender `[agent]`, and the `[content]` of the message
Selected Message: "Selected message: `[content]`"	The message `[content]` is selected by the agent for processing
New Speech Act Message: "New speech act message `[performative]` from `[agent]`: `[content]`"	Creation of a new speech act message with its `[performative]`, sender `[agent]`, and `[content]`
Plan Added: "Plan `[plan]` added to the plan library"	The `[plan]` is added to the agent's plan library
Plan Removed: "Plan `[plan]` removed from the plan library"	The `[plan]` is removed from the agent's plan library
Belief Added: "Added belief `[belief]` from source `[source]`"	Addition of a new `[belief]` perceived from `[source]` to the agent's belief base
Belief From Src Added: "Added source `[source]` to belief `[belief]`"	Addition of a new `[source]` annotation in a `[belief]` already present in the agent's belief base
Belief From Src Removed: "Removed source `[source]` from belief `[belief]`"	Removal of a specific `[source]` from a `[belief]`, the belief itself may still be maintained by the agent through other sources
Belief Removed: "Removed belief `[belief]`"	The `[belief]` is removed from the agent belief base, i.e. the agent no longer maintains that belief

The initial state of the goal is pending,[2] indicating that the goal has been added to the agent's event queue. Upon discovery of an applicable plan, the goal transitions to the execution state and becomes an intention. An intention is created and can assume the states of running, suspended, or waiting, depending on the goal. The agent fully commits to pursue it by executing the actions in the plan. Actions can be classified into internal and external actions. Internal actions impact the agent internal state, and only their completion is recorded. External actions, which change the environment, are not directly controllable by the agent itself. For these actions, we record the event that triggered the action, and their completion or failure.

Certain events could cause the goal to wait or stop. The transition to a waiting state can occur under two circumstances: (i) the agent needs to execute an action; or (ii) it is invoked explicitly by the `.wait` internal action, with the information of the amount of time or event (e.g., a specific belief to be added or goal to be triggered) to wait for. For actions that may take time to execute, the intention is suspended, enabling the agent to await feedback on the action's execution and manage other intentions in the following reasoning cycles. The goal may be

[2] Refer to https://github.com/jason-lang/jason/blob/main/doc/tech/goal-states.pdf for more details about all possible goal states.

Table 2 Event set at the implementation level related to goals and plan selection

Event type at Implementation Level	Definition
Goal Created: "Goal `[goal]` created, state: pending"	The goal `[goal]` is created with the initial state pending
Select Plan: "Plan options for `[goal]` are: `@[label] [goal]: [ctxt] <- [body].` `@[label] [goal]: [ctxt] <- [body].` The plan selected for `[goal]` is: `@[label]` "	A set of plans to achieve the `[goal]` is built from the available plan options. Each plan is denoted by a label `[label]` and consists of a context `[ctxt]` and a body `[body]`
Plan Selected: "Plan `@[label]` selected for goal `[goal]`, state: executing"	The plan denoted `[label]` with the context `[ctxt]` and the body `[body]` is selected for the goal `[goal]` and is now in the executing state
Goal Suspended: "Goal `[goal]` suspended because `[reason]`"	The state of the goal `[goal]` is suspended, because of `[reason]`. For instance, a goal may be suspended when the internal action `.suspend` is called, leading to the suspension of the current intention
Goal Removed: "Goal `[goal]` removed because the goal is `[final_state]`"	The goal `[goal]` is removed, indicated with its `[final_state]`, which is dropped, achieved, or failed

resumed and executed again. The goal concludes by moving to one of the three final states: (i) *dropped*, invoked by the internal action `.drop_intention` or `.drop_desire`; (ii) *achieved*, with the successful completion of the plan; or (iii) *failed*, occurring when no applicable plan exists, the plan has failed, or the internal action `.fail_goal` is invoked.

4.2.2 An explanation function: ρ^0

As introduced in Sect. 4.1, the function ρ explains events based on other events at the same level. At the implementation level, explanations can directly refer to the operational semantics of the agent programming language (Jason, in this case), which rigorously defines—either directly or indirectly—causal rules among events. For example, the event produced by the execution of the action `get(beer)` may be caused by the successful execution of the action `open(fridge)`, this causality is ultimately defined by the agent plan. Another possibility is that the event is caused by the goal `has(owner,beer)`. To highlight the importance of goals in Jason agents, we initially decided to base explanations on goals: *events are caused (and explained) by goals*. Thus, in our example, the event `get(beer)` is explained by the goal `has(owner,beer)`.

At the implementation level thus, for some events, we are able to identify the related goal. For example, the third event of a narrative e_3^0 ="External action `get(beer)` triggered" is explained by the first event of that narrative e_1^0 = "Goal `has(owner,beer)` created"; $\rho^0(e_3^0) = \{e_1^0\}$. We can identify the link between these two events because they are related to the same intention. Some other events however do not have associated goals and therefore no explanation. For instance, for the event e_4^0="New Percept" we have $\rho^0(e_4^0) = \{\}$.

To exemplify the implementation of such an explanation function ρ^0 for the implementation level, we introduce two functions that give some properties of the events:

Table 3 Event Set at the Implementation Level related to intention, action, and send message

Event type at implementation level	Definition
Intention Created: "Intention `[id]` `[goal]` created, state: running, current step: `[intention_step]`"	The intention `[id]` corresponding to the goal `[goal]` is created, and its state is running. The `[intention_step]` refers to the step in the body of the plan in execution
Intention Waiting: "Intention `[id]` waiting because `[reason]`, state: waiting, current step: `[intention_step]`"	The intention `[id]` transitions to a waiting state due to the start of the executing an action or for a certain period (i.e., `[reason]`). The current step of the intention `[intention_step]` is recorded
Intention Suspended: "Intention `[id]` suspended because `[reason]`, state: suspended, current step: `[intention_step]`"	The intention `[id]` is suspended to facilitate the execution of actions that may require time (i.e., `[reason]`). The current `[intention_step]` is registered
Intention Removed: "Intention `[id]` removed because `[reason]`, state: undefined"	The intention `[id]` is removed when all the courses of action in the plan are finished (as indicated by the `[reason]`). Removing the intention directly corresponds to the removal of a goal
Internal Action Finished: "Internal action `[action]` finished"	This event is triggered when the execution of the internal action `[action]` (i.e., actions run internally within the agent) is completed
External Action Triggered: "External action `[action]` triggered"	This event marks the initiation of the execution of an external action `[action]` (i.e., actions that will affect the environment, executed outside the agent's mind)
External Action Finished: "External action `[action]` executed"	This event signifies the completion of the external action `[action]` with the final state `[state]`
External Action Failed: "External action `[action]` failed, error `[error]`"	This event signifies the failure of the external action `[action]`
Executed Deed: "Deed `[deed]` executed, type: `[type]`"	This event registers the execution of the deed `[deed]`. The `[type]` of the deed specifies if it is an achievement goal, test goal, mental note, or expression
Send Message: Send `[type]` message to `[agent]`: `[content]`	This event records the message to be sent with its `[performative]` (i.e., tell, achieve, or signal), the destination `[agent]`, and the `[content]` of the message

$type : E^0 \rightarrow T^0$, where T^0 is the set of types of events identified in Tables 1, 2 and 3 ($T^0 = \{\texttt{goal\_created}, \texttt{select\_plan}, ...\}$). $type(e)$ is thus the type of event e.

$int : E^0 \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of numbers identifying intentions, since in Jason every intention is identified by a number. $int(e)$ is thus the intention of event e.

Then we can define ρ^0, given a narrative $S^0 = \langle e_1^0, e_2^0, \ldots, e_n^0 \rangle$ at the implementation level, for some event e_i^0 ($1 \leq i \leq n$) as follows:

$$\rho^0(e_i^0) = \begin{cases} \{e_j^0\} \textbf{ if } \exists_{e_j^0 \in S^0}\ j < i \wedge type(e_j^0) = \texttt{goal\_created} \wedge int(e_j^0) = int(e_i^0) \\ \{\} \quad \textbf{otherwise} \end{cases}$$

The condition $int(e_j^0) = int(e_i^0)$ ensures the link between the two events: they are produced by the same intention. In our example, that very intention is the intention executing the first plan of Listing 1. Considering that j of e_j^0 is the closest to i of e_i^0, we are also *chaining* the explanation of sub-goals based on other goals. For instance, in the case that goal `g0` has `g1` has a sub-goal and the plan for `g1` executes action `a1`, the explanation of `a1` is `g1` and the explanation of `g1` is `g0`.

This explanation function is introduced here as a kind of example for Jason and it is quite simple (many events are explained by just a single other event i.e., the originating goal). Of course, this function could be further improved by considering more complex forms of explanation as we discuss in Sect. 6.

4.3 Building the design level

As discussed in Sect. 3.4.2, the Knowledge Level and BDI provide an effective level of abstraction for our purposes to describe agent behaviour. In this section, we describe a concrete model to define the narratives at the Design Level based on BDI concepts, in spite of the specific technologies used to implement agent(s) at the layer below.

We follow the same process as the previous section: first, we define the set of events E^1 and its explanation function ρ^1. Then we define the mapping function b^1 to associate the sequences of events from the implementation level to events at the design level.

4.3.1 Defining the event set for the design level: E^1

The set of the main events identified for the Design Level is listed in Table 4. As discussed in Sect. 3.4.2, they allow for tracking the behaviour of each agent of the system as a rational BDI agent, in spite of the concrete programming language or technology used for its implementation.

The set includes events related to the agent's beliefs, goals, intentions, and actions. Events related to beliefs involve modifications to the agent's belief base, such as the addition of new beliefs, the removal of outdated ones, or updates to existing beliefs. The goal may be adopted and becomes a new goal for the agent to pursue. New intentions are created to accomplish the goal using a specific plan that ultimately leads to their accomplishment. There could also be situations when the execution of an adopted goal encounters failure or even dropping a goal if it is unfeasible to be accomplished. Events related to intentions focus on the creation of new intentions and their eventual accomplishment or failure. Action related events capture the execution or failure of actions driven by ongoing intentions. It is worth remarking that this set could be further refined to consider a more fine-grained tracking of the behaviour as a

Table 4 Events types at the design level related to goals, intentions, and actions

Event type at design level	Description
New Belief: "New belief `[belief]` from `[what]`"	The agent has a new belief `[belief]`. The `[what]` denotes where the belief comes from (e.g. from a new perception of the environment, or from a message sent by another agent)
Belief Removed: "Belief `[belief]` removed"	The agent does not believe anymore `[belief]`
Belief Updated: "Belief updated from `[old_belief]` to `[new_belief]`"	The agent updated its belief `[old_belief]` into `[new_belief]`
New Goal: "New goal adopted `[goal]` from `[what]`"	The agent adopted a new goal `[goal]` to pursue. The `[what]` denotes where the goal comes from (e.g. from a new perception of the environment, or from a message sent by another agent)
Goal Achieved: "Goal `[goal]` was achieved through intention `[intention]`"	The agent completed the intention `[intention]`, achieving the goal `[goal]`
Goal Dropped: "Goal `[goal]` cannot be accomplished because `[reason]`"	The agent cannot go on with the goal `[goal]` because the agent has no means to build or find a feasible plan
Goal Failed: "Goal `[goal]` failed through intention `[intention]`"	The agent failed to accomplish the goal `[goal]` following the intention `[intention]`
New Intention: "New intention `[intention]` created for goal `[goal]` using the plan `[plan-ref]`"	The agent found a feasible plan `[plan-ref]` to accomplish the goal `[goal]`
Intention Accomplished: "Intention `[intention]` accomplished for goal `[goal]` using plan `[plan-ref]`"	The intention is created to accomplish the goal `[goal]` using the plan `[plan-ref]` is accomplished
Intention Failed: "Intention `[intention]` failed to accomplish goal `[goal]` using plan `[plan-ref]` - `[reason]`"	The intention is created to accomplish the goal `[goal]` using the plan `[plan-ref]` failed because of the reason described in `[reason]`
Action Executed: "Action `[action]` executed following intention `[intention]`"	The agent executed the action `[action]` within the context of intention `[intention]`
Action Failed - "Action `[action]` failed following intention `[intention]` - `[error]`"	The agent failed to execute the action `[action]` in the context of intention `[intention]`, reporting the error message `[error]`

Table 5 Events and explanation relationships established by ρ^1: links between events create a chain that can build the full explanation composed of relevant past events

Event	Explaining events	Rationale
New Intention	A **New Goal**	The new intention event is explained by the goal that motivated its creation. That goal is represented by the event that created the goal
Intention Accomplished	Set of **Action Executed**	The intention's accomplishment is explained by all actions executed by its plan
Intention Failed	An **Action Failed**	The intention's failure is explained by the very action in the intention's plan that failed to execute
Action Executed	A **New Intention**	The explanation for the action's execution is the intention that contains a plan to execute this action. That intention is represented by the event that created the intention
Goal Achieved	An **Intention Accomplished**	The explanation for the achievement of a goal is the achievement of the corresponding intention. That intention is represented by the event that created the intention
Goal Failed	An **Intention Failed**	The explanation for the failure of a goal is the failure of the corresponding intention

BDI agent. For instance: here we consider only goals as those desires that are consistent and that the agent decides to commit to. However, tracking desires can also be useful in making observable decisions about which goals to accomplish given multiple possibly inconsistent objectives.

4.3.2 An explanation function: ρ^1

At the design level, the explanation function is meant to capture those cause-effect relationships among events that are useful to explain the agent behaviour for designers and architects. In this case, to define the ρ^1 function we can refer to the high-level semantics of abstract BDI architectures, as defined e.g. in [45], which is independent of specific technologies and implementation details.

Table 5 shows the cause-effect relationships that we considered in this paper for defining ρ^1. New goals and new beliefs events have no explanation based on other events, they carry all the relevant information for their explanation within the event itself. Differently, the creation of a new intention is explained by a goal that was previously adopted and the execution of an action is explained by an intention that was previously created to achieve the goal. In case of a failure of a goal, this would be explained by an intention failure, which would be further explained by the action causing the failure. We consider explaining the causes of such action failure out of the scope of this work as that would require introducing an additional level of reasoning in the generation of the explanations starting from the information presented in the narratives. Additionally, the true cause of the failure might not be known to the agent performing an external action and depend only on a failure in the external environment. Conversely, when a goal is not accomplished due to the agent being unable to find an appropriate plan to handle it, the event in the narrative will be a "Goal Dropped"

event because no intention was created to handle that specific goal and the explanation will report the conditions of failing to match the plan in the payload of the event (See Table 4).

As in the implementation level, some auxiliary functions are introduced to help us define the explanation function ρ^1:

- $type : E^1 \rightarrow T^1$, where T^1 is the set of types of events identified in Table 5 ($T^1 = \{\texttt{new\_intention}, \texttt{action\_executed}, ...\}$). $type(e)$ is thus the type of event e.
- $goal : E^1 \rightarrow E^1 \cup \{\epsilon\}$, $goal(e)$ is the event that created the goal that explains the `New Intention` event e as defined in Table 5. If $goal(e) = \epsilon$, there is no goal assigned to the event e.
- $aint : E^1 \rightarrow E^1 \cup \{\epsilon\}$, $aint(e)$ is a new intention event representing the intention that produced the execution of the action represented by the event e. If $aint(e) = \epsilon$, there is no intention to assign the event e.

We assume here that the assignments defined by $goal$ and $aint$ reflect the deliberation process of the BDI engine that links goals, intentions, and actions. When defining these functions, it is important to preserve the consistency of the explanations with regard to the implementation level, as such this function should also consider how events are mapped between levels.

Given a narrative $S^1 = \langle e_1^1, e_2^1, \ldots, e_n^1 \rangle$ at the design level, we can define ρ^1 for some event e_i^1 $(1 \leq i \leq n)$ as follows:

$$\rho^1(e_i^1) = \begin{cases} \{goal(e_i^1)\} & \textbf{if } type(e_i^1) = \texttt{new\_intention} \wedge goal(e_i^1) \neq \epsilon \\ sa & \textbf{if } type(e_i^1) = \texttt{intention\_accomplished} \wedge \\ & \quad sa = \{e_j^1 | e_j^1 \in S^1 \wedge type(e_j^1) = \texttt{action\_executed} \wedge aint(e_j^1) = e_i^1\} \\ \{e_j^1\} & \textbf{if } type(e_i^1) = \texttt{intention\_failed} \wedge \\ & \quad \exists_{e_j^1 \in S^1}\ type(e_j^1) = \texttt{action\_failed} \wedge aint(e_j^1) = e_i^1 \\ \{aint(e_i^1)\} & \textbf{if } type(e_i^1) = \texttt{action\_executed} \\ \{e_j^1\} & \textbf{if } type(e_i^1) = \texttt{goal\_achieved} \wedge \\ & \quad \exists_{e_j^1 \in S^1}\ type(e_j^1) = \texttt{intention\_accomplished} \wedge goal(e_j^1) = e_i^1 \\ \{e_j^1\} & \textbf{if } type(e_i^1) = \texttt{goal\_failed} \wedge \\ & \quad \exists_{e_j^1 \in S^1}\ type(e_j^1) = \texttt{intention\_failed} \wedge goal(e_j^1) = e_i^1 \\ \{\} & \textbf{otherwise} \end{cases}$$

Through the definition of such basic links, designers can go back into the narrative of the execution, only following the relevant events for the event to explain, isolating specific courses of action from the others. The complete explanation for an event would then be considered as the graph of past connected events through a relationship established by ρ^1, as illustrated in Fig. 6.

4.3.3 From implementation to design: b^1

The function b^1 in the design level describes how events at this level can be generated (mapped) from events occurring at the implementation level, that is: what are the conditions that make it possible to assert that a certain event at the design level occurred, given a set of events occurring at the implementation level.

While the event set is predefined regardless of the technology itself, the generation of each event depends on the event sequence at the implementation level. For instance, as illustrated in Fig. 7 the event "New Intention" is added to the design level because both events "Plan

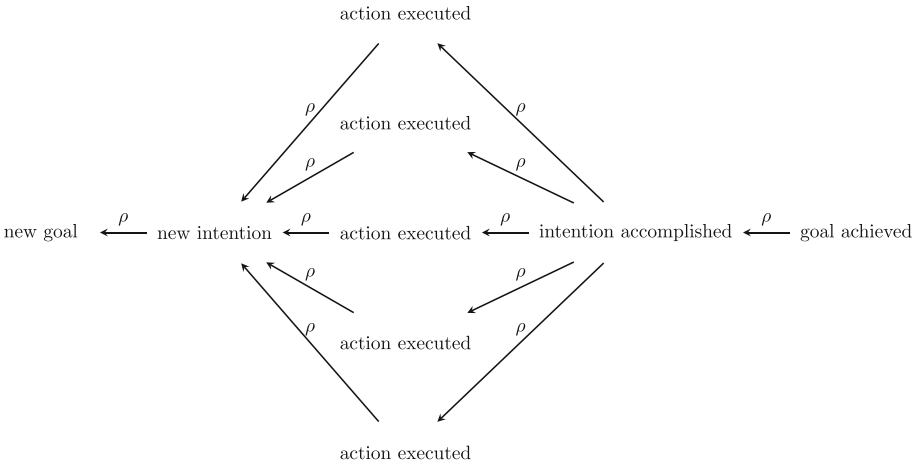


Fig. 6 Typical explanation graph at Design Level as produced by ρ^1

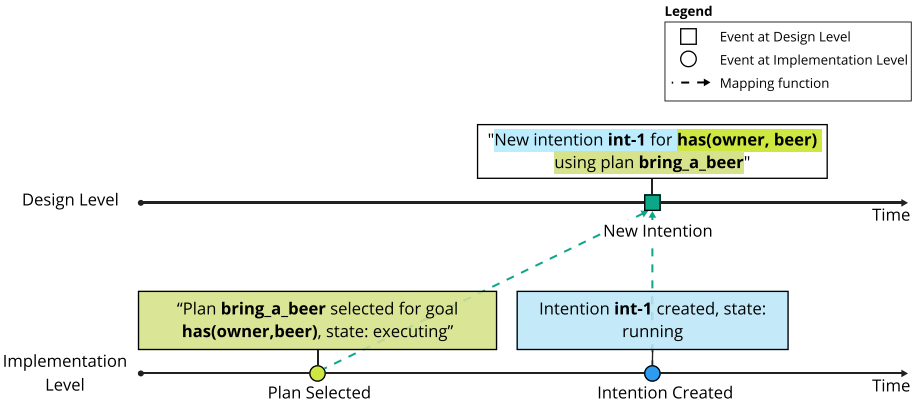


Fig. 7 Example of the process of building a narrative from the implementation level to the design level

Selected" and "Intention Created" happened at the implementation level. In the following, we describe the mapping function b^1 for each event in E^1 identified in Sect. 4.3.1.

Table 6 illustrates the mapping of events related to the agent's belief. Each row of the table is a possible mapping. If we identify the sequence of events in the first column at the implementation level, we can add the mapped event in the second column at the design level. Some placeholders of events in the table are replaced by specific instances that form the mapping, represented in italics. The "New Belief" can be generated in three different patterns of events at the implementation level indicated by the `source` of the belief. Firstly, an agent acquires certain beliefs through perception of the environment. In this scenario, a "New Belief" event (e_0^1) in a narrative at the design level is mapped to the sequence of events "New Percept" (e_1^0) and "Belief Added" (e_2^0) in a narrative at the implementation level. Formally, $b^1(e_0^1) = \{e_1^0, e_2^0\}$. The formalization of the next mappings is similar and quite straightforward and thus will be omitted. Secondly, an agent may acquire a certain belief based on information received from other agents, communicated through a speech act message. Here, a "New Belief" event is associated with the event sequence composed of

Table 6 Belief event mapping

Event sequence at implementation level		Mapped new belief event at design level
1. **New Percept**: "New `[type]` percept `[percept]` from `[source]`" 2. **Belief Added**: "Added belief `[belief]` from source *percept*"	⇒	**New Belief** "New belief `[belief]` from `[source]`"
1. **New Speech Act Message**: "New speech act message *tell* from `[source]`: `[content]`" 2. **Belief Added**: "Added belief `[content]` from source `[source]`"	⇒	**New Belief**"New belief `[content]` from `[source]`"
1. **Belief Added**: "Added belief `[belief]` from source *self*"	⇒	**New Belief** "New belief `[belief]` from *itself*"
1. **Belief Removed**: "Removed belief `[old_belief]` from source `[source]`" 2. **Belief Added**: "Added belief `[new_belief]` from source `[source]`"	⇒	**Belief Updated** "Update belief `[old_belief]` to `[new_belief]`"
1. **Belief Removed**: "Removed belief `[belief]`"	⇒	**Belief Removed** "Removed belief `[belief]`"

Table 7 New goal event mapping

Event sequence at implementation level		Mapped event at design level
1. **Belief Added**: "Added belief `[belief]` from source `[source]`" 2. **Select Plan**: "Plan options for `[belief]` are: `[plan-ref] [belief]: [ctxt] <- [goal].` The plan selected for `[belief]` is `[plan-ref]` 3. **Goal Created**: "Goal `[goal]` created, state: pending"	⇒	**New Goal** "New goal adopted `[goal]` from `[belief]`"
1. **Goal Created**: "Goal `[goal]` created, state: pending" 2. **Intention Created**: "Intention `[id]` `[goal]` created, current step: `[subgoal]`" 3. **Goal Created**: "Goal `[subgoal]` created"	⇒	**New Goal** "New goal adopted `[subgoal]` from `[goal]`"
1. **Goal Created**: "Goal `[goal]` created, state: pending"	⇒	**New Goal** "New goal adopted `[goal]`"

"New Speech Act Message" and "Belief Added" events. Lastly, a belief may be added by the agent itself (denoted by the source annotation *self*) for future reference.

Table 7 shows the corresponding mapping events for a new goal. There are three distinct ways through which an agent can acquire a new goal, specified by the `source` placeholder: (i) the agent believes something ("New Belief") that brings to a new goal; (ii) a *sub*-goal formed while pursuing another ("Goal Created") through the execution of a plan; (iii) it is the agent's initial goal assigned by the developer. We map the "Goal Created" event at the implementation level (with the events that signal its creation) to a "New Goal" event, with the specification of where the source of the goal comes from. To better understand this mapping and the reason by using the concept of goal instead of desire in the BDI abstraction, it is useful to recall that in the abstract BDI model [13] taken as a reference at the design level, a desire is not immediately a goal; it becomes a goal when the agent decides to commit to

Table 8 Intention and action events to achieve a goal mapping

Event sequence at implementation level		Mapped event at design level
1. **Plan Selected**: "Plan `[plan-ref]` selected for goal `[goal]`, state: executing" 2. **Intention Created**: "Intention `[id]` created, state: *running*"	⇒	**New Intention** "New intention `[id]` for `[goal]` using plan `[plan-ref]`
1. **Intention Created**: "Intention `[id]` created, state: *running*, current step: `[action]`" 2. **External Action Triggered:** "External action `[action]` triggered" 3. **External Action Finished**: "External action `[action]` executed"	⇒	**Action Executed** "The action `[action]` executed following the intention `[id]`"
1. **Intention Removed**: "Intention `[id]` removed because *plan finished* 2. **Goal Removed**: "Goal `[goal]` removed because the goal is *achieved*"	⇒	**Goal Achieved** "The goal `[goal]` was achieved through intention `[id]`"

it. This distinction is blurred instead in Jason, adopted at the implementation level: there is not an explicit account for desires, or every desire becomes immediately a goal that the agent tries to achieve. Different mappings with other technologies that account for desires may instead include the concept at this level, for instance, if the agent has a more complex deliberation process among desires.

As soon as the agent has a goal, it starts looking for the means (plans) to achieve it. Table 8 illustrates the events related to the intention and action mapping functions for achieving a goal. When (if) a plan is found and selected, a "New Intention" is created and the agent commits to execute the corresponding course of actions. For the execution of an action ("Action Executed") at the design level, we decided to keep only the external actions. Indeed, the internal actions are not meaningful at this level and are strictly related to the implementing technology. When all actions present in the plan are successfully completed, the intention is accomplished, and the goal is considered achieved. Thus, "Goal Achieved" is generated when the intention *finished* the plan and the goal moves to the state *achieved*.

Table 9 shows the events generated in case of some failure to achieve the goal. An action may fail, in this case, the event "Action Failed" comes from the failure of the external action associated with its intention. Then, the intention is failed ("Intention Failed") with the information of the reason (e.g., failure of an action).

There are two different events related to the failure of a goal: (i) "Goal Dropped", when the goal cannot be accomplished and the goal moves to the state *dropped*, (ii) "Goal Failed", the intention for the goal is failed (e.g., due to a failed action).

4.4 Building the domain level

Increasing the abstraction further, we are abstracting from how the system has been designed, focusing more on what functionalities the system is meant to provide. In this case, the set of events and explanation link between events come from the formalisation of the requirements—as shown in Sect. 3.4.1—and it is totally domain-dependent. In this section, we briefly show an example of how it would be possible to build this level following a similar process to the other levels below.

Table 9 Mapping of action and intention events related to the failure of a goal

Event sequence at implementation level		Mapped event at design level
1. **Intention Created**: "Intention `[id]` created, state: *running*, current step: `[action]`" 2. **External Action Triggered:** "External action `[action]` triggered" 3. **External Action Failed**: "External action `[action]` failed, error: `[error]`"	⇒	**Action Failed** "The action `[action]` failed following intention `[id]` – `[error]`"
1. **Plan Selected**: "Plan `[plan-ref]` selected for goal `[goal]`, state: executing" 2. **Intention Removed**: "Intention `[id]` removed because *it failed*"	⇒	**Intention Failed** "Intention `[id]` failed to accomplish goal `[goal]` using plan `[plan-ref]` - `[reason]`"
1. **Goal Removed**: "Goal `[goal]` removed because `[reason]`" 2. **Select Plan**: "Plan options for the *failure* of `[goal]` are: `[plan-ref] [goal]: [ctxt] <- [body].` `[plan-ref] [goal]: [ctxt] <- [body].` The plan selected for `[goal]` is `[plan-ref]` 3. **Plan Selected**: "Plan `[plan-ref]` selected for goal `[goal]`"	⇒	**Goal Dropped** "The goal `[goal]` cannot be accomplished for reason `[reason]` - start executing failure plan `[plan-ref]`"
1. **Intention Removed**: "Intention `[id]` removed because *it failed*, state: *undefined*" 2. **Goal Removed**: "Goal `[goal]` removed because `[reason]`]"	⇒	**Goal Failed** "The goal `[goal]` failed through intention `[id]`"

4.4.1 Defining the event set for the domain level: E^2

The event set corresponds to domain events that can be identified from e.g. use cases, or any artefacts capturing functional and non-functional requirements. Table 10 shows an example related to the "Get a beer" use case, described in Sect. 3. Differently from the design level, here events describe the unfolding story of the owner as the primary actor pursuing her goal to get a beer. Some events refer to the main scenario (the successful one), while others refer to the extensions related to the impossibility of accomplishing the goal.

4.4.2 An explanation function: ρ^2

The explanation function ρ for this level is a formalisation of the knowledge about causes and effects relationships at the domain level, in particular those that could be relevant to build explanations. In the running example of the domestic robot, explanations could concern the reasons why the robot refused to deliver a beer, or delivered a beer.

User stories and system stories [25] can be a valuable source to grasp this knowledge, additionally to use cases, since they focus on the reasons why events related to users and the system happen. Table 11 shows some main examples of explanation links among events described in Table 10, based on the "Get a beer" user and system stories described in Sect. 3.

4.4.3 From design to domain: b^2

The function b^2 formalises the knowledge bridging the design and domain level, by identifying how a new domain event at the domain level corresponds to (or, is generated by) a pattern

Table 10 Events types at the domain level, related to the domestic robot example

Event type at domain level	Description
New beer requested	The owner asked the robot for a beer
Daily limit validated	The robot validated the daily limit
Beers availability checked	The robot verified the availability of at least one beer in the fridge
Beer delivered	The robot delivered the beer to the owner
Daily limit not validated	The robot did not validate the daily limit
Beer request refused due to daily limits	the robot informed the owner about the impossibility of delivering a beer due to daily limits
Beer request refused due to unavailability	the robot informed the owner about the impossibility of delivering a beer due unavailability
New stock ordered	the robot ordered a new stock of beer from the supermarket

Table 11 Explanation links between events at the domain level

Event Type	Reason
Beer request refused due to daily limits	**Daily limit not validated**
Beer delivered	**Daily limit validated**, **Beer available**
New stock ordered	**New beer requested**, **Beer not available**

Table 12 Incomplete examples of design to domain mapping

Event sequence at design level		Mapped event at design level
New Goal: `has(owner, beer)`	⇒	**New beer requested**
Goal Dropped: `has(owner, beer)` with repairing plan `too_much_beer`	⇒	**Beer request refused due to daily limits**

of events at the design level. Like for previous levels, this pattern could be just a single event, or a sequence (pattern) of events, in the most general case.

Simple examples of mapping rules for the "Get a beer" use case are shown in Table 12. The first is about binding the generation of a domain event "New Beer Requested" from a "New Desire" event at the design level, specifically matching `has(owner, beer)` desire. The second is about binding the generation of a domain event "Beer request refused due to daily limits" from a sequence of events: a "Goal Failed" event related to the `has(owner, beer)` goal and the corresponding "Plan Selected" event related to the `too_much_beer` plan, that is a repair plan to manage the goal failure. It is worth remarking that the mapping is clearly not unique and depends on designers' (and domain experts') viewpoints.

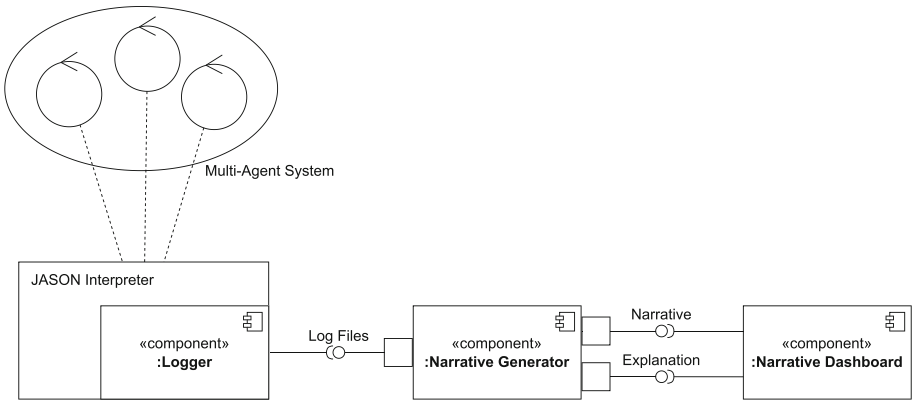


Fig. 8 Overview of the main components of the framework

5 Prototype implementation

To put our conceptual framework into practice, we have experimented with the development of a prototype tool for this multi-level explanation idea, using the approach described in Sect. 4. The prototype serves the purpose of practically generating the narratives and providing a basic interface to navigate them. In its current version, the prototype works offline and can read and load the logs recorded after a system run. It then generates the narratives and displays them in a Web Dashboard. Users can navigate the narratives at the implementation and design level, filter events, and ask for explanations of events.

The main components of the multi-level explainability tool we developed are represented in Fig. 8. Since the approach starts by collecting logs of the system, an important component is the **Logger** (Sect. 5.1), which is responsible for generating the logging trace for each entity in the MAS. The logger is implemented by extending the Jason Interpreter. It records all main events following the reasoning cycle in the Jason architecture, such as belief updates, goal and plan applicability, agent perceptions, and speech-act messages. Logs are stored in a file that serves as the foundation to generate narratives.

The second main component is the **Narrative Generator** (Sect. 5.2), which is responsible for processing the log and building narratives at different levels. The narratives are then presented to the user through a dashboard. Users can access the application, explore and analyse the narrative at different levels and find explanations to understand the behaviour of the agents.

5.1 Logger

For developing the Logger[3] we have extended the Jason interpreter leveraging the feature of customisation of the agent architecture [10, Chapter 7]. Since Jason is developed in Java,[4] the customisation and extension are achieved through the development of libraries and components in Java as well. The overall architecture of the logging component is presented in

[3] The logger component prototype is available at https://github.com/yan-elena/agent-logging, a configured example of domestic robot application can be found at https://github.com/yan-elena/domestic-robot-example.

[4] https://github.com/jason-lang/jason/

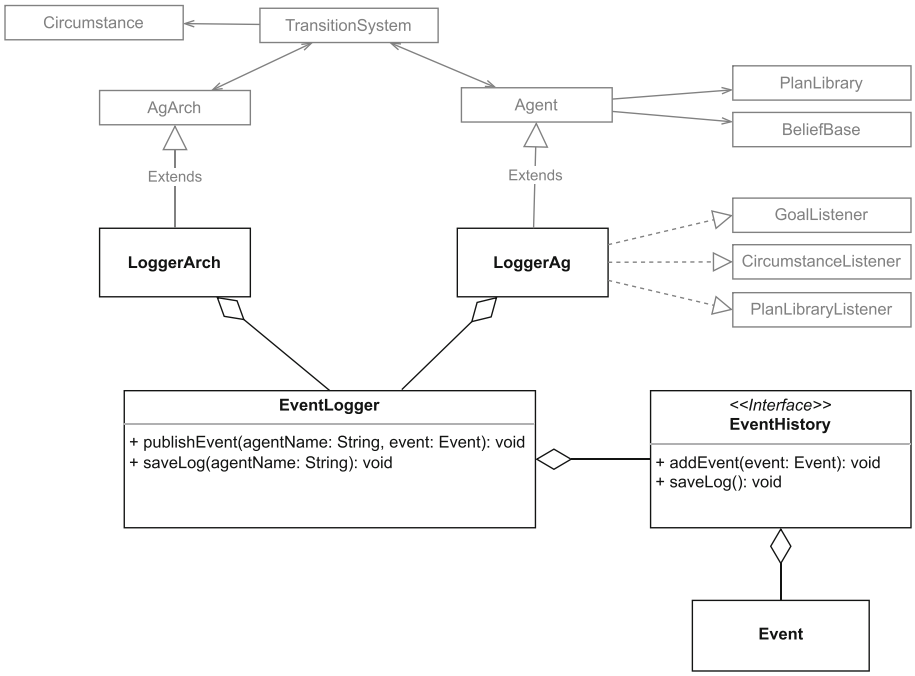


Fig. 9 Logger class diagram

Fig. 9, classes in grey are internal classes of the Jason interpreter. This customisation primarily focuses on the `Agent` class and the `AgArch` class of Jason.

The `Agent` class encompasses the agent's essential components such as the `BeliefBase`, `PlanLibrary`, and the `Circumstance`. The `LoggerAg` extends `Agent` and implements various listener interfaces to enable the agent to listen to related events, such as `GoalListener`, `CircumstanceListener`, and `PlanLibraryListener`.

On the other side, the `LoggerArch` class extends Jason's `AgArch`, which serves as the foundation of the agent's architecture, dictating how the agent interacts with the external world, specifically with the environment and other agents. The agent architecture encapsulates the functionality of the agent's sensory perception, action execution, and communication mechanisms. The `LoggerArch` class, in conjunction with `LoggerAg`, is associated with agents that require logging functionality, in the multi-agent system configuration file.

The central element for the logger is represented by the `EventLogger` component, which manages the logs of all agents in the system. It is responsible for publishing the events associated with a specific agent and subsequently saving the log to a file. As we did not intend to run complex or long-running MAS with this prototype, we store the history of all agents in memory and dump them into files. The history of events for a given agent is represented by the `Event History` entity, providing methods for adding new events and saving the log into files. Events are encapsulated by the corresponding `Event` interface, providing a uniform structure to represent different types of events.

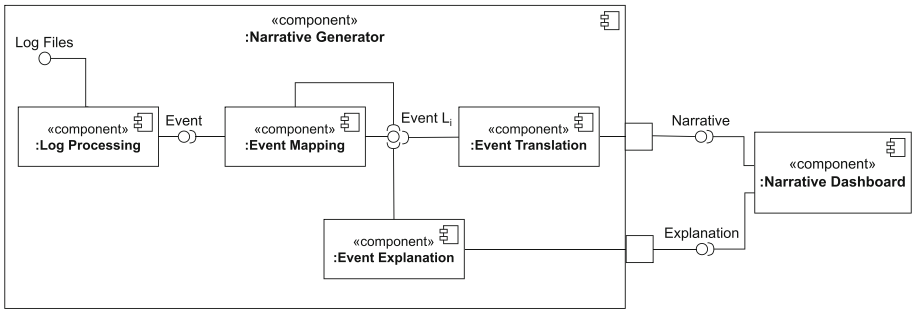


Fig. 10 Main components of the narrative generator

5.2 Narrative generator

The Narrative Generator component[5] is responsible for generating the narrative of an agent on multiple levels. As illustrated in Fig. 10, given the log files, the *log processing* component extracts the events from the log files.

The *event mapping* component applies the mapping functions b^i identified in Sect. 4 by looking at the sequence of events to map to obtain the event at level i. The *event translation* component then collects all events at the requested level to present in the *narrative dashboard*. For events that require explanations, the *event explanation* component is responsible for applying the explanation function ρ^i and retrieving the relevant event to display on the dashboard.

The narrative is then presented in a dashboard that allows users to access and visualise the narrative of the system behaviour. We opted for a Web application for its accessibility and ease of use. Once users have executed the MAS and obtained the logs, they can upload them to the Web application for inspection. The dashboard shows an overview of all agents of the MAS. Users can select which agent and which level of abstraction they want to use to inspect the narrative.

Uploading the log files generated by the *domestic robot* example, described in Sect. 3.4, allows us to view the page illustrated in Fig. 11. This page enables the user to choose the level of abstraction (implementation or design level) for the three agents in the system: owner, robot, and supermarket.

When the user selects an agent and a level to inspect the narrative, the application provides a view of all registered events (Fig. 12). It is possible to notice that, the narrative at the implementation level (Fig. 12a) records all technical and detailed events, even taking into account the internal working of the agent. The narrative is closely related to the work of Jason interpreter. At the design level (Fig. 12b), the narrative is straightforward and immediate, related to BDI concept. Events with an associated explanation (see Section 4.2.2 for the implementation level and Section 4.3.2 for the design level) allow users to visualise the cause of the event. Figure 13 provides an example of such an explanation, where the event type "Action Executed" for the action `move_towards(P)` is explained by the event "New Intention". Thus, the cause of the action, is explained by the creation of a new intention to accomplish the goal `at(robot, P)` and using the plan named `p__6`.

[5] The narrative generator component prototype is available at https://github.com/yan-elena/agent-explanation, an online UI application can be accessed directly via this link: https://yan-elena.github.io/agent-explanation.

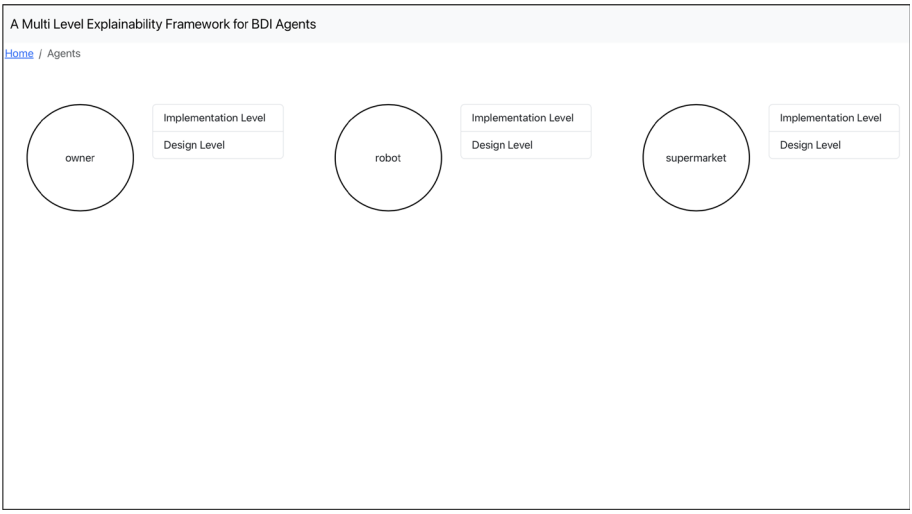
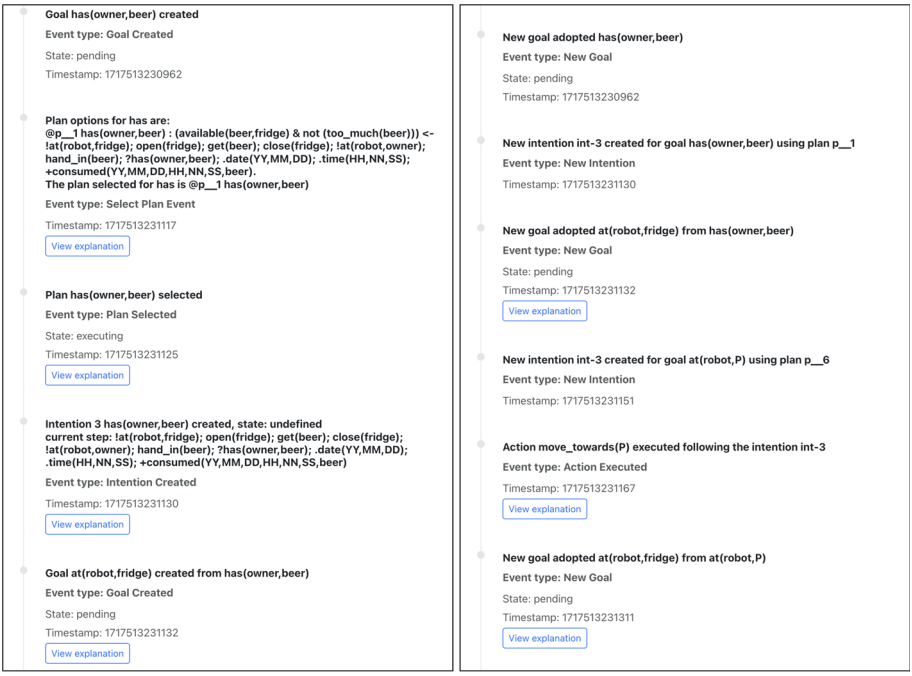
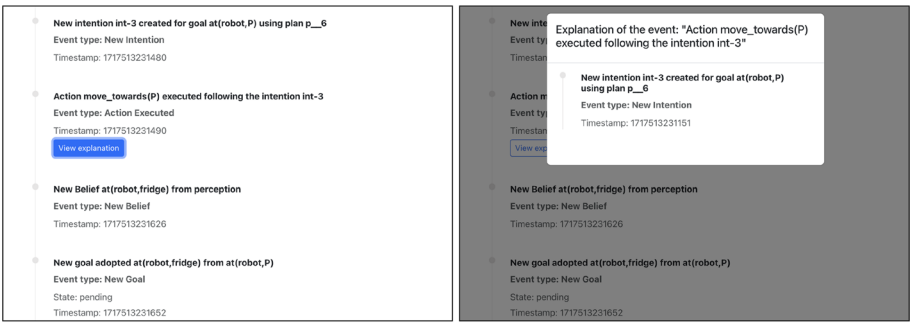


Fig. 11 A screenshot of the web application's overview page displaying all agents in the domestic robot multi-agent system. Users can select individual agents with the level of abstraction to view their respective narratives



(a) Narrative at the implementation level (b) Narrative at the design level

Fig. 12 A screenshot illustrating the narrative of the *robot* agent at implementation and design levels



(a) Events at implementation level (b) Explanation of the event

Fig. 13 A screenshot from the web application. It illustrates the explanation of the event "Action Executed" for the action `move_towards(P)`

A component for filtering the events of interest is also implemented in the prototype, this functionality helps to visualise the entire life cycle of a particular event (e.g., goal or intention at the implementation level and desire at the design level) in somehow similar to the narrative for the single component [29].

5.3 Practical examples

In this section, we delve into practical examples of how the tool can serve its purpose in different situations. We use the domestic robot running example introduced in Sect. 3.4 and answer the questions identified for the design and implementation level (Sect. 4) to verify our generated narrative and test our prototype.

5.3.1 Example at the design level

At the Design Level, the narrative may support the architects and designers in understanding the behaviour to improve and reformulate the requirements of the system. Architects could find answers in the narrative of this level about why the agent intends to do something, or why the agent has some desire as presented in Sect. 4.3.

The narrative at this level abstracts from all the technical details and focuses only on the agent's behaviour in terms of BDI practical reasoning.

(Q2) Why does the robot have the intention of not bringing me the beer?

Scenario: The owner requests beer from the robot, but the robot does not satisfy the request. The owner receives a message from the robot saying "The Department of Health does not allow me to give you more than 10 beers a day! I am very sorry about that!" The user wants to understand why the robot made this decision and why this message was sent.

Explanation: By looking at the explanation of sending the message, we can find the cause illustrated in Fig. 14. The action is executed because of the intention `int-62` created to achieve the goal `has(owner,beer)`, using the plan named `p__3`.

Narrative: The narrative of this scenario is illustrated below.

```
[New Goal] New goal adopted has(owner, beer)
[Goal Dropped] Goal has(owner, beer) cannot be accomplished because
    there was no applicable plan
```

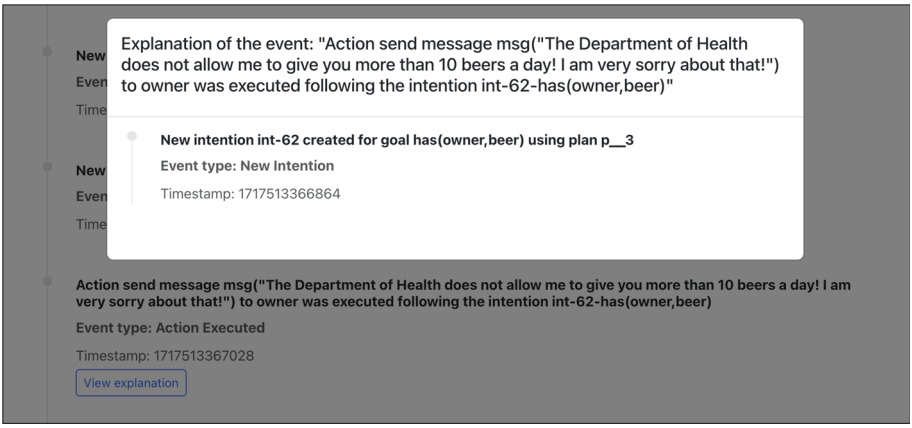


Fig. 14 Explanation of the action executed

```
[New Intention] Intention int-62 has to manage a failure in goal
    has(owner,beer) using the plan p__3
[Actioin Executed] Action send message msg("The Department of Health
    does not allow me to give you more than 10 beers a day! I am very
    sorry about that!") to the owner was executed following the
    intention int-62
...
```

By analysing the narrative starting from the goal `has(beer)` adopted by the robot, we can find the events that lead to this decision. After the robot has adopted the goal, it is dropped and could not be accomplished because there was no applicable plan. Then, the robot has a new intention to handle the failure of the goal, by choosing the plan `plan__3`.

5.3.2 Examples at implementation level

Since the narrative of the implementation level is closely related to the technology of the multi-agent system, it can be understood by the developers and could be used to support the debugging phase.

The developer can find answers in the narrative of this level about why an error occurred, why a plan failed, or why an action was not executed as expected. In this section, we will simulate these situations by modifying the code of the domestic robot MAS to see how the narrative can support the developer in finding the cause of the bug we purposefully introduce.

(Q3) Why did the robot agent fail the plan **`has(owner,beer)`**?

Let's consider an example in which there is a failure in the agent's behaviour and see how the narrative could assist in finding the cause.

Scenario:

```
+!has(owner,beer)
   :  available(beer,fridge) &
      not too_much(beer)
   <- !at(robot,fridge);
      //open(fridge);
      get(beer);
      //...
```

Listing 2 Modified code of the goal `has(owner, beer)` where the robot does not open the fridge before taking the beer

Consider the case where the developer forgot to insert the action `open(fridge)` before the action `get(beer)` in the `robot` agent's code as illustrated in Listing 2. When the programmer runs the system, he notices that the plan has failed and wants to understand the reason for the failure in order to correct it.

Console output. First, let's begin by examining the output generated in the console. There is an exception to indicate the failure to achieve the goal `has(_,_)` with the information about the current intention. There is no specific mention regarding the reason for this failure.

```
...
[robot] doing: get(beer)
[robot] .current_intention( intention(I,S) ) was replaced by
    .intention(I,_,S,current)
[robot] Failed to achieve goal '!has(_,_)'. Current intention is:
intention(3,[im(p__4[code_line(44), code_src("file:src/agt/robot.asl"),
    source(self), url("file:src/agt/robot.asl")], {
    -!has(owner,beer)[code(get(beer)), code_line(22),
    code_src("file:src/agt/robot.asl"), error(action_failed),
    error_msg(""), source(owner)] }, { .current_intention(_63);
    .print("Failed to achieve goal '!has(_,_)'. Current intention is:
    ",_63) },[map(_64_4,owner), map(_65_5,beer)]),
    im(p__1[code_line(19), code_src("file:src/agt/robot.asl"),
    source(self), url("file:src/agt/robot.asl")], {
    +!has(owner,beer)[source(owner)] }, { get(beer); close(fridge);
    !at(robot,owner); hand_in(beer); ?has(owner,beer);
    .date(_66,_67,_68); .time(_69,_70,_71);
    +consumed(_66,_67,_68,_69,_70,_71,beer) },[])])
...
```

Narrative generated by the tool. In addition to the logs from the stack trace, the narrative generated by our tool adds a new layer of explanation to support developers in their debugging phase. The aim of our tool is not to replace existing debugging tools but to provide additional information to find the cause of the failure. Thanks to the events collected by the logs in addition to the explanation causality defined by ρ, it is possible to infer the reason for the failure.

The narrative of the robot agent at the implementation level is presented as follows.

```
...
[External Action Triggered] External action get(beer) triggered
[External Action Finished] External action get(beer) failed
[Goal Removed] Goal has(owner, beer) removed because the action
    .get(beer) failed
...
```

Analysing the sequence, it is possible to notice from the last event that the goal is removed because of a failure, specifically because the action `.get(beer)` failed. Going back a few events, we notice that the action `.get(beer)` is triggered and starts its execution. But its execution fails and leads to the failure of the plan `has(owner, beer)`. Having identified the cause of the failure, the narrative can answer that the robot agent is unable to finish the execution of the action `.get(beer)` and possibly there are not all the preconditions to attempt that action.

An effective explanation of why this external action fails could be provided by integrating explanations from the external environment (e.g., by extending the framework with explanations from artifacts in the environment). See section 6 for further discussion on this..

(Q4) Why did the robot agent send the message `order(beer, 5)` to the supermarket again?

Consider now another example, in which there is no explicit exception at runtime. The developer has to understand why the execution of the system does not behave as expected.

Scenario: When the supply of beer is finished, the *robot* agent orders new beers by sending a message `.send(supermarket, achieve, order(beer, 5))` to the *supermarket* agent. When the beers are delivered, the supermarket agent sends a 'tell' message to inform the robot. The robot agent should update the beer availability in the fridge and then bring the beer to the owner. Suppose that the developer forgets to update the belief `available(beer, fridge)` and tries to achieve the goal `has(owner, beer)` again, as illustrated in Listing 3.

```
// when the supermarket makes a delivery, try the 'has' goal again
+delivered(beer,_Qtd,_OrderId)[source(supermarket)]
  :  true
  <- //+available(beer,fridge);
     !has(owner,beer).
```

Listing 3 Modified code of the event `delivered(beer, _Qtd, _OrderId)`: when the supermarket makes a delivery, the robot tries the `has` goal again without update the belief `available`

Console output. The output generated in the console is illustrated below.

```
...
[robot] doing: move_towards(fridge)
[supermarket] doing: deliver(beer,5)
[robot] Check Time=10:16:12
[owner] time("10:16:12")[source(robot)]
[supermarket] doing: deliver(beer,5)
...
```

After the supermarket has delivered the beer, the robot does not bring the beer to the owner, but there is another beer delivery from the supermarket. Looking at this log, it is not straightforward to identify the reason for this misbehaviour.

Narrative generated by the tool: The narrative at this level records all actions and decisions made by the agent. The narrative of this behaviour for the *robot* agent is illustrated below.

```
...
[Goal Created] Goal has(owner beer) created
[Reasoning Cycle Started] New reasoning cycle started: 146
[Select Plan Event] Plan options for has(owner,beer)[source(self)] are:
    @p__2 has(owner,beer) : not (available(beer,fridge))
        <- .send(supermarket,achieve,order(beer,5)); !at(robot,fridge).
[Plan Selected] The plan selected for has(owner,beer)[source(self)] is
    @p__2
...
```

When the goal `has(owner, beer)` is created, it selects the plan with the condition `not(available(beer, fridge))`. The robot does not believe that the beer is in the fridge (`available(beer, fridge)`), so it reorders the beer from the supermarket and this event leads to the misbehaviour. The narrative generated by the tool could help

the developer understand the cause of this behaviour and realise that he may have missed updating the belief after the supermarket had delivered the beer.

6 Discussion and future works

In this paper, we have explored a new approach to explainability in BDI agents that goes in the direction of considering multiple levels of explanations originating from the same set of logs. The main idea is that explanations should be tailored to the technical proficiency and task of different classes of users, to better support them in understanding the behaviour of the system. Instead of trying to provide only one comprehensive global explanation, we foster the coexistence of several narratives. Each narrative focuses on different aspects of the system, further abstracting the level of details, in order for a user to pick a different level of description of the system behaviour depending on the task.

The identification of the different levels of explanation, as well as of the main elements characterising a level and its mapping, is the main contribution of this paper. In particular, in Section 4 we illustrated a formal process for building narratives at multiple levels, showing a concrete implementation based on Jason. In principle, the process can be adapted to other BDI-oriented programming approaches, eventually refining the set of events and mappings. In practice, this adaptation may involve several complexities, so that the applicability of our approach to different agent-oriented programming frameworks is an interesting problem that will be further investigated in our future work.

Another aspect that needs to be further investigated in future work is the generation of the explanation at the domain level, as discussed in Sect. 4.4. This feature is missing in the current prototype. On the one hand, the adoption of domain-driven methodologies such as domain-driven design allows for effectively identifying domain events as part of the analysis and development process, along with user stories and acceptance scenarios. Such domain events are exactly the set of events that can be used to properly configure the explainer. On the other hand, our approach further requires to go deeper in knowledge crunching, by explicitly identifying relevant causes and effects relationships among domain events, and mappings between domain events and patterns of events at the design level. System stories [25] will be a valuable reference to consider for this extension.

Following the state of the art in explainability for BDI agents, our approach builds on the concept of agents narrating their behaviour in natural language, as a way to improve the level of understanding of the produced explanations for different users. We additionally provide a dedicated user interface to interact with the narrative, which, despite being a prototype, is already supporting the navigation of the agent timeline of events in the system.

Furthermore, we keep the alignment with the Whyline approach [17] as a way to interrogate the system for the explanation of a specific event happening. Notably, though, the current system setup does not support the possibility of asking why the agent *did not* do something but only asks for explanations of what has been done.

We intend that our framework is customisable with regard to the complexity of the definition of *explanation* between events. In this paper, we adopt a simple, yet conceptually sound, notion of explanation, linking events using the explicit knowledge about cause-and-effect relationships, as defined by the (operational) semantics of the level. A limitation of this simple approach concerns the explanation of failures. Our prototype returns straightforward answers concerning *what* led to a failure and not necessarily *why* the failure happened and how to prevent it. Given the importance that these kinds of explanations have for users to consider

the explanations useful [46, 47], future works might combine more complex approaches, for instance exploiting causal reasoning [48] to identify and understand cause-and-effect relationships among events that could be unknown a priori.

We have yet to test this tool with users, in a study that could help prove whether the approach actually serves the purpose of helping users perform specific tasks. Several other works suggest already that an explanation in terms of BDI abstractions is generally understandable as humans explain their behaviour with these concepts [49]. It would still be interesting to see how users leverage the multi-level capabilities of the tool to navigate and understand the behaviour of agents since this has not been tested yet, to the best of our knowledge, in other user-evaluated settings of BDI explainers. The design of the evaluation settings and analysis of results will be one of the objectives of future works.

In the future, we are also interested in further expanding to the multi-level explainability concept. Namely, in this initial exploration of the multi-level approach, we are limited to the agent dimension of MAS and thus we have explanations for individual agents. In the Multi-Agent-Oriented Programming community (and in the JaCaMo platform that we used for our prototype), a MAS is generally composed of other dimensions, namely the environment, organisation, and interaction dimensions [50]. The absence of this extension introduces some limitations to the current state of our tool. First, without incorporating the environment dimension, the agent is not able to explain the cause of events that come from the environment (e.g., failures in external actions). Second, the current tool requires users to manually align the narratives of multiple agents to figure out how they influence each other. This could be improved by extending the tool to include the interaction dimension, enabling explanations of communication between agents, and the organization dimension by explaining its structural, functional, and normative specifications. Given the relevance of the other dimensions, they should be considered for the development of a multi-level approach that could then be seen as multi-dimensional as well. As it goes for the agent dimension, and also for the other dimensions of MAS, we might apply the same levels of abstraction that we propose—implementation, design, and domain. Revisiting Fig. 2, we would then obtain a much richer navigation space for the users to explore the behaviour of the system (Fig. 15), i.e., considering both relevant dimensions and the level of abstraction. From this point of view, selecting the appropriate perspective for analysing the behaviour of the MAS, could potentially offer benefits in improving the understanding of the overall behaviour of the system. We plan to work on this concept and expand the framework that we presented to include this multi-dimensional view in the future.

Finally, we have proposed this multi-level explainability framework for BDI agents. In future works it would be interesting to expand the idea to non-BDI agents as well, trying to understand what shape the different abstraction levels could have. We speculate that given the innate nature of the BDI model to explain the behaviour of autonomous entities with mentalistic notions, it would be interesting to keep it as the main design level, regardless of the underlying execution platform to act as a *cognitive neck* [51] to hide the implementation details and support the explanation at the domain level.

7 Conclusion

Given the predominant role that explainability is taking as a feature of intelligent systems, in this paper we built on top of state of the art in explainability for BDI agents presenting a

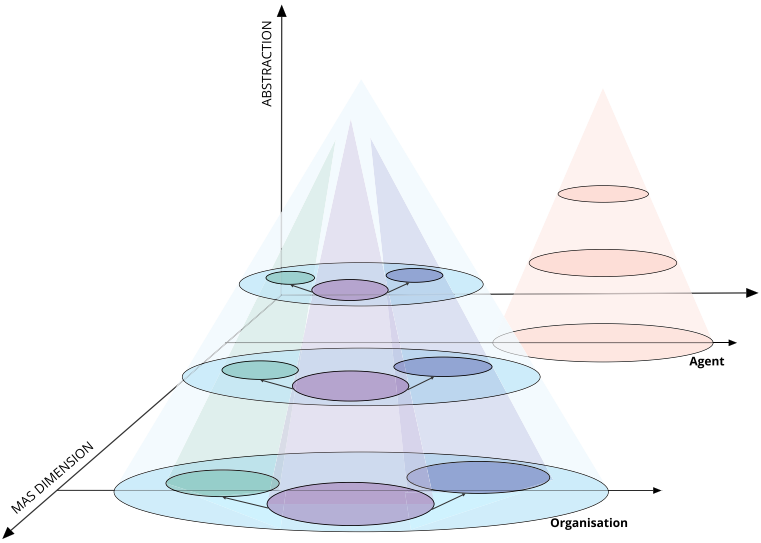


Fig. 15 Multiple dimensions and levels of abstraction in a multi-agent system

new approach to the generation from system logs of narratives that can be read by users to understand the behaviour of the system.

We put forward a new idea of *multi-level* explainability, in which from the same system we are able to generate increasingly abstract narratives, mapping the sets of events that are being logged in one level to the one above to achieve a higher level of abstraction of the explanation. Users who may have questions about the behaviour of the system can then find the events in the narrative that they feel are misaligned with the expected behaviour and ask for a more detailed explanation of why they happened.

We argue that users may benefit from the availability of multiple levels that are tailored to different tasks and skills. Developers may require detailed insights for debugging, designers may need to verify its functional properties, or end-users who simply need to understand whether the system is behaving as expected (and eventually why it is not).

We identified three possible levels. At the low level, we have the *Implementation Level*, which concerns low-level details about the implementing technology of the MAS (i.e. the specific language or platform). As an intermediate level, we have the *Design Level* which concerns the high-level architectural components of the system. Finally, at the highest level, we have the *Domain Level* which concerns the system specifications from the point of view of end-users and domain experts and should shield the explanation from any technical detail about how the system is implemented while relating it to the requirement specifications.

We have shown the process that we undertook to identify the event sets for our prototype tool built upon the Jason agent programming language and the JaCaMo platform. We detailed how the logs were extracted from the platform and we have shown how our Web-based interface allows users to read narratives and answer potential inquiries on the behaviour of the system, comparing it with the raw processing of logs or system exceptions to highlight the potential benefits of the approach.

Finally, we discussed the current benefits and limitations of the approach, highlighting open directions that we plan to address in future works. The contributions of this work, i.e., the identification of different levels for explaining the system behaviour tailored to different

users and the process to build explanations at multiple levels with a prototype implementation, could serve as a reference and suggestions for potential research directions to advance in the field of explainability for BDI agents and MAS in general.

Acknowledgements Partially funded by ANR-FAPESP NAIMAN project (ANR-22-CE23-0018-01) and by CAPES, the Brazilian Agency for Higher Education, under the project PrInt CAPES-UFSC "Automation 4.0".

Funding Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement.

References

1. Xu, F., Uszkoreit, H., Du, Y., Fan, W., Zhao, D., & Zhu, J. (2019). Explainable AI: A brief survey on history, research areas, approaches and challenges. In J. Tang, M.-Y. Kan, D. Zhao, S. Li, & H. Zan (Eds.), *Natural language processing and Chinese computing* (pp. 563–574). Springer. https://doi.org/10.1007/978-3-030-32236-6_51
2. Barredo Arrieta, A., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion, 58*, 82–115. https://doi.org/10.1016/j.inffus.2019.12.012
3. Malhi, A., Knapic, S., & Främling, K. (2020). Explainable agents for less bias in human-agent decision making. In D. Calvaresi, A. Najjar, M. Winikoff, & K. Främling (Eds.), *Explainable, transparent autonomous agents and multi-agent systems* (pp. 129–146). Springer. https://doi.org/10.1007/978-3-030-51924-7_8
4. Bratman, M. (1987). *Intention, plans, and practical reason*. Harvard University Press. https://doi.org/10.2307/2185304
5. Harbers, M., Bosch, K., & Meyer, J.-J. (2010). Design and evaluation of explainable BDI agents. In *2010 IEEE/WIC/ACM international conference on web intelligence and intelligent agent technology* (Vol. 2, pp. 125–132). https://doi.org/10.1109/WI-IAT.2010.115
6. Broekens, J., Harbers, M., Hindriks, K., Bosch, K., Jonker, C., & Meyer, J.-J. (2010). Do you get it? user-evaluated explainable BDI agents. In *Multiagent system technologies: 8th German conference, MATES 2010, Leipzig, Germany, September 27–29, 2010. Proceedings 8* (Vol. 6251, pp. 28–39). Springer. https://doi.org/10.1007/978-3-642-16178-0_5
7. Hindriks, K. V. (2012). Debugging is explaining. In I. Rahwan, W. Wobcke, S. Sen, & T. Sugawara (Eds.), *PRIMA 2012: Principles and practice of multi-agent systems* (pp. 31–45). Springer. https://doi.org/10.1007/978-3-642-32729-2_3
8. Winikoff, M. (2017). Debugging agent programs with why? Questions. In *Proceedings of the 16th conference on autonomous agents and multiagent systems. AAMAS '17* (pp. 251–259). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. https://doi.org/10.5555/3091125.3091166
9. Winikoff, M., Sidorenko, G., Dignum, V., & Dignum, F. (2021). Why bad coffee? Explaining BDI agent behaviour with valuings. *Artificial Intelligence, 300*, 103554. https://doi.org/10.1016/j.artint.2021.103554
10. Bordini, R., Hübner, J., & Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason* (Vol. 8). Wiley. https://doi.org/10.1002/9780470061848
11. Langley, P., Meadows, B., Sridharan, M., & Choi, D. (2017). Explainable Agency for Intelligent Autonomous Systems. In S. Singh & S. Markovitch (Eds.), *Procees of the thirty-first AAAI conference on artificial intelligence, February 4–9, 2017* (pp. 4762–4764). AAAI Press. https://doi.org/10.1609/aaai.v31i2.19108

12. Anjomshoae, S., Najjar, A., Calvaresi, D., & Främling, K. (2019). Explainable Agents and Robots: Results from a Systematic Literature Review. In *Proceedings of the 18th international conference on autonomous agents and multiagent systems. AAMAS '19* (pp. 1078–1088). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. https://doi.org/10.5555/3306127.3331806
13. Rao, A. S., Georgeff, M. P., et al. (1995). BDI agents: From theory to practice. *ICMAS, 95*, 312–319.
14. Bordini, R. H., El Fallah Seghrouchni, A., Hindriks, K., Logan, B., & Ricci, A. (2020). Agent programming in the cognitive era. *Autonomous Agents and Multi-Agent Systems*. https://doi.org/10.1007/s10458-020-09453-y
15. Pothier, G., & Tanter, E. (2009). Back to the future: Omniscient debugging. *IEEE Software, 26*(6), 78–85. https://doi.org/10.1109/MS.2009.169
16. Koeman, V. J., Hindriks, K. V., & Jonker, C. M. (2017). Omniscient debugging for cognitive agent programs. In *Proceedings of the twenty-sixth international joint conference on artificial intelligence, IJCAI-17* (pp. 265–272). https://doi.org/10.24963/ijcai.2017/38
17. Ko, A. J., & Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on software engineering. ICSE '08* (pp. 301–310). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1368088.1368130
18. Dennis, L. A., & Oren, N. (2021). Explaining BDI agent behaviour through dialogue. In *Proceedings of the 20th international conference on autonomous agents and MultiAgent systems. AAMAS'21* (pp. 429–437). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. https://doi.org/10.5555/3463952.3464007
19. Lam, D. N., & Barber, K. S. (2005). Comprehending agent software. In *Proceedings of the fourth international joint conference on autonomous agents and multiagent systems. AAMAS'05* (pp. 586–593). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1082473.1082562
20. Lam, D. N., & Barber, K. S. (2005). Debugging agent behavior in an implemented agent system. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *Programming multi-agent systems* (pp. 104–125). Springer. https://doi.org/10.1007/978-3-540-32260-3_6
21. Bosse, T., Lam, D. N., & Barber, K. S. (2008). Tools for analyzing intelligent agent systems. *Web Intelligence and Agent Systems: An International Journal, 6*(4), 355–371. https://doi.org/10.3233/wia-2008-0145
22. Ahlbrecht, T. (2023). An algorithmic debugging approach for belief-desire-intention agents. *Annals of Mathematics and Artificial Intelligence*. https://doi.org/10.1007/s10472-023-09843-4
23. Cockburn, A. (2000). *Writing effective use cases* (1st ed.). Addison-Wesley Longman Publishing Co., Inc.
24. Carrera, Á., Iglesias, C. A., & Garijo, M. (2013). Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. *Information Systems Frontiers, 16*(2), 169–182. https://doi.org/10.1007/s10796-013-9438-5
25. Rodriguez, S., Thangarajah, J., & Winikoff, M. (2021). User and system stories: An agile approach for managing requirements in AOSE. In *Proceedings of the 20th international conference on autonomous agents and multiagent systems. AAMAS'21* (pp. 1064–1072). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. https://doi.org/10.5555/3463952.3464076
26. Rodriguez, S., Thangarajah, J., Winikoff, M., & Singh, D. (2022). Testing requirements via user and system stories in agent systems. In: P. Faliszewski, V. Mascardi, C. Pelachaud, & M. E. Taylor (Eds.) *21st international conference on autonomous agents and multiagent systems, AAMAS 2022* (pp. 1119–1127). International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), Auckland, New Zealand, May 9–13, 2022. https://doi.org/10.5555/3535850.3535975
27. Rodriguez, S., Thangarajah, J., & Winikoff, M. (2023). A behaviour-driven approach for testing requirements via user and system stories in agent systems. In N. Agmon, B. An, A. Ricci, & W. Yeoh (Eds.) *Proceedings of the 2023 international conference on autonomous agents and multiagent systems, AAMAS 2023, London, UK, 29 May 2023–2 June 2023* (pp. 1182–1190). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. https://doi.org/10.5555/3545946.3598761
28. Rodriguez, S., Thangarajah, J., & Davey, A. (2024). Design patterns for explainable agents (XAg). In M. Dastani, J.S. Sichman, N. Alechina, & V. Dignum (Eds.) Proceedings of the 23rd international conference on autonomous agents and multiagent systems, AAMAS 2024, Auckland, New Zealand, May 6–10, 2024 (pp. 1621–1629). ACM, Richland, SC. https://doi.org/10.5555/3635637.3663023
29. Winikoff, M. (2024). Towards engineering explainable autonomous systems. In D. Briola, R. C. Cardoso, & B. Logan (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. https://doi.org/10.1007/978-3-031-71152-7_9

30. Gregor, S., & Benbasat, I. (1999). Explanations from intelligent systems: Theoretical foundations and implications for practice. *MIS Q., 23*(4), 497–530. https://doi.org/10.2307/249487
31. Langer, M., Oster, D., Speith, T., Hermanns, H., Kästner, L., Schmidt, E., Sesing, A., & Baum, K. (2021). What do we want from explainable artificial intelligence (XAI)? A stakeholder perspective on XAI and a conceptual model guiding interdisciplinary XAI research. *Artificial Intelligence, 296*, 103473. https://doi.org/10.1016/j.artint.2021.103473
32. Poutakidis, D., Padgham, L., & Winikoff, M. (2002). Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the first international joint conference on autonomous agents and multiagent systems: Part 2. AAMAS'02* (pp. 960–967). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/544862.544966
33. Dastani, M., Brandsema, J., Dubel, A., & Meyer, J.-J.C. (2010). Debugging BDI-based multi-agent programs. In L. Braubach, J.-P. Briot, & J. Thangarajah (Eds.), *Programming multi-agent systems* (pp. 151–169). Berlin: Springer. https://doi.org/10.1007/978-3-642-14843-9_10
34. Ekinci, E. E., Tiryaki, A. M., Çetin, Ö., & Dikenelli, O. (2009). Goal-oriented agent testing revisited. In M. Luck & J. J. Gomez-Sanz (Eds.), *Agent-oriented software engineering IX* (pp. 173–186). Berlin: Springer. https://doi.org/10.1007/978-3-642-01338-6_13
35. Martin, R. C. (2003). *Agile software development: Principles, patterns, and practices*. Prentice Hall PTR.
36. Cohn, M. (2004). *User stories applied: for agile software development*. Addison Wesley Longman Publishing Co., Inc.
37. Newell, A. (1982). The knowledge level. *Artificial Intelligence, 18*(1), 87–127. https://doi.org/10.1016/0004-3702(82)90012-1
38. Newell, A. (1993). Reflections on the knowledge level. *Artificial Intelligence, 59*(1–2), 31–38.
39. Dennett, D. C. (1989). *The intentional stance*. MIT Press.
40. Jennings, N. R. (2000). On agent-based software engineering. *Artificial Intelligence, 117*(2), 277–296. https://doi.org/10.1016/S0004-3702(99)00107-1
41. Newell, A. (1990). *Unified theories of cognition*. Harvard University Press.
42. Padgham, L., & Winikoff, M. (2003). Prometheus: A methodology for developing intelligent agents. In F. Giunchiglia, J. Odell, & G. Weiß (Eds.), *Agent-oriented software engineering III* (pp. 174–185). Springer. https://doi.org/10.1007/3-540-36540-0_14
43. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems, 8*(3), 203–236. https://doi.org/10.1023/B:AGNT.0000018806.20944.EF
44. Evertsz, R., Thangarajah, J., & Ly, T. (2019). *Practical modelling of dynamic decision making*. Springer. https://doi.org/10.1007/978-3-319-95195-9
45. Wooldridge, M. (2002). *Introduction to multiagent systems*. Wiley.
46. Miller, T. (2019). Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence, 267*, 1–38. https://doi.org/10.1016/J.ARTINT.2018.07.007
47. Lim, B. Y., Dey, A. K., & Avrahami, D. (2009). Why and why not explanations improve the intelligibility of context-aware intelligent systems. In In D. R. Olsen Jr., R. B. Arthur, K. Hinckley, M. R. Morris, S. E. Hudson,& S. Greenberg (Eds.), *Proceedings of the 27th international conference on human factors in computing systems, CHI 2009, Boston, MA, USA, April 4–9, 2009* (pp. 2119–2128). ACM. https://doi.org/10.1145/1518701.1519023
48. Pearl, J., & Mackenzie, D. (2018). *The book of why: The new science of cause and effect* (1st ed.). Basic Books Inc.
49. Malle, B. F. (2004). *How the mind explains behavior: Folk explanations, meaning, and social interaction*. The MIT Press. https://doi.org/10.7551/mitpress/3586.001.0001
50. Boissier, O., Bordini, R. H., Hübner, J. F., & Ricci, A. (2019). Dimensions in programming multi-agent systems. *The Knowledge Engineering Review, 34*, 2. https://doi.org/10.1017/S026988891800005X
51. Ricci, A., Mariani, S., Zambonelli, F., Burattini, S., & Castelfranchi, C. (2024). The cognitive hourglass: Agent abstractions in the large models era. In *Proceedings of the 23rd international conference on autonomous agents and multiagent systems* (pp. 2706–2711). https://doi.org/10.5555/3635637.3663262