Computational implementation of an explainable state machine-based agent

Henrique E. Viana[1], Cesar A. Tacla[1], Jean M. Simão[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 80230-901 – Curitiba – PR – Brasil
Programa de Pós-Graduação em
Engenharia Elétrica e Informática Industrial (CPGEI)

Abstract. *There are cases in which a system must act autonomously. An example is conducting an exploration in an accident or natural catastrophe scenario to rescue victims. One of the areas dealing with autonomous systems is called Multiagent. For various reasons, it may be necessary for the system to generate explanations of its actions. For example, why did a certain agent rescue person A instead of B? When a system impacts human lives, generating explanations can be a requirement. Based on this, this work aims to carry out the computational implementation of an explainable agent. The implementation is based on a structural model and an explainability model, existing in the literature. There are also simplifications to ease the computational implementation, which is state machine based. The implementation was built in the Python language, and encapsulated in a generic library that can be configured for any scenario. This implementation has been tested for one scenario. In this scenario, the agent needs to rescue victims of an accident. The purpose of the tests is to analyze the generated explanations. The implementation proved to be able to capture the essential elements to explain the agent's decisions and generate explanations from them.*

Resumo. *Existem casos em que um sistema deve atuar de forma autônoma. Um exemplo é realizar uma exploração em um cenário de acidente ou catástrofe natural para resgatar as vítimas. Uma das áreas que trata de sistemas autônomos é nomeada de Multiagente. Por diversas razões, pode ser necessário que o sistema gere explicações sobre suas ações. Por exemplo, por que um certo agente resgatou a pessoa A em vez da B? Quando um sistema tem impacto sobre vidas humanas, gerar explicações pode ser um requisito. Com base nisso, esse trabalho tem como objetivo realizar a implementação computacional de um agente explicável. A implementação é baseada em um modelo estrutural e um modelo de explicabilidade, existentes na literatura. Existem também simplificações para facilitar a implementação computacional, que é baseada em máquina de estados. A implementação foi construída na linguagem Python, e encapsulada em uma biblioteca genérica que pode ser configurada para qualquer cenário. Essa implementação foi testada para um cenário. Nesse cenário o agente precisa resgatar vítimas de um acidente. O objetivo dos testes é analisar as explicações geradas. A implementação demonstrou ser capaz de capturar os elementos essenciais para explicar as decisões do agente e gerar explicações a partir delas.*

1. Introdução

Existem diversos problemas na Ciência da Computação nos quais é necessário que um sistema de agentes atue de forma autônoma para resolvê-lo. Isto é, cada agente deve perceber o ambiente a sua volta e se adaptar às mudanças, para alcançar seus objetivos.

Além disso, muitas vezes é necessário que um agente gere explicações sobre a escolha de seus objetivos. Por exemplo, porque o agente escolheu resgatar a pessoa A ao invés da pessoa B? Quando um agente tem impacto sobre vidas humanas, gerar explicações pode ser um requisito. A área de estudo que engloba essa geração de explicações é nomeada de Inteligência Artificial Explicável (IAE).

Com base nisso, esse trabalho tem como objetivo realizar a implementação computacional de um agente explicável. A implementação tem como base um modelo arquitetural de agentes [Castelfranchi and Paglieri 2007] e um de explicabilidade[1] [Jasinski and Tacla 2022], ambos abstratos, existentes na literatura. São também realizadas simplificações para facilitar a implementação computacional, que é baseada em máquina de estados. Isso ocorre pois nem todos os recursos do modelo arquitetural são implementados. Além disso, como o modelo de explicabilidade é implementado para uma arquitetura de agentes específica, muitas de suas interfaces genéricas puderam ser omitidas. Tal simplificação permite facilitar também uma implementação com paralelismos e/ou distribuição na execução. A arquitetura simplificada também cria uma base para explorar relações estatísticas de causa e efeito (modelos causais). Além disso, a união de um modelo arquitetural e um de explicabilidade ajuda a garantir a fidelidade das explicações ao que realmente está implementado no núcleo do agente. O modelo arquitetural é baseado em uma arquitetura de agentes de nome BBGP[2].

Este trabalho está estruturado na seguinte forma: a Seção 2 contém uma arquitetura abstrata de agente orientada a objetivos. Tal arquitetura é o ponto de partida para a implementação. A Seção 3 contém o modelo abstrato de geração de explicações. A Seção 4 contém a instanciação do modelo de agente e de geração de explicações, utilizado na implementação. O foco são as simplificações realizadas. A Seção 5 contém o desenvolvimento da implementação computacional. A Seção 6 contém os testes e discute os resultados do trabalho. A Seção 7 contém alguns trabalhos correlatos e a Seção 8 as conclusões e trabalhos futuros.

2. Modelo BBGP

Para entender a arquitetura BBGP é necessário primeiro entender a arquitetura BDI[3]. O BDI é uma arquitetura abstrata de agente orientada a objetivos. Isto é, estados do mundo que o agente persegue para torná-los realidade [Georgeff et al. 1999]. Além disso, são utilizadas crenças. Crenças são as coisas em que um agente acredita a respeito do ambiente, dos outros agentes e de si mesmo [Georgeff et al. 1999]. No BDI, os objetivos possuem apenas dois estados: desejos e intenções. Dessa maneira, o BDI utiliza as crenças para promover objetivos para desejos, e de desejos para intenções.

O BBGP, por sua vez, refina a arquitetura BDI, também utilizando crenças e objetivos. O refinamento ocorre por conter uma representação explícita das crenças

[1]Agradecimentos ao projeto do CNPq, número 409523/2021-6.

[2]Do inglês: *Belief-Based Goal Processing*.

[3]Do Inglês: *Belief–Desire–Intention*.

que motivam ou impedem o avanço de um objetivo [Castelfranchi and Paglieri 2007]. Com tais crenças organizadas em diversas categorias, o número de estados possíveis para um objetivo é maior na arquitetura BBGP do que na BDI (Tabela 1). No BBGP, um objetivo pode passar por mais que dois estados. Nesse contexto, as crenças são utilizadas para construir as intenções do agente, por meio da promoção de objetivos [Castelfranchi and Paglieri 2007].

Tabela 1. Categorias de promoções de objetivos e crenças no BBGP. O "+" indica crenças que motivam uma promoção, e o "-" crenças que impedem uma promoção. Adaptado de [Castelfranchi and Paglieri 2007].

Objetivos	Etapa	Categoria de crenças
Ativos	1) Ativação.	Crenças de gatilho (+) Crenças condicionais (+)
Alcançáveis	2) Avaliação.	Crenças de autorrealização: (-) Crenças de satisfação (-) Crenças de impossibilidade (-)
Escolhidos	3) Deliberação.	Crenças de custo (-) Crenças de incompatibilidade (-) Crenças de preferência (+) - Crenças de valor (+) - Crenças de urgência (+)
Executivos	4) Checagem.	Crenças de pré-condição (-) - Crenças de incompetência (-) - Crenças crenças de falta de condições (-) Crenças de meio-fim (+)

Essas arquiteturas abstratas, com objetivos e estados de objetivos, podem ser utilizadas em modelos de explicabilidade [Morveli-Espinoza et al. 2022]. Além disso, como o processo de decisão no BBGP é mais granular que no BDI, no BBGP as explicações podem ser mais sofisticadas [Morveli-Espinoza et al. 2022]. Existem inclusive diversos tipos de explicações que podem utilizar tal arquitetura.

3. Geração de explicações contrastivas

Entender os diversos tipos de explicações é importante para entender o que são explicações contrastivas. Os autores [Haynes et al. 2009] destacam 4 tipos:

- Ontológicas: ajuda o solicitante a entender um conceito. Por exemplo, o que Y significa? Qual a categoria de Z? Em que X se assemelha a Y?
- Mecanicistas: ajuda o solicitante a entender relações de causa e efeito. Por exemplo, qual evento X causou o evento Y? Quais as consequências do evento Z?
- Operacionais: ajuda o solicitante a entender etapas, procedimentos e formas de se realizar algo. Por exemplo, quais as etapas para realizar o procedimento X?
- Racionais: ajuda o solicitante a entender o porquê de algo. Por exemplo, qual o objetivo de realizar X?

É importante também entender os diversos princípios desejáveis para uma explicação. Os autores [Grice 1975] e [Jasinski and Tacla 2022] descrevem 4 princípios:

- Quantidade: Faça sua contribuição tão informativa quanto for necessário. Não faça sua contribuição mais informativa do que o necessário.
- Qualidade: Não diga o que você acredita ser falso. Não diga aquilo para o qual você não tem evidências adequadas.

- Relação: Seja relevante.
- Maneira: Evite se expressar de forma obscura, evite a ambiguidade, seja breve, explique de forma ordenada.

Buscando tais propriedades para gerar explicações acerca da escolha de objetivos, [Jasinski and Tacla 2022] criaram um modelo abstrato e genérico. Tal modelo gera explicações contrastivas em arquiteturas de agentes orientadas a objetivos. Explicações contrastivas buscam explicar porque X ocorreu ao invés de Y. Para tal os autores definiram interfaces que o agente deve implementar (Figura 1) para salvar o estado mental do agente no momento da escolha de um objetivo. As interfaces contém os elementos que definem a escolha e são utilizadas para desacoplar a implementação do agente da implementação do gerador de explicações.

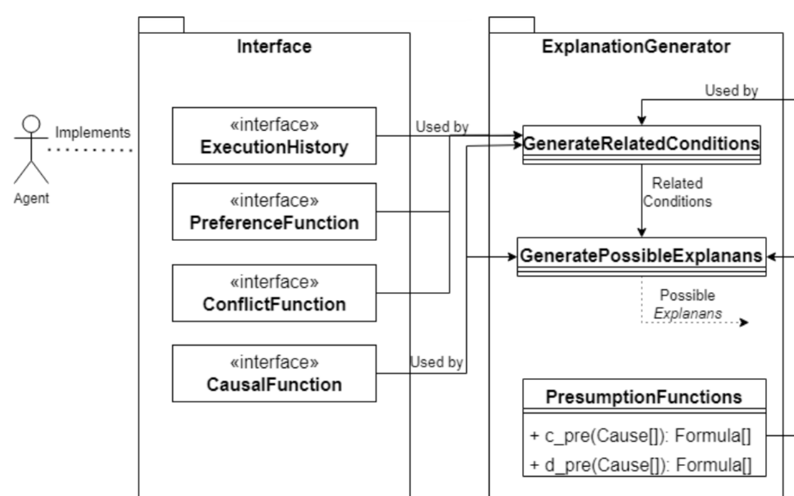

Figura 1. Interfaces para geração de explicações contrastivas [Jasinski and Tacla 2022].

As interfaces de [Jasinski and Tacla 2022] possuem a seguinte finalidade:

- *ExecutionHistory*: contém um histórico do processamento do agente. Cada entrada associa o índice do ciclo de raciocínio do agente aos objetivos que mudaram de estado. São também associadas às respectivas mudanças nas crenças responsáveis pelas promoções dos objetivos.
- *PreferenceFunction*: é uma função que representa as preferências entre objetivos de acordo com o que foi implementado no agente. Dado um par de objetivos $\langle g_1, g_2 \rangle$, a função retorna 0 se g_1 e g_2 tem preferências iguais, 1 se g_1 é preferencial sobre g_2, e -1 caso contrário.
- *ConflictFunction*: é uma função que traduz a implementação da resolução de conflitos entre um par de objetivos $\langle g_1, g_2 \rangle$ implementada no agente. Dado $\langle g_1, g_2 \rangle$ esta função retorna verdadeiro se há conflito e falso, caso contrário.
- *CausalFunction*: contém uma associação de eventos e possíveis causas para esses eventos. No caso de objetivos, o evento é o objetivo e as crenças são as causas que habilitam o agente a executá-lo. A representação é dada por um par $\langle g, [causes] \rangle$ onde g é o objetivo e $[causes]$ a lista de crenças.

Embora as interfaces de [Jasinski and Tacla 2022] sejam genéricas, não há como garantir a fidelidade entre o que efetivamente foi implementado no agente (e.g. atribuição

de preferências entre objetivos, resolução de conflitos entre objetivos) e o que o programador implementou nos métodos da interface do gerador de explicações. Além disso, em uma representação de objetivos estrutural específica, algumas interfaces poderiam ser omitidas. Um exemplo seria uma representação por máquina de estados. Se os conflitos e preferências entre objetivos forem entidades dotadas de estado, suas mudanças de estado poderiam ser salvas em um histórico. A recuperação desse histórico (em essência a interface *ExecutionHistory*) eliminaria a necessidade das interfaces *PreferenceFunction* e *ConflictFunction*. Além disso, [Jasinski and Tacla 2022] não abordam a seleção de uma explicação única. Com base nisso, é possível definir um modelo arquitetural de agente que unifica a representação de objetivos e a geração de explicações.

4. Modelo Arquitetural de Agente

Ao analisar a arquitetura BBGP e o trabalho de [Jasinski and Tacla 2022], é possível definir uma simplificação de ambos baseada em máquina de estados. Para isso, todas as entidades de um agente, tais como atributos do ambiente, atributos do canal de comunicação (para comunicação entre agentes), crenças, funções de revisão de crenças, promoções de objetivos, objetivos, conflitos, planos e ações são entidades dotadas de estados. Essas entidades são independentes entre si e se comunicam enviando suas mudanças de estados (Figura 2) como forma de inferência. Tais mudanças de estado podem ser enviadas entre *threads* ou por rede. Isso possibilita a criação de implementações paralelas e/ou distribuídas. Todavia, implementações concorrentes, paralelas ou distribuídas podem exigir mecanismos adicionais de sincronização. Essa arquitetura é inspirada em um paradigma de programação de nome PON[4][dos Santos Neves and Linhares 2021].

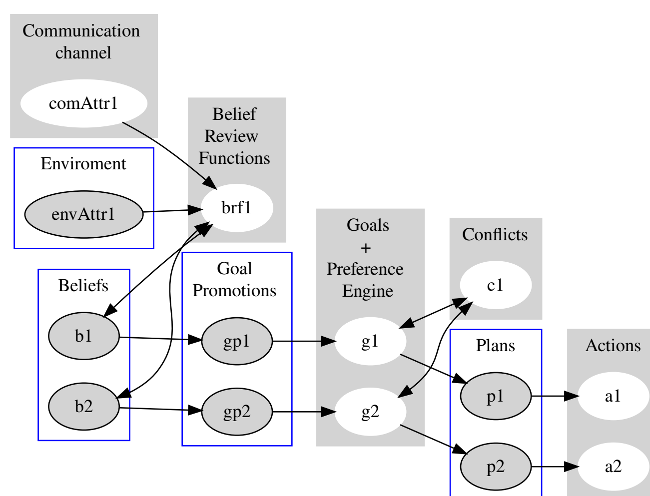

Figura 2. Entidades em um BBGP simplificado, conectadas entre si de forma a propagar suas mudanças de estado.

Nessa simplificação, não são implementadas as categorias de crenças e de promoções de objetivos do BBGP. Essas categorias são abertas. Dessa maneira, o programador pode instanciar quantas promoções quiser para um objetivo, assim como quantas

[4]Paradigma Orientado a Notificações.

categorias de crenças desejar. Embora isso resulte em uma implementação incompleta do BBGP, possibilita a generalização para outras arquiteturas orientadas a objetivos. As promoções de objetivos são independentes entre si. Dessa maneira, não é necessário um objetivo ter tido a promoção X para poder obter a promoção X+1. Todavia, ainda é necessário que todas as promoções tenham ocorrido para o agente decidir executar o objetivo. Como forma computacional de preferência entre objetivos, é utilizado prioridades. Isso é, objetivos com maior prioridade (que é um valor numérico) tem preferência sobre objetivos com menor prioridade. Cada promoção de objetivo pode opcionalmente incrementar a prioridade do subsequente objetivo em algum valor, ao promovê-lo.

Outro detalhe é que as promoções devem sempre propagar suas mudanças de estado, mesmo que tais mudanças não indiquem o sucesso da promoção. Isso possibilita o funcionamento de despromoções. Em caso de conflitos, o objetivo com maior prioridade é selecionado, e os outros removidos. Ao executar objetivos, o agente utiliza uma fila de objetivos ordenada pela prioridade (intenções), executando os objetivos com maior prioridade primeiro. Pode-se também ter várias instâncias de um mesmo objetivo. Além disso, a seleção de planos se dá também pela prioridade dos objetivos. Ao definir um plano para um objetivo, é possível definir a prioridade mínima para selecioná-lo. Se um objetivo tem prioridade 5, e possui um plano A com prioridade mínima 3 e um plano B com prioridade mínima 4, o plano B é selecionado.

Um exemplo de implementação concorrente pode ser visualizada no Algoritmo 1. Nessa implementação, as propagações de mudança de estado das entidades são utilizadas para direcionar a inferência. Isso permite diminuir a quantidade de *loops* e quantidade de código.

Algoritmo 1 Processamento concorrente em um BBGP baseado em propagação de mudanças de estado.

```
1: entities ← [envAttrs, channelAttrs, beliefs, brfs, promotions, goals, conflicts, plans, actions]
2: while true do                                          ▷ Agent execution
3:     for entity ∈ entities do
4:         if receivedStateChanges(entity) then
5:             exec entity(time())
6:             if hasStateChange(entity) then
7:                 sendToConnectedEntities(entity, getState(entity))
8:             end if
9:         end if
10:    end for
11: end while
```

Para gerar explicações, as entidades salvam suas mudanças de estado em um histórico. Como a inferência é assíncrona, é salvo também o tempo em que essas mudanças ocorreram. Para gerar explicações nessa arquitetura específica, o agente necessita implementar 3 interfaces:

- *Description*: são tuplas $\langle e_{id}, d \rangle$. Cada tupla associa um ID de entidade (e_{id}) a uma descrição ontológica d. Um exemplo seria associar o texto "Objetivo de salvar uma vítima" (d) a entidade $id(g_1)$. Essa descrição d é definida estaticamente junto com as definições das entidades.
- *StateHistory*: são tuplas $\langle from_{id}, to_{id}, s, t_a, t \rangle$. Cada tupla associa um ID de entidade ($from_{id}$) a uma informação de estado s e um tempo t. Além disso, em muitos casos a informação de estado é propagada para uma entidade de destino. Nesses casos, to_{id} tem um valor não vazio e representa o ID dessa entidade de

destino. Por fim, o cálculo do estado pode demorar, ou pode ser assíncrono. Por isso, é importante salvar o tempo em que o cálculo de transição de estado começou (t_a).

- *CausalFunction*: são tuplas $\langle e_{id}, c_{id} \rangle$. Cada tupla associa um ID de entidade (e_{id}) que representa um evento, a um ID de entidade que representa uma possível causa c_{id} para esse evento. Por exemplo, toda crença é uma possível causa para uma promoção de objetivo. Todo objetivo é uma possível causa para planos relacionados a esse objetivo. Todo plano é uma possível causa para ações relacionadas a esse plano. Essa interface é inspirada nos autores [Jasinski and Tacla 2022]. Todavia, diferente de [Jasinski and Tacla 2022], são consideradas todas as entidades que fazem parte do agente (como planos e ações).

A geração de explicações tem natureza ontológica com a ajuda da interface *Description*. Além disso, tem também natureza mecanicistas de causa e efeito com a ajuda das interfaces *StateHistory* e *CausalFunction*. Diferente dos autores [Jasinski and Tacla 2022], a funcionalidade da interface *ConflictFunction* é suprida pela interface *StateHistory*. Isso ocorre porque o conflito é uma entidade dotada de estado. Sendo assim, suas mudanças de estado são salvas no histórico (*StateHistory*). A interface *PreferenceFunction* também é omitida. Tal omissão ocorre pois os objetivos contém suas prioridades em seus valores de mudança de estado (salvas em *StateHistory*). Uma vez que essa informação é salva e o mecanismo computacional de preferência é fixo e conhecido, não existe necessidade da *PreferenceFunction*. É importante destacar que, o histórico das entidades que propagam mudanças de estado formam uma base para implementação de modelos causais, como o do autor [Pearl 2009]. Isso permitiria gerar explicações probabilísticas, como por exemplo "X aconteceu provavelmente por causa de Y e Z", ou "se Z acontecer, então provavelmente X acontece".

Nas explicações contrastivas, é necessário explicar "porque ocorreu X ao invés de Y". Para isso, é inicialmente explicado "o porque X ocorreu" (Algoritmo 2). De início, considera-se que é conhecida a ação X do agente que deve ser explicada. Essa ação é o efeito inicial do procedimento, na forma de uma tupla de histórico. Na sequência a interface *CausalFunction* recupera as possíveis causas para esse efeito. Para cada possível causa, recupera-se do histórico a transição de estado dessa causa. Essa transição de estado tem como origem a causa e como destino o efeito em questão. Além disso, o tempo $t - 1$ da causa é o mais próximo do tempo de ativação do efeito t_a no histórico. Esse procedimento é executado de forma recursiva. Nesse contexto, a variável r representa o nível da recursão.

Algoritmo 2 Geração de explicações.

```
1:  procedure XHISTORY(effectHistEntry, r = 1, past)
2:      if origin(effectHistEntry) IN past then
3:          return                                          ▷ Cycle prevention
4:      else
5:          add(origin(effectHistEntry), past)
6:      end if
7:      causes ← [ ]
8:      for c ∈ causalFunction(origin(effectHist)) do
9:          hist ← lastHist(c, target(effectHist), actTime(effectHist))
10:         add(causes, hist)
11:     end for
12:     sortByTime(causes)
13:     for causeHist ∈ causes do
14:         explain(causeHist, r)
15:         xHistory(causeHist, r + 1, past)
16:     end for
17: end procedure
```

A continuação da explicação (os contrastes, caso existam), como no exemplo "porque Y não ocorreu", são gerados por um outro procedimento. Esse outro procedimento também tem como parâmetro de entrada uma ação do agente como efeito inicial (Y), na forma de uma tupla de histórico. Essa tupla é inventada, para simular o evento que deveria ter acontecido. O objetivo é saber o porque tal efeito não ocorreu. Para isso, são exploradas somente as causas que não ocorreram. Para auxiliar a seleção de uma explicação única, cada retorno do procedimento possui uma causa associada a um score. Esse score é um fator que representa a quantidade de causas multiplicado pelo inverso do nível de recursão ($\frac{1}{r} * \frac{count(causes)}{count(possibleCauses)+1}$). Esse score soma também os scores das causas aninhadas (as causas das causas). Pode-se utilizar esse score para visualizar causas que tiveram mais (ou menos) motivos para acontecer. É considerado também que, um maior nível de recursão indica uma menor relevância. Isso ocorre pois quanto maior o nível da recursão, mais distante o algoritmo está do efeito inicial a ser explicado. Esse procedimento complementar para explicações contrastivas pode ser visualizado no Algoritmo 3.

Algoritmo 3 Geração de explicações de contraste.

```
1: procedure xNot(effectHistEntry, r = 1, past)
2:     if origin(effectHistEntry) IN past then
3:         return 0                                    ▷ Cycle prevention
4:     else
5:         add(origin(effectHistEntry), past)
6:     end if
7:     causes ← [ ]
8:     toExplore ← [ ]
9:     minTime ← 0
10:    score ← 0
11:    lastSimilarHist ← lastSimilarHistBefore(effectHist)
12:    if lastSimilarHist IS NOT NULL then
13:        minTime ← actTime(lastSimilarHist) + 1
14:    end if
15:    possibleCauses ← causalFunction(origin(effectHist))
16:    for c ∈ possibleCauses do
17:        hist ← lastHist(c, target(effectHist), actTime(effectHist), minTime)
18:        if hist IS NULL then
19:            add(causes, hist)
20:        else
21:            add(toExplore, c)
22:        end if
23:    end for
24:    score ← score + 1/r * count(causes)/(count(possibleCauses)+1)
25:    for c ∈ toExplore do
26:        possibleHist ← createPossibleHist(c, effectHist)
27:        score = score + xNot(possibleHist, r + 1, past)
28:    end for
29:    explain(effectHistEntry, score, r)
30:    return score
31: end procedure
```

5. Desenvolvimento do Trabalho

Esta seção descreve o desenvolvimento do trabalho. O desenvolvimento se baseia na implementação computacional (uma biblioteca) da arquitetura descrita na Seção 4. São também discutidos aspectos do código, técnicas e tecnologias utilizadas. A biblioteca é desenvolvida em Python[5]. Na biblioteca, existem também diversas classes abstratas (Figura 3) que permitem especificar alguns mecanismos.

[5]Disponível em `https://github.com/hviana/goal_processing`.

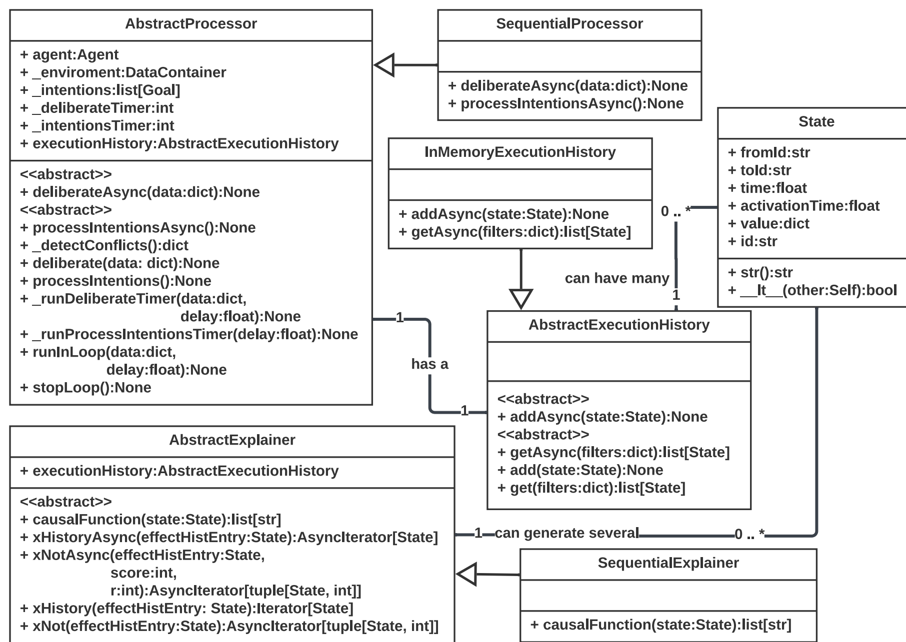

Figura 3. Diagrama UML de parte da biblioteca contendo as classes abstratas.

Ao observar o diagrama UML, é possível visualizar que existem 3 classes abstratas, *AbstractProcessor*, *AbstractExecutionHistory* e *AbstractExplainer*. A classe *AbstractProcessor* abstrai o modelo de ciclo de raciocínio do agente. Tal classe considera que o processo de deliberação (revisão de crenças e promoções de objetivos) e a execução de objetivos são concorrentes entre si. Dessa maneira, existe um método abstrato para a deliberação e um outro método abstrato para a execução de objetivos. Esses métodos compartilham essencialmente a mesma fila de intenções. Internamente, o mecanismo de inferência executa esses dois métodos de forma concorrente. Isso faz com que não seja possível uma implementação totalmente sequencial. Na biblioteca, existe uma implementação dessa classe abstrata (*SequentialProcessor*). Nessa implementação o método de deliberação e o método de execução de objetivos são sequenciais, embora sejam executados concorrentemente entre si.

A concorrência entre a deliberação e a execução de objetivos é intencional. Sem essa concorrência, podem ocorrer algumas situações indesejadas. Por exemplo, pode ser que em uma iteração de inferência o agente recebe as coordenadas de 3 acidentes com prioridades baixas e insere os respectivos 3 objetivos na fila. Todavia, enquanto a fila é processada, as variáveis de ambiente mudam e indicam um novo acidente com uma prioridade alta (representando um alto risco para a vítima). Nesse caso, o agente irá terminar o ciclo atual de inferência, resgatando as vítimas dos 3 acidentes com prioridades baixas. Só no próximo ciclo de inferência o agente irá resgatar a vítima com o acidente com prioridade alta. Com a concorrência, o novo acidente com uma prioridade alta terá o respectivo objetivo inserido na fila de intenções que ainda está em processamento. Dessa maneira, uma vez que a fila de intenções é ordenada, ele irá para o topo da fila e será o próximo objetivo executado pelo agente, antes dos objetivos dos acidentes com prioridades baixas.

A classe abstrata *AbstractExplainer* implementa os métodos de gerar explicações. Todavia, tal classe possui um método abstrato que representa a interface *CausalFunction*.

As interfaces Description e *StateHistory* já possuem uma implementação padrão. Na biblioteca, existe uma implementação dessa classe abstrata (*SequentialExplainer*). Essa implementação foi criada para trabalhar em conjunto com a classe *SequentialProcessor*. Os métodos de geração de explicação utilizam *generators*, com o recurso *yield* da linguagem. Com isso, a geração de explicações é dinamicamente iterativa e pode ser interrompida a qualquer momento se for detectada ser suficiente.

Por fim, a classe abstrata *AbstractExecutionHistory* tem como objetivo abstrair como as mudanças de estados são salvas. Tal classe possui basicamente um método abstrato para recuperar estados e um método abstrato para salvar estados. Na biblioteca, existe uma implementação dessa classe abstrata (*InMemoryExecutionHistory*). Essa classe salva todas as mudanças de estados em um vetor ordenado pelo tempo do relógio da máquina na memória RAM[6].

A implementação possui alguns destaques. Ela foi criada em uma estrutura para ser publicada no gerenciador de pacotes da linguagem. A implementação também é amplamente genérica e pode ser configurada para diversos casos. É possível inclusive instanciar quantos agentes forem necessários em um mesmo programa (sistemas Multiagentes). Esses agentes podem se comunicar pelo canal de comunicação entre agentes, implementado na biblioteca. O código por sua vez é otimizado para comunicação por rede. Dessa maneira, possui suas abstrações internas implementadas com métodos assíncronos.

6. Testes e Resultados

Esta seção contém os testes e resultados desse trabalho. Para testar a biblioteca desenvolvida, é utilizado um cenário de resgate de vítimas. O objetivo dos testes é analisar as explicações providenciadas pelo agente.

No cenário utilizado, existem dois tipos de agentes, os exploradores (AE) e os socorristas (AS). Ambos não conhecem o terreno do acidente e nem a posição das vítimas. O AE deve explorar o ambiente e construir um mapa contendo os obstáculos e as vítimas que conseguir localizar. Uma vez localizada uma vítima, o AE tem a capacidade de ler os sinais-vitais (e.g. frequência respiratória e cardíaca, pressão arterial). O AS deve ir até as vítimas já localizadas. Para isso, recebe um mapa do AE. Ambos têm um tempo limite devido à carga de bateria. Findado este tempo, devem retornar à posição base. A ideia é que o AE explore o ambiente, retorne à base e passe o mapa de vítimas e obstáculos ao AS. Em seguida, o AS entra em operação. Ele define as vítimas que irá socorrer em função do tempo que lhe for dado. O AS também responsável por levar os kits de primeiros-socorros a cada vítima escolhida e retorna à base antes do término do tempo limite.

O AE trabalha somente com os seguintes objetivos:

1. Localizar vítima. Podem haver várias instâncias desse objetivo.
2. Construir mapa. Esse objetivo estará sempre em execução.
3. Carregar bateria. Necessário para sobrevivência.

O AS terá os seguintes objetivos competindo entre si:

1. Salvar vítimas, que são localizadas pelo AE. Podem haver várias instâncias desse objetivo.

[6] Do inglês: *Random Access Memory*.

2. Atualizar o mapa. Caso o ambiente mude, obstáculos mudam se novos deslizamentos ocorrem, a posição das vítimas também.
3. Carregar bateria. Necessário para sobrevivência.

As explicações geram uma lista de mudanças de estados, por meio dos procedimentos geradores de explicação. Cada item da lista é impresso em uma linha. No começo de cada linha, existe o nível de recursão do procedimento. O nível de recursão permite saber o quão avançada está a inferência. Por exemplo, as variáveis de ambiente estão no nível 7 de recursão. As funções de revisão de crença estão no nível 6 e as crenças no nível 5. As promoções de objetivos estão no nível 4, os objetivos no nível 3, os planos no nível 2 e as ações no nível 1. Ao final de cada linha, existe o tempo em que a mudança de estado foi propagada. Nas explicações, o tempo foi traduzido de nanosegundos para um número inteiro sequencial, para melhor visualização.

Outro detalhe é que para cada solicitação de explicação, é criado um arquivo em Python. Cada arquivo possui as variáveis de ambiente definidas estaticamente. O arquivo também contém a instanciação do agente, com todos os seus elementos como os objetivos. Ao final do arquivo, existe a chamada ao procedimento gerador de explicações. A entrada para o procedimento é uma tupla de histórico. Dessa maneira, existe também uma chamada a um procedimento de recuperação do histórico, para recuperar a tupla que contém ação que o agente executou e deve ser explicada.

Com base no cenário e nos procedimentos geradores de explicação, são solicitadas diversas explicações aos agentes. São solicitadas as seguintes explicações aos AE:

1. Por que você decidiu voltar à base no tempo T em vez de continuar com a localização das vítimas?

Tabela 2. Agente volta a base primeiro para depois continuar com a localização das vítimas. Trecho do objetivo de localizar coordenadas.

query: explainer.xHistory(effectStateGoal1Action)
top: Action - Recording victim coordinates - time: 9
0: Plan - Localizing victims coordinates - time: 8
1: Goal - Localizing victims - Priority: 0 - time: 5
2: GoalPromotion: executive - Promoting accident by risk - Priority increment: 0 - time: 4
3: Attribute: beliefs.accident - {'coordinates': [20, 40], 'risk': 'low'} - time: 3
4: BeliefReviewFunction - Review accidents found - time: 2
5: Attribute: env.accidents - [{'coordinates': [20, 40], 'risk': 'low'}] - time: 1
3: Attribute: beliefs.accident.risk - low - time: 3

Tabela 3. Agente volta a base primeiro para depois continuar com a localização das vítimas. Trecho do objetivo de voltar a base.

query: explainer.xHistory(effectStateGoal2Action)
top: Action - Recharge battery in base - time: 7
0: Plan - Recharge battery in base - time: 6
1: Goal - Go to base - Priority: 1 - time: 5
2: GoalPromotion: executive - Promote if battery level is low - Priority increment: 1 - time: 4
3: Attribute: beliefs.resources.battery - low - time: 3
4: BeliefReviewFunction - Review battery level - time: 2
5: Attribute: env.battery - 25 - time: 1

Nas Tabelas 2 e 3, é possível visualizar que a ação de recarregar bateria ocorreu primeiro. Isso ocorreu porque o objetivo da ação subsequente (voltar para a base), teve uma prioridade maior (Tabelas 2 e 3, linha da recursão 1).

2. Por que você conseguiu localizar a vítima A e não decidiu não ler seus sinais-vitais?

Tabela 4. Agente não leu os sinais da vítima. Trecho do objetivo de localizar coordenadas.

query: explainer.xHistory(effectStateGoal1Action)
top: Action - Recording victim coordinates - time: 7
0: Plan - Recording victim coordinates - time: 6
1: Goal - Locate Victim - Priority: 0 - time: 5
2: GoalPromotion: executive - Promote location - Priority increment: 0 - time: 4
3: Attribute: beliefs.victim - {'coordinates': [20, 40], 'risk': 'medium'} - time: 3
4: BeliefReviewFunction - Review victims found - time: 2
5: Attribute: env.victims - [{'coordinates': [20, 40], 'risk': 'medium'}] - time: 1

Tabela 5. Agente não leu os sinais da vítima. Trecho do objetivo de ler sinais da vítima.

query: explainer.xNot(possibleStateGoal2Action)
3: Score: 0.16. GoalPromotion: executive - Read vital signs only of serious casualties - time: 0
2: Score: 0.16. Goal - Read vital signs - time: 0
1: Score: 0.33. Plan - Read vital signs - time: 7
0: Action - Read vital signs - time: 7

Nas Tabelas 4 e 5, é possível visualizar que a ação de localizar vítimas ocorreu. A ação de ler os sinais vitais não ocorreu, sendo explorada pelo procedimento *xNot*. Tal procedimento encerra a recursão em uma promoção de objetivo. Essa promoção só promove o objetivo de ler sinais vitais de acidentes graves. Como essa promoção não ocorreu, ela é o motivo para a ação subsequente não ter ocorrido. O tempo 7 está relacionado com o possível estado, e foi inserido manualmente. O tempo 0 nos outros estados ocorre pois eles não existem no histórico.

3. Por que você conseguiu localizar a vítima A e não localizou a B que estava nas coordenadas (xb, yb)?

Tabela 6. Agente não leu coordenadas da vítima. Trecho do acidente A.

query: explainer.xHistory(effectStateGoal1Action1)
top: Action - Recording victim coordinates - time: 7
0: Plan - Recording victim coordinates - time: 6
1: Goal - Localizing victims - Priority: 0 - time: 5
2: GoalPromotion: executive - Promote location - Priority increment: 0 - time: 4
3: Attribute: beliefs.accident - {'coordinates': [25, 40], 'risk': 'high'} - time: 3
4: BeliefReviewFunction - Review accidents found - time: 2
5: Attribute: env.accidents - [{'coordinates': [20, 30], 'risk': 'high'}, {'coordinates': [25, 40], 'risk': 'high'}] - time: 1
3: Attribute: beliefs.accident.risk - high - time: 3

Tabela 7. Agente não leu coordenadas da vítima. Trecho do acidente B.

query: explainer.xHistory(effectStateGoal1Action2)
top: Action - Recording victim coordinates - Error: Inaccessible victim - time: 13
0: Plan - Recording victim coordinates - time: 12
1: Goal - Localizing victims - Priority: 0 - time: 11
2: GoalPromotion: executive - Promote location - Priority increment: 0 - time: 10
3: Attribute: beliefs.accident - {'coordinates': [20, 30], 'risk': 'high'} - time: 9
4: BeliefReviewFunction - Review accidents found - time: 8
5: Attribute: env.accidents - [{'coordinates': [20, 30], 'risk': 'high'}] - time: 7
3: Attribute: beliefs.accident.risk - high - time: 9

Nas Tabelas 6 e 7, é possível visualizar que houve uma falha na segunda ação de localizar coordenadas (acidente B, Tabela 7, linha da recursão "top").

Para o AS, são solicitadas as seguintes explicações:

1. Por que você decidiu não salvar a vítima A?

Tabela 8. Agente decidiu não salvar a vítima A.

query: explainer.xHistory(effectStateGoal1Action1)
top: Action - Recharge battery in base - time: 7
0: Plan - Recharge battery in base - time: 6
1: Goal - Recharge battery - Priority: 2 - time: 5
2: Conflict - Charge battery instead of rescuing victim - time: 4
2: GoalPromotion: executive - Promote recharge battery - Priority increment: 2 - time: 4
3: Attribute: beliefs.resources.battery - low - time: 3
4: BeliefReviewFunction - Review battery level - time: 2
5: Attribute: env.battery - 25 - time: 1

Ao analisar a Tabela 8, é possível visualizar que houve um conflito (primeira linha da recursão 2). Nesse conflito não é possível resgatar uma vítima e recarregar a bateria ao mesmo tempo. Todavia, recarregar a bateria possui uma prioridade maior. Por isso, o objetivo de resgatar a vítima foi removido pelo conflito.

2. Por que você decidiu salvar a vítima A e não a vítima B que estavam próximas?

Tabela 9. Agente resgata vítima A.

query: explainer.xHistory(effectStateGoal1Action1)
top: Action - Rescue victims - time: 7
0: Plan - Rescue victims - time: 6
1: Goal - Rescue victims - Priority: 1 - time: 5
2: GoalPromotion: executive - Rescue serious accidents - Priority increment: 1 - time: 4
3: Attribute: beliefs.accident - {'coordinates': [20, 40], 'risk': 'high'} - time: 3
4: BeliefReviewFunction - Review environmental accidents - time: 2
5: Attribute: env.accidents - [{'coordinates': [15, 30], 'risk': 'medium'}, {'coordinates': [20, 40], 'risk': 'high'}]- time: 1
3: Attribute: beliefs.accident.risk - high - time: 3

Tabela 10. Agente resgata vítima B, somente após a vítima A.

query: explainer.xHistory(effectStateGoal1Action2)
top: Action - Rescue victims - time: 9
0: Plan - Rescue victims - time: 8
1: Goal - Rescue victims - Priority: 0 - time: 5
2: GoalPromotion: executive - Rescue serious accidents - Priority increment: 0 - time: 4
3: Attribute: beliefs.accident - {'coordinates': [15, 30], 'risk': 'medium'} - time: 3
4: BeliefReviewFunction - Review environmental accidents - time: 2
5: Attribute: env.accidents - [{'coordinates': [15, 30], 'risk': 'medium'}]- time: 1
3: Attribute: beliefs.accident.risk - medium - time: 3

Nas Tabelas 9 e 10, é possível visualizar que o objetivo de salvar a vítima A tinha uma prioridade maior (Tabelas 9 e 10, linha da recursão 1). Por isso, esse objetivo foi executado primeiro.

7. Trabalhos correlatos

Esta seção contém alguns trabalhos correlatos. Na Tabela 11 é possível visualizar tais trabalhos de acordo com algumas propriedades em comum.

Tabela 11. Trabalhos correlatos.

Trabalhos	Propriedades em comum
[Harbers et al. 2010], [Sado et al. 2023], [Jasinski and Tacla 2022]	Orientados a objetivos
[Jasinski and Tacla 2022], [Morveli-Espinoza et al. 2022]	Baseados no histórico de execução

É de destaque que os autores [Harbers et al. 2010] classificam a geração de explicações em agentes orientados a objetivos em 3 categorias:

- Histórico causal: baseada no histórico de execução do agente. Possui foco na origem das crenças e dos objetivos.
- Razão: explica as crenças e objetivos. Todavia, não explica quando tais objetivos e crenças são utilizados.
- Fatores possíveis: considera as capacidades de um agente. Isso é, dado um contexto e conhecendo as capacidades do agente, explica o que este poderia ter feito para ter chego no respectivo contexto.

8. Conclusão e Trabalhos Futuros

Nesse trabalho foi realizado a implementação computacional de um agente explicável. A implementação foi baseada em modelos teóricos existentes na literatura. Para isso, foram realizadas simplificações em relação aos modelos existentes na literatura. As simplificações serviram para facilitar a implementação computacional. Todavia, com tais simplificações, os modelos foram implementados parcialmente. Apesar disso, é uma implementação integrada de um modelo arquitetural orientado a objetivos com um modelo genérico de explicabilidade. Isso ajuda a garantir que as explicações geradas sejam totalmente compatíveis com a implementação do agente. Os modelos abstratos BBGP e de explicabilidade de [Jasinski and Tacla 2022] serviram de guia para a presente implementação.

A implementação foi construída na linguagem Python, e encapsulada em uma biblioteca genérica que pode ser configurada para qualquer cenário. Isso é importante, pois permite utilizar a biblioteca em outros contextos. Pode-se por exemplo criar agentes integrados com modelos de linguagem, como a solução de nome AgentGPT[7].

A biblioteca foi testada em um cenário. A implementação demonstrou ser capaz de gerar explicações sobre as decisões do agente. Além disso, as explicações trouxeram algumas vantagens em relação aos modelos originais. A implementação traz a possibilidade, por exemplo, de gerar explicações com informações de planos e ações das decisões do agente. As explicações, todavia, se parecem mais com apenas uma leitura do histórico de execução das entidades de um agente. Isso significa que ao invés de existir uma explicação clara, existe apenas uma justificativa em termos computacionais.

Como trabalhos futuros, pode-se explorar uma implementação similar para a arquitetura BDI e em linguagens específicas (e.g. AgentSpeak[Rao 1996], GOAL[Hindriks 2009]). Pode-se explorar também ferramentas e técnicas de programação que explorem o paralelismo e/ou distribuição. Uma vez que a arquitetura utilizada serve de base para explorar modelos causais, é possível também realizar análises estatísticas com base no histórico de mudança de estados. Isso permitiria gerar explicações probabilísticas. Além disso, pode-se também utilizar

[7] Disponível em: `https://github.com/reworkd/AgentGPT`.

simuladores e interfaces gráficas para promover testes mais sofisticados da biblioteca. Outra sugestão seria solicitar explicações por linguagem natural, assim como gerar explicações em linguagem natural.

Referências

[Castelfranchi and Paglieri 2007] Castelfranchi, C. and Paglieri, F. (2007). The role of beliefs in goal dynamics: prolegomena to a constructive theory of intentions. *Synthese*, 155(2):237–263.

[dos Santos Neves and Linhares 2021] dos Santos Neves, F. and Linhares, R. R. (2021). Framework nop 4.0: contribution to the development of applications in the notification oriented paradigm through generic programming. *Institutional Repository of the Federal Technology University – Paraná (RIUT)*.

[Georgeff et al. 1999] Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (1999). The belief-desire-intention model of agency. In Müller, J. P., Rao, A. S., and Singh, M. P., editors, *Intelligent Agents V: Agents Theories, Architectures, and Languages*, pages 1–10, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Grice 1975] Grice, H. P. (1975). *Logic and Conversation*, pages 41 – 58. Brill, Leiden, The Netherlands.

[Harbers et al. 2010] Harbers, M., van den Bosch, K., and 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*, volume 2, pages 125–132.

[Haynes et al. 2009] Haynes, S. R., Cohen, M. A., and Ritter, F. E. (2009). Designs for explaining intelligent agents. *International Journal of Human-Computer Studies*, 67(1):90–110.

[Hindriks 2009] Hindriks, K. V. (2009). *Programming Rational Agents in GOAL*, pages 119–157. Springer US, Boston, MA.

[Jasinski and Tacla 2022] Jasinski, H. M. R. and Tacla, C. A. (2022). Generating contrastive explanations for bdi-based goal selection. *Institutional Repository of the Federal Technology University – Paraná (RIUT)*.

[Morveli-Espinoza et al. 2022] Morveli-Espinoza, M., Nieves, J. C., Tacla, C. A., and Jasinski, H. M. R. (2022). An Argumentation-Based Approach for Goal Reasoning and Explanations Generation. *Journal of Logic and Computation*. exac052.

[Pearl 2009] Pearl, J. (2009). Causal inference in statistics: An overview. *Statistics Surveys*, 3(none):96 – 146.

[Rao 1996] Rao, A. S. (1996). Agentspeak(l): Bdi agents speak out in a logical computable language. In Van de Velde, W. and Perram, J. W., editors, *Agents Breaking Away*, pages 42–55, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Sado et al. 2023] Sado, F., Loo, C. K., Liew, W. S., Kerzel, M., and Wermter, S. (2023). Explainable goal-driven agents and robots - a comprehensive review. *ACM Comput. Surv.*, 55(10).