Design Patterns for Explainable Agents (XAg)

Sebastian Rodriguez
RMIT University
Melbourne, Australia
sebastian.rodriguez@rmit.edu.au

John Thangarajah
RMIT University
Melbourne, Australia
john.thangarajah@rmit.edu.au

Andrew Davey
RMIT University
Melbourne, Australia
andrew.davey2@rmit.edu.au

ABSTRACT

The ability to explain the behaviour of the AI systems is a key aspect of building trust, especially for autonomous agent systems - how does one trust an agent whose behaviour can not be explained? In this work, we advocate the use of *design patterns* for developing explainable-by-design agents (XAg), to ensure explainability is an integral feature of agent systems rather than an "add-on" feature. We present TriQPAN (Trigger, Query, Process, Action and Notify), a design pattern for XAg. TriQPAN can be used to explain behaviours of any agent architecture and we show how this can be done to explain decisions such as why the agent chose to pursue a particular goal, why or why didn't the agent choose a particular plan to achieve a goal, and so on. We term these queries as *direct queries*. Our framework also supports *temporal correlation queries* such as asking a search and rescue drone, "which locations did you visit and why? ". We implemented TriQPAN in the SARL agent language, built-in to the goal reasoning engine, affording developers XAg with minimal overhead. The implementation will be made available for public use. We describe that implementation and apply it to two case studies illustrating the explanations produced, in practice.

KEYWORDS

AOSE; Design Patterns; Explainable Agents; EMAS

ACM Reference Format:
Sebastian Rodriguez, John Thangarajah, and Andrew Davey. 2024. Design Patterns for Explainable Agents (XAg). In *Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), Auckland, New Zealand, May 6 – 10, 2024*, IFAAMAS, 9 pages.

1 INTRODUCTION

The likes of ChatGPT has propelled artificial intelligence techniques into a new era, where more and more industries are seeking to leverage the power of AI to automate and optimise business processes and productivity. This increased interest has also raised the important question of whether these AI systems can be trusted. The ability to explain the behaviour of AI systems is a key aspect of building trust [14]. This is particularly important for multi-agent systems (MAS) that typically perform tasks on behalf of humans.

In this work, we advocate the use of *design patterns* for the purpose of developing eXplainable-by-design Agents (XAg).

In traditional Software Engineering, the concept of *Design Patterns* [8] have been successfully used for decades to improve a vast number of quality attributes of the software products. These quality attributes (sometimes referred as "-ity" attributes) include modularity, usability, interoperability, and so on. We argue that *explainability* should be included in this list for AI systems. The design pattern also ensures that explainability is an integral feature when developing multi-agent systems, rather than an "add-on".

Despite the call for the use of design patterns in MAS over a decade ago [3] to increase the broader acceptance of the technology, there has been little work on the use of design patterns in practical MAS. A notable exception is the work of Dastani and Testerink [5] that provided design patterns for agent-oriented programming providing templates for realising some agent-oriented concepts and abstractions in an object-oriented technology. There has also been recent work by Washizaki et al. [28] exploring design patterns for machine learning systems, also highlighting the importance of this approach in gaining wider acceptance by the broader software engineering community.

In this work, we propose TriQPAN (Trigger, Query, Process, Action and Notify), a design pattern, which when implemented can be used to explain the behaviours such as why the agent chose to pursue a particular goal, why or why didn't the agent choose a particular plan to achieve a goal, why a particular state is true and so on. The underlying system needs to capture and store events related to a TriQPAN design process and the event store is queried to explain behaviours and outcomes.

Recently, Winikoff et al. [30–32] presented a formal framework for constructing explanations of the behaviour of BDI agent systems [17]. Their work builds on, formalises and extends the work of Habers et al. [10, 11], and utilise the goal-plan structures typically found in BDI agents. Whilst our design pattern TriQPAN could be used for any decision-making AI system, and we show how this can be done, we incorporate TriQPAN natively into the goal-reasoning engine of the SARL agent programming language [18]. Underlying SARL is an event driven architecture. We incorporate a state-of-the-art event stream database - EventStoreDB [1] to log and query the event streams, which can be done in real-time.

We facilitate three types of explanation queries: (i) *direct queries* related to goals, plans and beliefs. (e.g. why did you decide to get the coffee from the shop?); (ii) *temporal correlation queries* that relate a sequence of decisions (e.g. Which series of locations did you visit to get the coffee?); and (iii) *continuous queries* that monitors for the query result continuously and returns values (e.g. what times of the day is the kitchen coffee replenished?). To the best of our knowledge previous work was only able to deal with direct queries. The key differentiator to previous work is that our approach of using an event store to manage the stream of events related to the design pattern. This affords us the luxury of rich queries that goes beyond reasoning constrained to the goal-plan tree structures.

[1] https://www.eventstore.com

Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), N. Alechina, V. Dignum, M. Dastani, J.S. Sichman (eds.), May 6 – 10, 2024, Auckland, New Zealand. © 2024 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

In summary, our contributions are: (i) we present a design pattern - TriQPAN , for developing explainable MAS (Section 3); (ii) implemented TriQPAN into the SARL agent programming language, natively, together with EventStoreDB. We describe the implementation and will make available the tools for public use (Section 4); and (iii) present practical experiments to illustrate the use of TriQPAN in practice with two case studies - the first, an abstract example of an agent getting coffee as used in [31], and the second, a search and rescue drone example as used in [22] (Section 5).

2 BACKGROUND

We first present some preliminaries on design patterns, goal-plan agents and our implementation platforms SARL and EventStoreDB. We also briefly describe the related work [31] and [10] on XAg.

2.1 Design Patterns

Design patterns [9] in software engineering are time-tested solutions to recurring design challenges, tracing their roots back to the early days of computer programming. These patterns provide solutions to recurring design problems, ensuring that developers don't have to reinvent the wheel with each new project. Instead, they can leverage proven strategies for specific design challenges, promoting code reusability and maintainability. A pattern is a common solution to a common problem in a given context [13].

Benefits of design patterns include reusability, improved communication, flexibility, and efficiency. However, their application must be context-aware to avoid unnecessary complexity. We present our design pattern for XAg in the context of multi-agent systems.

2.2 Goal-plan Agents

Although the XAg approach we present applies to any agent architecture, our native implementation in SARL is for goal-plan agents. That is, an agent that has goals to achieve and plans that achieve these goals. Goal-plan agents are also the context of the aforementioned previous work [10, 31].

A goal is typically a state of the world the agent wants to achieve, and plans are recipes for achieving them. A plan has a context condition that determines the applicability of the plan to achieve the goal in a particular situation and a body that may have steps that are a combination of actions (considered atomic) and sub-goals. These steps may be unordered or ordered sequentially. Each subgoal in turn would have its own set of plans that can be utilised to achieve them and so on. This goal-plan relationship leads to a natural tree-like structure, commonly referred to as a goal-plan tree (GPT) [25–27]. These GPTs are a general abstraction of a wide range of BDI agent [17] platforms (e.g. JACK [29], JadeX [16], Jason [2]) and we have recently extended SARL [18] to also include this goal-plan architecture.

Figure 1, illustrates the goal-plan structure for the "get coffee" example from [31]. We also use their example to also illustrate some aspects of the use of TriQPAN in the sections ahead. A succinct version of the scenario is as follows: An academic visitor requires coffee. There are a number of ways to get coffee. There is a coffee-pod machine in the host academic's office which serves good coffee and can be used freely if she is in her office. There is a staff kitchenette that has a machine that serves coffee-like substance, which is

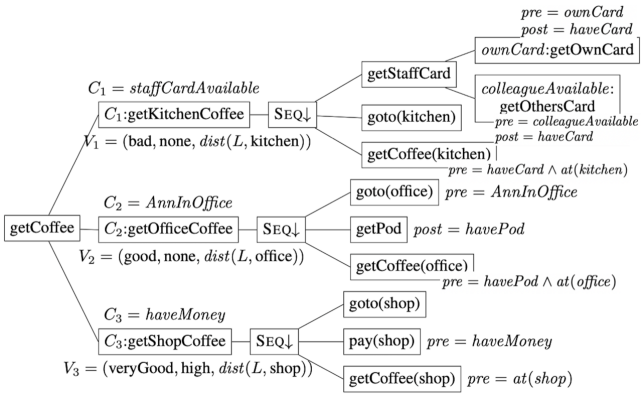

Figure 1: Example of "Get Coffee" as presented in [31]. getCoffee is the top-level goal, plans are written as C:N, where C is the condition and N the plan name, V_i are value effect annotations of the form (quality, cost, distance), where dist(L1,L2) is the distance between locations L1 and L2; and pre and post indicates the conditions that must be true for the plan or action to begin and what is true after execution, respectively.

not as nice but it is also free. The best coffee however is at a nearby coffee shop which can be purchased at a cost. A key preference is coffee over coffee-like substances. Less-important preferences are to save money, and to use the nearest coffee source. These give rise to the three relevant quality attributes (in order): quality (coffee preferred to coffee-like), money (free preferred to expensive), and location (smallest distance from starting location).

Winikoff et al. also introduce the notion of "valuings" to a typical goal-plan tree which indicates the positive or negative affect of a plan (or action) and its outcome. For example in Figure 1, the valuings of the getKitchenCoffee plan are - coffee quality is bad, cost is none, and the distance between locations. Their experiments show that using these valuings enable better explanations.

2.3 SARL & EventStoreDB

SARL is a general-purpose agent-oriented programming language designed for building intelligent multi agent systems [18]. Developed as an open-source project, SARL aims to simplify the creation of MAS by providing a high-level language with native constructs for modelling and implementing agent behaviours.

SARL agents are characterized by their autonomous, event-driven behavior responding to changes in their environment, and more recently goal-oriented behaviours [20]. Its interoperability with the Java ecosystem has further expanded its reach, allowing seamless integration with established software libraries and frameworks. Due to its generic and highly extensible architecture, SARL is able to integrate new concepts and features quickly. This quality coupled with its features, has seen it adopted by a number of academic and industrial institutions to develop a wide range of applications [1, 15, 19, 20, 24]. The underlying event-driven architecture of SARL makes it highly amenable towards XAg as storing and querying the events enables explainability. We store and query event streams in SARL by incorporating the state-of-the-art EventStoreDB.

EventStoreDB (https://www.eventstore.com/) is a specialized database system optimized for event sourcing [7] — a software architectural pattern where changes to the system state are stored as a sequence of events, rather than merely storing the current state itself. By capturing every single change as a distinct event, EventStoreDB allows systems to reconstruct their state at any point in time, thereby providing a robust mechanism for versioning, auditing, and in our use case, explainability. We utilise some of its rich features such as temporal queries and projections in order to derive explanations as we describe in Section 4.2.

2.4 Related Work on XAg

Although there has been a huge emphasis on the need for explainable AI there is not much work in MAS research on explainable agents. Notable exceptions are the aforementioned recent work of Winikoff et al. [31] and much earlier body of work by Habers et al. [10, 11]. All of these works utilise GPTs as a base for providing explanations as the GPT allows traceability. For example, explaining why an action is performed could be done by tracing the tree. In the simplest form it could be - action 'a' was performed by plan 'p' as part of achieving goal 'g'.

Habers et al. [11] requires the agents to store any decisions that may require explanation via an explicit logging mechanism. For example, when the agent adopts a goal G at time t, it logs that it adopted goal G at t. In the 2APL [4] programming language that they use, it would be written as follows:
Monitor <- true | [adopt(Check(X)); UpdateLog(Check(X),t)].
This *explanation log* can store beliefs, goals, actions etc. The decision what to log and what not should depend on the information that is desired in an explanation. Whilst they provide templates for some case studies their approach was not a general formalism.

The approach of Winikoff et al. [31] strictly generalises the work of Habers et al. and is able to do more. In particular, they introduce "valuings" as mentioned above. Their experiments show that valuings provide for better explanations. An important contribution of [31] is the mechanisms for generating explanations in a more human-friendly natural language than say a simple trace of a GPT. They provide formal definitions and detailed algorithms.

We note that our design pattern TriqPAN is complementary and generalises all of the previous work. That is, TriqPAN allows all of the explanations provided in previous work, which we term *direct queries*, and go beyond by allowing *temporal correlation queries* as we describe in Section 4. We use the event-driven architecture of SARL combined with EventStoreDB to *automatically* log all the events in such a way they can be queried to provide rich explanations, beyond simply tracing the goal-plan tree structures.

We integrate TriQPAN into the SARL goal-reasoning engine to automatically capture all the relevant events to provide *in-built explainability* for goal-oriented behaviours. This allows the use of SARL with built-in XAg that requires little to no additional overhead from the developers. We also show how TriqPAN can be implemented in any agent architecture, to explain any type of (ad-hoc) behaviours.

To the best of our knowledge, the work of Winikoff et al. was not implemented in an agent programming language but rather prototypes in Haskell and Python to evaluate the formalisms and algorithms presented. In contrast, we fully implement our design pattern based approach in SARL, and will make this XAg version of SARL available for use via GitHub. This can then be used to develop agent systems with built-in explainability and also expand upon as required. Note however that currently our explanations do not provide rich natural language expressions, as it is not the intended purpose of our work. Instead, in future work, we aim to utilise and implement the algorithms provided by Winikoff et al. for this purpose.

3 TRIQPAN - A DESIGN PATTERN FOR XAG

The TriQPAN (Trigger, Query, Process, Action and Notify) pattern is designed to create explainable agent processes that can be recorded to explain the agent's reasoning and decision processes. As with any design pattern, TriQPAN needs to be adapted to the specific process being implemented. In a typical MAS, every decision that an agent makes, for example, which goal to pursue, which plan to execute to achieve a goal, and so on, entails a process that is composed of a sequence of clearly identifiable steps - *trigger, query, process, action and notify* as follows.

Once the decision process has been *triggered* (e.g. by a perception), it will *query* its state or known information (e.g., its belief sets), compute or *process* this information to select the *actions* to perform, and finally, *notify* of its actions and completion to other modules (i.e., building blocks for the agent's architecture). This observation and the steps are at the core of the TriQPAN pattern.

A graphical notation of the TriQPAN pattern is shown in Figure 2. Each icon is annotated with the TriQPAN steps it represents and the arrows between them indicate control flow.

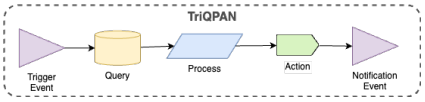

Figure 2: TriQPAN design pattern

Triggers are instances of events that begin a TriQPAN process. Perceptions are the most typical triggers of behaviours for most agent types, though they can take different forms depending on the specific architecture used. If the agent is using a goal-oriented architecture, events such as *Goal Adopted, Goal Activated* or *Belief Updated*, are candidates triggers.

Query step retrieves information from the agent's mental state. This can be as simple as data structures (e.g., variables) or as complex as querying an ontology knowledge base.

Process step queries the data and the trigger to select the appropriate actions to take (if any).

Action step executes one or more actions in response to the trigger. Actions in this pattern should be interpreted as operations the agent is able to do in a broader sense. They can be *internal* (e.g. update beliefs) or *external* (e.g., write to a file, move towards a location).

Notify ends the TriQPAN process and is twofold. First, all actions must notify of any effective change of state (e.g., belief updates). Second, the TriQPAN should fire an event that informs of all the components used. We term this event the

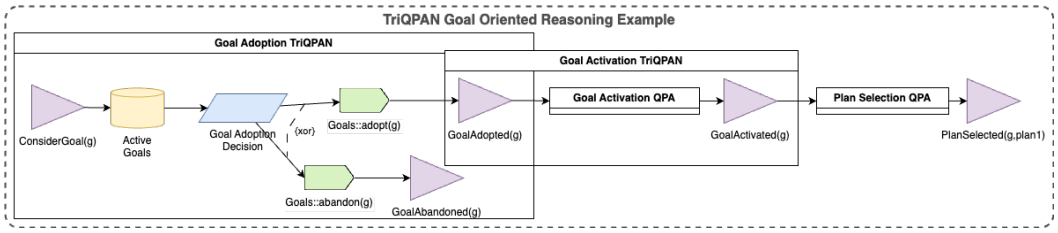

Figure 3: Goal Reasoning processing using the TriQPAN patterns

XAgentProcess event . Note that the notifications of one TriQPAN can be the trigger of another TriQPAN , for chained decision processes (See Figure 3).

The XAgentProcess event is a central element for explainability in our approach. It is captured and stored to enable interactive queries. We describe this in detail in Section 4 ahead.

TriQPAN adoption considerations. Like any design pattern, TriQPAN must be adapted to the specific agent process it is applied to. We highlight some important considerations when adapting it:

- Each TriQPAN must fire a XAgentProcess event that contains all relevant information about the TriQPAN components (i.e., trigger, queries and outputs, actions applied).
- Every action should fire an event that notifies other modules of effective state changes. For instance, when updating a belief a *Belief Updated* event should be fired. If actions do not directly affect the agent's state (e.g. actions that affect the external environment only), changes will be captured later by the agent as perceptions as the environment changes.
- A TriQPAN must not contain *Action-Query* loops. Any process required as a response to an action *a* should be modelled as a separate TriQPAN using the notification of *a*'s TriQPAN as the trigger.
- A TriQPAN must be *stateless*. That is, it can only depend on the information contained in the trigger and the queries made. This stateless feature also means that, given the same trigger and the same query results, the process must apply the same actions ensuring reproducibility. Furthermore, if the XAgentProcess event is correctly constructed, every TriQPAN can be reproduced and verified against its XAgentProcess event . This feature is a result of TriQPAN 's foundations and inspiration on well-established event-driven patterns such as Event Sourcing [7] and CQRS [6].

The TriQPAN pattern can be used in any agent architecture to model and explain key decision processes. While TriQPAN can be used in ad-hoc agent decision processes, its main benefits appear when the framework used for reasoning natively integrates the TriQPAN pattern. E.g., a goal-oriented reasoning engine built using TriQPAN would enable explainability without any (or minimal) developer overhead. As described ahead in section 4.2, we have implemented such an engine in SARL. This allows developers to focus on designing a goal oriented solution obtaining XAg *for free*.

Here we briefly describe the principle underlying the creation of a goal reasoning engine based on TriQPAN patterns. The fundamental idea is to ensure that the goal reasoning engine has a set of TriQPAN processes such that the process notifications of one TriQPAN are triggers for the next required TriQPAN process.

Consider the diagram in Figure 3. It represents three processes of the goal engine. First, the *Goal Adoption* TriQPAN is triggered by a request to *ConsiderGoal(g)* for adoption. This trigger could have originated from the request of another agent or a reflex following a perception. First, it queries all the currently *Active Goals* of the agent to complete then *Goal Adoption Decision* process. The ensuing flow is annotated with an *xor* constraint to denote that the decision can be to execute one of the actions - *adopt* goal or *abandon* goal *g*. Each of these actions, then fires the corresponding notifications of *GoalAdopted(g)* or *GoalAbandoned(g)*.

The *GoalAdopted(g)* event triggers the TriQPAN process of *Goal Activation*, hence forming a chain of TriqPANs where the notification of one, leads to the trigger of another.

In turn, the *Goal Activation* TriQPAN may trigger a *GoalActivated(g)* notification after the *Query-Process-Action (QPA)* steps of the TriQPAN . This *GoalActivated(g)* notification becomes the trigger of the *Plan Selection* TriQPAN . (Note that the notation allows to collapse these steps to focus on the sequence of events. To avoid cluttering, we choose to show the *QPA* steps container with the name of the TriQPAN process, and infer the trigger and notifications from the control flow arrows.)

4 EXPLAINING AGENT BEHAVIOURS

In this section, we present how TriQPAN pattern based architectures enable XAg. We use our implementation of TriQPAN to illustrate how this can be done. We also use this section to highlight what our implementation in SARL supports in terms of XAg.

4.1 Explanation query types

Notification events observed from an XAg system that implements TriQPAN contains rich information about the decision processes and its outcomes. They enable us to query the system behaviour regarding a large number of situations. In our current approach we support three types of queries with more planned for the future: (i) direct queries; (ii) temporally correlated queries; and (iii) continuous queries. Table 1 presents a summary of the direct and temporally correlated query types as implemented in this work. Continuous queries are essentially any explanation query (of the other types) where the agent is required to continuously provide answers on.

The table shows the questions available for each concept relevant to the query type, an example for each case with the actual query and the natural language description of it in italics below. We also note the level of support that our implementation in SARL

Table 1: Query types overview

	Concept	Question	SARL	Example
Direct	Goal (Achievement, Perform and Maintenance)	why(Goal, state, time)	Provided	why(GetCoffee, active, t10) *"Why did you decide to get a coffee at t10?"*
		why_not(Goal, state, time)	Partial	why_not(GetCoffee, dropped, t10) *"Why did you stop trying to get a coffee at t10?"*
		why_pref(Goal_1, Goal_2,time)	Planned	why(GetCoffee, GetTea, t10) *"Why did you prefer a coffee over a tea at t10?"*
	Plan	why(Plan, Goal, time)	Provided	why(GetKitchenCoffee, GetCoffee, t10) *"Why did you get a coffee from the kitchen?"*
		why_not(Plan, Goal, time)	Provided	why_not(GetShopCoffee, GetCoffee,t10) *"Why didn't you get a coffee from the shop?"*
		why_pref(Plan_1,Plan_2, time)	Provided	why_pref(GetKitchenCoffee, GetShopCoffee,t10) *"Why did you prefer the kitchen coffee over shop coffee at t10?"*
	Beliefs	why(BeliefSet, Belief, value, time)	Provided	why(OfficeBeliefs, coffee.quality, BAD, t10) *"Why did you get BAD coffee"? (aka "Why bad coffee?)"*
	Ad-hoc Process	Domain specific	Supported	
	Action Outcome	why(ActionOutcome, value, time)	Supported	why(FileWritten, paper.tex, t10) *"Why was the file "paper.tex" written at t10?"*
Temporal Correlation	Generic	how_many((Notification, matcher?)+, time_window, time_frame?)	Supported	how_many(GoalActivated(StartWork), GoalActivated(GetCoffee),10 minutes) *"How many times did you get coffee within 10 mins of starting work?*
		what_sequence(Notification, attribute?, time_frame?)	Provided	what_sequence(LocationUpdated, event.location) *"What was the sequence of changes to your location?"*
		is_it_always(Trigger, Notication, tolerance)	Provided	is_it_always(GoalActivated(StartWork), GoalActivated(GetCoffee), 10 minutes) *"Is it always the case that you get a coffee within 10 minutes of starting work?"*
		is_it_never(Trigger, Notification, tolerance)	Provided	is_it_never(TimeUpdated(3pm), GoalActivated(GetCoffee)) *"Is it never the case that you get coffee after 3pm?"*
	Goals	how_many(Goal+,window, time_frame?)	Supported	how_many(GetCoffee, StartWork, within 10 minutes) *"How many times do you get a coffee after starting work?"*
	Beliefs	how_many(Belief+, window, time_frame?)	Supported	how_many(location==KITCHEN, coffee==true) *"How many times do you have a coffee when at the kitchen?"*
		what_sequence(BelSet, Belief)	Provided	what_sequence(OfficeBeliefs, location) *"What is the sequence of locations you went through?"*

provides out-of-the-box for each case. The SARL support is categorized as *Provided:* the feature is fully implemented; *Partial:* not all functionalities are fully implemented at this stage, but implementation is planned; *Planned:* the functionality is not implemented but planned; *Supported:* the mechanisms are provided for future development if required.

4.1.1 Direct queries. In a goal-oriented agent system we are interested in obtaining explanations related to three core agent concepts - *goals, plans* and *beliefs*.

Our goal engine in SARL is inspired by [12] where the life-cyle of a goal is defined in terms of states and operational semantics are defined for the transitions. In our approach, we can query the agent about any decision related the operations of goals, not just the goal activation. E.g., *why was Goal X dropped ?* or *why suspended?*. While in most of this paper we focus on *achievement* goals, the goal-engine also supports *maintenance* and *perform* goals.

For both goals and plans, we are able to ask: (i) *why* questions; (ii) *why_not* questions; and (iii) *why_pref* questions. (see Table 1). For beliefs, *why* questions are the only applicable query type.

Finally, Ad-hoc processes and action outcome notifications can be queried using the information able in the TriQPAN notifications.

4.1.2 Temporal Correlation queries. In many cases, we are not only interested only in decisions made at a particularly given time (e.g., *why did you choose Plan A over B at* t_x*?*), that is, direct queries, but also explanations on sequences of decisions. For example, *which locations did you visit to get coffee?*). This requires processing the event stream log to extract the information needed.

We define a number of different types of temporally correlated queries: (i) *how_many* which is a count of some occurrence; (ii) *is_it_always* and (iii) *is_it_never* to query whether it is always the case or not case of an occurrence, respectively; and (iv) *what_sequence* to identify the trace of updates to a belief. See Table 1 for examples and concepts related to each type. Finally, we support questions targeting the TriQPAN notifications independently of any agent architecture, labelled as *Generic*. Using these queries, developers are not restricted to the set of questions proposed in this work and are able to create new ones.

4.1.3 Continuous explanations. In its very nature, events in the agent system are continuously streamed from the system to that event store. As a result of this live stream, humans are able to query the agents not only after the execution, but as continuously running queries that expand as the system executes. EventStoreDB support this type of *Continuous queries* using projections. Continuous queries combined with (temporal) correlation queries and explainability opens the door to new forms of live interactions between humans and XAgents; not only for explanations but also monitoring, cooperation, human veto and much more.

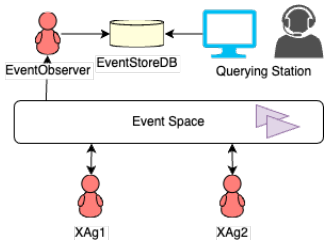

Figure 4: SARL TriQPAN architecture overview

4.2 SARL XAg

We selected SARL for implementing our XAg design pattern TriQPAN . While the examples below will use SARL to illustrate how to adapt the TriQPAN pattern to different agent designs, TriQPAN can be implemented and adopted to any existing agent framework.

4.2.1 Architecture. To enable explainability queries, we have integrated SARL platform with EventStoreDB, a state-of-the-art event stream database. A high level overview in show in Figure 4. An `EventObserver` stores the information produced by XAgents into the event database. Then a human operator can use specific queries using an event query language to answer *why* questions. Systems implementing the TriQPAN pattern will fire the events into SARL a `Space`. Spaces in SARL are communication support components able to transport events. These events can be observed by other agents to trigger their own behaviours[2].

4.2.2 Goal engine implementation. The engine is designed using a sequence of TriQPAN to chain core algorithms, such as goal selection, plan selection, and so on. Each one of these processes fires an XAgentProcess event to notify of the TriQPAN components used (as mentioned in Section 3).

We created a XAgentProcess event implementation in SARL to broadcast information associated with the TriQPAN pattern (see Figure 5). This event is populated by the TriQPAN process with all relevant information, such as process name; the implementation; trigger; queries; and actions selected. Additionally, it captures the criteria used to decide the actions taken.

To illustrate how this process is implemented, consider the *Plan Selection* process shown in Figure 5. The process is triggered by the `ReviewPlans` event. Then it will find all active goals (i.e., `intentions`) that do not have a plan attached (i.e., `unattachedGoals`). It will then find all applicable plan for that goal. This information is stored in the `queries` map of the XAgentProcess event .

As a selection criteria, it uses the *highest applicability* in the set of *applicablePlans* recorded in the `queries`. In our engine, the applicability is calculated for each plan based on the agent's beliefs. The winner is recorded as part of the actions taken.

4.2.3 Preference reasoning via valuings. We extended the SARL engine to support *Plan valuings* inspired by [30]. In addition, the goal engine also provides the criteria used for determining the applicability rating for each plan. The criteria allows to disambiguate on how the agent reached these ratings, an often overlooked feature. See section 5 for an example.

[2]Please refer to [18] for details on SARL communication mechanisms

```
behavior ApplicabilityRatingPlanSelection {
 uses Behaviors, IntentionStackManagement,
     PlanSelectionConstraints
 on ReviewPlans [isFromMe] {
  for (g : intentions) {
   val unattachedGoals = g.unattachedGoalIntentionFrames
   for (ug : unattachedGoals) {
    val xag = new XAgentProcess("PlanSelection", this.
     class, occurrence)
    val applicableLst = ug.goal.applicablePlans

    xag.queries.put("unattachedGoal", ug)
    xag.queries.put("applicablePlans", applicableLst)

    xag.criteria.put("winner", "highest applicability")
    val winner = applicableLst.maxBy[ap|ap.applicability]
    ug.attachPlan(winner.plan)
    xag.actions.put("attachPlan", winner)
    wake(xag) //fire internal event
   }
  }
  wake(new PlansUpdated)
 }
}
```

Figure 5: Rating based Plan Selection using TriQPAN

4.2.4 TriQPAN compliant mental states via beliefs. In goal-oriented agents, beliefs play a crucial role and changes to the beliefs should be notified via events. As this can be an overwhelming task for developers, the framework should provide native support for *belief set management*.

Our approach consist of identifying data structures with the `@Belief` annotation. The framework can automatically generate SARL capacities (and related skills) to query and update beliefs. Capacities and Skills are SARL's mechanisms to implement actions that are brought into scope by the useskeyword.[3] Update actions will automatically trigger notification events. For example the `setLocation` action will fire the `OfficeBeliefsLocationUpdated` event.

Belief update events include a list of `IntentionStackTrace-Elements`. This trace contains information regarding the agent process that updated the belief.

4.2.5 Ad-hoc processes support. For explanations outside of the above, the SARL engine offers full access to the TriQPAN API, allowing developers to replace / extend any of the processes and/or actions provided. This gives great flexibility to tailor for particular use cases supporting different agent architectures. For example, the agent system can integrate domain specific actions to modify their environment, and notify the outcomes of the following the TriQPAN process. For instance, "Why was file paper.tex modified?"(*why(FileWritten, paper.tex, t10)*).

4.3 Translating questions into event queries

In the EventStoreDB, events are stored in streams that can be queried using a query framework based on javascript. To answer *why* questions, we must first translate them into event queries. This allows us to harness the power of industry-grade systems to

[3]Please refer to SARL documentation for details.

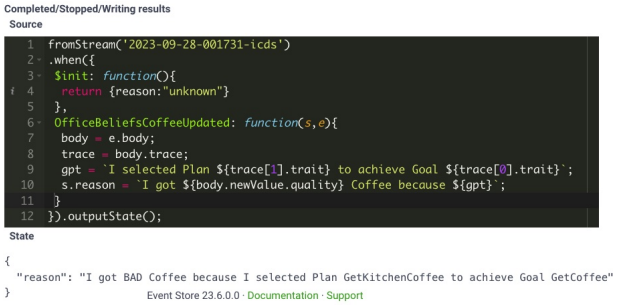

Figure 6: *Why bad coffee?* query and result

explain agent behaviours. It also opens the door to community collaboration and innovative contributions of shared modules.

In Figure 6, we show how to translate our example question, "why bad coffee?", into the event query language and the corresponding query result. To do this, we first need to select the stream of events of the application under consideration. Then, in the *when* function, we pass handlers for each type of event that hold information to answer the question at hand. The first handler (*$init*) allows to initialize the state that following handlers will receive as parameter *s*. The second handler is applied for every event *e* of type *OfficeBeliefsCoffeeUpdated* (i.e., notifications of updates to the *Coffee Belief*). The event's *body* format depends on the type of event. In our case, the OfficeBeliefsCoffeeUpdated event contains the updated value of the coffee belief in the *newValue* attribute and the trace of goals and plans that caused this belief update, in the *trace* attribute. Using this information we can construct the English explanation stored in the state's *s.reason* - "I got BAD coffee because I selected Plan GetKitchenCoffee to achieve Goal GetCoffee".

While our current translation is simplistic, we can plug-in an explanation engine as the one proposed by [30] to automatically create richer natural language explanations.

4.4 Explaining ad-hoc behaviours

Agent technology expands beyond goal oriented agents. In many applications, reactive agents (or specific reactive behaviours) are implemented. For instance, consider a `Drone` agent exploring an area to identify objects. When an object is detected it must decide whether to identify the newly detected object or its previous target. Additionally when the battery level becomes "LOW", it must return immediately to base to recharge. Such behaviours, could be simply implemented as a reactive decision based on the notification of *BatteryLevelUpdated*. In SARL, this behaviour could be implemented as shown in Figure 7. We are now able to answer *"Why are you returning to base?"*.

5 EXPERIMENTAL EVALUATION

We implemented two systems from published works, using the TriQPAN pattern presented in this work. First, we developed the *GetCoffee* goal-plan tree presented in [30]. Second, we adapted part of a *Search and Rescue* application presented in [22], Table 2 presents a sample of the questions and answers generated by the current implementation.

```
behavior Drone {
 uses DroneStateBeliefs // Belief set
 on ObjectDetected { // Trigger
  // TriQPAN process to decide on
  // object identification
 }
 on BatteryLevelUpdated [LOW == newValue] {
  val xag = new XAgentProcess("BatteryMonitor", Drone,
     occurrence)
  xag.queries.put("batteryLevel", batteryLevel)
  xag.criteria.put("destinationSelection", "Recharge on
     LOW Battery")
  destination = baseLocation
  xag.actions.put("destination", baseLocation)
  wake(xag)
 }
}
```

Figure 7: Basic XAgentProcess

We use the *GetCoffee* example to present goal, plan and belief explanations. First we find the explanation for *why bad coffee?* by querying when did the belief of *coffee* get updated and who made the change. The next obvious question, *why did the agent select the GetKitchenCoffee plan?*, is answered by presenting the ratings obtained by each plan. This same explanation can be presented for *why did not select GetOfficeCoffee plan?*.

We then explain *why GetSfaffCard goal was activated?*. And finally we present an example of *temporal correlation query* to explain *the path followed by the agent*.

As explained in previous sections, TriQPAN is not limited to goal-plan agents. The *Search and Rescue* application uses *ad-hoc* decision processes. The *Drone* agent was implemented using reactive behaviours that follow the TriQPAN pattern as shown in figure 7. This allows generating the explanation as show in table 2. This long running application benefits from *continuous queries*. The query runs continuously presenting the user with "live" updates of the decisions made by the *Drone*. For instance, we see the decision to return to base *because battery is low and must recharge*.

We notice that the system offers an (obscure) answer to *why GetKitchenCoffee plan was selected?*(i.e., *GetKitchenCoffee had 30% rating, while the others had 0%*). However, the framework is capable of generating more elaborate answers using the *valuings* and *criteria* information of the XAgentProcess event of the *Plan Selection* process. Figure 8 shows the details available for plans *GetKitchenCoffee* (left) and *GetOfficeCoffee* (right). For each plan we find: (i) the *valuings* that represent the information used when rating the plan; and (ii) the *criteria* used to generate the rating, including the formula to generate the final rating.

In the case of *GetKitchenCoffee* the criteria to generate the rating is $coffee.quality * cost * staffCardAvailable$. If we replace components using their criteria, we get $0.3 * 1 * 1 = 0.3$ In the case of *GetOfficeCoffee*, the value of *annInOffice* is `false` so this yields a rating of 0. If both plans were applicable (e.g, *annInOffice=true* and *staffCardAvailable=true*), *GetOfficeCoffee* would receive a rating of 0.5 since the agent rates *GOOD* coffee at 0.5 and *BAD* at 0.3. With this information, more elaborate answers can be easily generated enabling users to find answers to their own questions.

	Question	Output
		Coffee System
Direct Belief	why(OfficeBeliefs, coffee.quality, BAD)	I got BAD Coffee because I selected Plan GetKitchenCoffee to achieve Goal GetCoffee
Direct Plan	why(GetKitchenCoffee, GetCoffee) why_not(GetOfficeCoffee, GetCoffee)	Options for GetCoffee were: Plan GetKitchenCoffee with rating 30% Plan GetOfficeCoffee with rating 0% Plan GetShopCoffee with rating 0%"
Direct Goal	why(GetStaffCard)	Goal GetStaffCard was activated by GetKitchenCoffee Plan
Temporal	what_sequence(OfficeBeliefs, location)	Started in OFFICE then moved to KITCHEN
		Drone Search and Rescue
Ad-hoc Process (continuous query allowing monitoring)	where and why destination	"ObjectDetected(92,73) so Going to (92, 73) because I identify the closest object first", "ObjectDetected(25,47) so Going to (25, 47) because I identify the closest object first", "ObjectDetected(52,27) but Going to (25, 47) because I identify the closest object first", "ObjectDetected(51,13) but Going to (25, 47) because I identify the closest object first", "BatteryLevelUpdated(LOW) so Going to (0, 0) because Recharge on LOW Battery", "ObjectDetected(60,56) but Going to (0, 0) because Return to base has priority",

Table 2: Sample of questions and answer of systems implemented using TriQPAN

```
{
 "name": "GetKitchenCoffee",
 "rating": 0.3,
 "valuings": {
  "coffee.quality": "BAD",
  "cost": "NONE",
  "distanceTo(KITCHEN)": 20,
  "staffCardAvailable": true
 },
 "criteria": {
  "coffee.quality": {
    "GOOD": 0.5,
    "VERY_GOOD": 1,
    "BAD": 0.3
   },
  "cost": {
    "LOW": 0.8,
    "NONE": 1,
    "HIGH": 0.5
   },
   "staffCardAvailable":
    "staffCardAvailable?␣
        1.0:0.0",
   "rating":"coffee.quality␣
       *␣cost␣*␣
       staffCardAvailable"
  }
}
```

```
{
 "name": "GetOfficeCoffee",
 "rating": 0,
 "valuings": {
  "coffee.quality": "GOOD",
  "distanceTo(OFFICE)": 0,
  "cost": "NONE",
  "annInOffice": false
 },
 "criteria": {
  "coffee.quality": {
   "GOOD": 0.5,
   "VERY_GOOD": 1,
   "BAD": 0.3
  },
  "cost": {
   "LOW": 0.8,
   "NONE": 1,
   "HIGH": 0.5
  },
  "annInOffice":
      "annInOffice?␣
      1.0:0.0",
  "rating":"coffee.quality␣
      *␣cost␣*␣annInOffice"
 }
}
```

Figure 8: Plan Selection XAgentProcess event

6 CONCLUSION

In this paper we have advocated the use of design patterns to develop explainable-by-design agents (XAg) as a design feature instead of an afterthought. The design pattern we present TriQPAN is based on the pattern that every decision making process an agent undertakes - trigger, query, process, action and notify. The underlying principle is logging the events related to the TriQPAN process via an event store, and querying the event store to provide explanations.

We show how TriQPAN can be implemented to explain behaviours for any agent architecture by generating and storing the required events, but more importantly, we extend the SARL agent programming platform by integrating TriQPAN into its goal-reasoning engine. This allows developers to design and implement an agent system as per usual, with no (or very little) overhead, be able to query the agent systems for explanations about its behaviours.

We tested our SARL extension with two case studies - the artificial 'get coffee' example used in [31] and a 'search and rescue' case study used in [21, 22], to illustrate its use in practice.

Our approach was inspired by previous work on explaining the behaviour of BDI agents [10, 31] by inspecting the goal-plan structures and event-driven software patterns [6, 7]. Our event based design pattern is able to provide the same type of explanations and go beyond as the event store can be queried for richer types of queries such as temporal correlation queries as described in this work.

TriQPAN also enables requirements verification in a similar approach to [23]. Indeed, more advanced verifications can be implemented as we have access to more detailed information (compared to logs of traces). This verification could also be done "live" using *continuous queries*.

Whilst we have presented a few different types of explanation queries, future work involves investigating other query types. This includes expanding and further exploring the potential of *correlation queries* for explainablity and auditing. Future work also includes providing full support in our SARL implementation for those cases where we have planned to do so as identified in Table 1. Additionally, we will explore "introspection" reasoning based on information made available by XAgentProcess event .

In our approach we provide a basic translation from formal event queries and responses to intuitive english language representations. Future work includes adapting more sophisticated formalisms presented in [31] to present better natural language explanations.

Our XAg extension to SARL will be made freely available via GitHub. We make a call-to-action for the engineering MAS community consider extending other agent development platforms to incorporate the TriQPAN design pattern into their respective reasoning engines, providing users with built-in XAg functionality. This would greatly enhance the acceptance and use of MAS in the mainstream.

ACKNOWLEDGMENTS

This research is supported by the Commonwealth of Australia as represented by the Defence Science and Technology Group and the C2IMPRESS project funded by the EU.

REFERENCES

[1] Elhadi Belghache, Jean-Pierre Georgé, and Marie-Pierre Gleizes. [n.d.]. Towards an Adaptive Multi-agent System for Dynamic Big Data Analytics. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)* (2016-07). 753–758. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0121

[2] Rafael Bordini, Jomi Hübner, and Michael Wooldridge. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Vol. 8. https://doi.org/10.1002/9780470061848

[3] Mario Henrique Cruz Torres, Tony Van Beers, and Tom Holvoet. 2011. (No) More Design Patterns for Multi-Agent Systems. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11* (Portland, Oregon, USA) *(SPLASH '11 Workshops)*. Association for Computing Machinery, New York, NY, USA, 213–220. https://doi.org/10.1145/2095050.2095083

[4] Mehdi Dastani. 2008. 2APL: A Practical Agent Programming Language. *Autonomous Agents and Multi-Agent Systems* 16, 3 (jun 2008), 214–248. https://doi.org/10.1007/s10458-008-9036-y

[5] Mehdi Dastani and Bas Testerink. 2016. Design Patterns for Multi-Agent Programming. *Int. J. Agent-Oriented Softw. Eng.* 5, 2/3 (jan 2016), 167–202. https://doi.org/10.1504/IJAOSE.2016.080896

[6] Martin Fowler. [n.d.]. *CQRS*. martinfowler.com. https://martinfowler.com/bliki/CQRS.html

[7] Martin Fowler. [n.d.]. *Event Sourcing*. martinfowler.com. https://martinfowler.com/eaaDev/EventSourcing.html

[8] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[10] Maaike Harbers, Karel Bosch, and John-Jules Ch. Meyer. 2009. A Study into Preferred Explanations of Virtual Agent Behavior. In *Proceedings of the 9th International Conference on Intelligent Virtual Agents* (Amsterdam, The Netherlands) *(IVA '09)*. Springer-Verlag, Berlin, Heidelberg, 132–145. https://doi.org/10.1007/978-3-642-04380-2_17

[11] Maaike Harbers, Karel van den Bosch, and John-Jules Meyer. 2009. A Methodology for Developing Self-Explaining Agents for Virtual Training. In *Proceedings of the Second International Conference on Languages, Methodologies, and Development Tools for Multi-Agent Systems* (Torino, Italy) *(LADS'09)*. Springer-Verlag, Berlin, Heidelberg, 168–182. https://doi.org/10.1007/978-3-642-13338-1_10

[12] James Harland, David N. Morley, John Thangarajah, and Neil Yorke-Smith. [n.d.]. An Operational Semantics for the Goal Life-Cycle in BDI Agents. 28, 4 ([n. d.]), 682–719. https://doi.org/10.1007/s10458-013-9238-9

[13] Ivar Jacobson, Grady Booch, and James Rumbaugh. [n.d.]. *The Unified Software Development Process*. Addison Wesley.

[14] Aniek F. Markus, Jan A. Kors, and Peter R. Rijnbeek. 2021. The role of explainability in creating trustworthy artificial intelligence for health care: A comprehensive survey of the terminology, design choices, and evaluation strategies. *Journal of Biomedical Informatics* 113 (2021), 103655. https://doi.org/10.1016/j.jbi.2020.103655

[15] Fatma Outay, Stéphane Galland, Nicolas Gaud, and Abdeljalil Abbas-Turki. [n.d.]. Simulation of Connected Driving in Hazardous Weather Conditions: General and Extensible Multiagent Architecture and Models. 104 ([n. d.]), 104412. https://doi.org/10.1016/j.engappai.2021.104412

[16] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. 2005. Jadex: A BDI Reasoning Engine. In *Multi-Agent Programming: Languages, Platforms and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Springer US, Boston, MA, 149–174. https://doi.org/10.1007/0-387-26350-0_6

[17] A. S. Rao and M. P. Georgeff. 1991. Modeling rational agents within a BDI-architecture. In *Principles of Knowledge Representation and Reasoning. Proceedings of the second International Conference*. Morgan Kaufmann, San Mateo, 473–484.

[18] Sebastian Rodriguez, Nicolas Gaud, and Stéphane Galland. [n.d.]. SARL: A General-Purpose Agent-Oriented Programming Language, Vol. 3. IEEE Computer Society Press, 103–110. https://doi.org/10.1109/WI-IAT.2014.156

[19] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. [n.d.]. A Behaviour-Driven Approach for Testing Requirements via User and System Stories in Agent Systems. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2023-05-30) *(AAMAS '23)*. International Foundation for Autonomous Agents and Multiagent Systems, 1182–1190. https://dl.acm.org/doi/abs/10.5555/3545946.3598761

[20] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. [n.d.]. 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* (Richland, SC, 2021-05-03) *(AAMAS '21)*. International Foundation for Autonomous Agents and Multiagent Systems, 1064–1072. https://doi.org/10.5555/3461017.3461136

[21] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. 2021. User and System Stories: An Agile Approach for Managing Requirements in AOSE. In *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé (Eds.). ACM, 1064–1072. https://doi.org/10.5555/3463952.3464076

[22] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. 2023. A Behaviour-Driven Approach for Testing Requirements via User and System Stories in Agent Systems. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems* (London, United Kingdom) *(AAMAS '23)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1182–1190.

[23] Sebastian Rodriguez, John Thangarajah, Michael Winikoff, and Dhirendra Singh. [n.d.]. Testing Requirements via User and System Stories in Agent Systems. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2022-05-09) *(AAMAS '22)*. International Foundation for Autonomous Agents and Multiagent Systems, 1119–1127. https://ifaamas.org/Proceedings/aamas2022/pdfs/p1119.pdf

[24] Igor Tchappi, Yazan Mualla, Stéphane Galland, André Bottaro, Vivient Corneille Kamla, and Jean Claude Kamgang. [n.d.]. Multilevel and Holonic Model for Dynamic Holarchy Management: Application to Large-Scale Road Traffic. 109 ([n. d.]), 104622. https://doi.org/10.1016/j.engappai.2021.104622

[25] John Thangarajah and Lin Padgham. 2011. Computationally Effective Reasoning About Goal Interactions. *J. Autom. Reasoning* 47 (06 2011), 17–56. https://doi.org/10.1007/s10817-010-9175-0

[26] John Thangarajah, Lin Padgham, and Michael Winikoff. 2003. Detecting & Avoiding Interference Between Goals in Intelligent Agents. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (11 2003), 721–726.

[27] John Thangarajah, Lin Padgham, and Michael Winikoff. 2003. Detecting & Exploiting Positive Goal Interaction in Intelligent Agents. *Proceedings of the Interantional Conference on Autonomous Agents* 2, 401–408. https://doi.org/10.1145/860575.860640

[28] Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Studying Software Engineering Patterns for Designing Machine Learning Systems. *CoRR* abs/1910.04736 (2019). arXiv:1910.04736 http://arxiv.org/abs/1910.04736

[29] Michael Winikoff. 2005. *Jack™ Intelligent Agents: An Industrial Strength Platform*. 175–193. https://doi.org/10.1007/0-387-26350-0_7

[30] Michael Winikoff, Virginia Dignum, and Frank Dignum. 2018. Why Bad Coffee? Explaining Agent Plans with Valuings. In *Computer Safety, Reliability, and Security*, Barbara Gallina, Amund Skavhaug, Erwin Schoitsch, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 521–534.

[31] Michael Winikoff, Galina Sidorenko, Virginia Dignum, and Frank Dignum. 2021. Why bad coffee? Explaining BDI agent behaviour with valuings. *Artificial Intelligence* 300 (2021), 103554. https://doi.org/10.1016/j.artint.2021.103554

[32] Michael Winikoff, Galina Sidorenko, Virginia Dignum, and Frank Dignum. 2022. Why Bad Coffee? Explaining BDI Agent Behaviour with Valuings (Extended Abstract). In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, Lud De Raedt (Ed.). International Joint Conferences on Artificial Intelligence Organization, 5782–5786. https://doi.org/10.24963/ijcai.2022/810 Journal Track.