UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RAFAEL KRAUSE CENCI

BDI4JADE Debugger: um Ambiente de Depuração para Agentes BDI

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Profa. Dra. Ingrid Nunes

Porto Alegre
2016

AGRADECIMENTOS

Gostaria de agradecer primeiramente a Deus, por todas as oportunidades que me deu.

Agradeço aos meus pais, Dario e Valéria, sem os quais eu não teria condições de ter chegado até aqui. Muito obrigado por todo o apoio, carinho e desvelo por me fazer alguém na vida.

Agradeço ao meu irmão Luciano, que me acompanhou desde o começo da faculdade e minha cunhada Elisângela.

Quero agradecer aos meus amigos, especialmente Marcelo Bihre que me ajudou com tanta disponibilidade neste trabalho, Augusto e Pedro.

Agradeço também a Maria e a Miguel.

Agradeço a todos os professores do Instituto de Informática, especialmente à professora Ingrid pela sua orientação para que este trabalho saísse.

A todas as pessoas que, de forma direta ou indireta, colaboraram para que esta etapa fosse realizada, meu sincero obrigado.

RESUMO

A Inteligência artificial vem mostrando que possui um grande potencial para a solução de problemas computacionais. Particularmente, a abordagem de Sistemas Multiagente tem interessado pela sua aplicação a várias áreas. Tanto a academia quanto a indústria têm investido nisso, e já existem diversas plataformas que implementam sistemas multiagente. Podemos citar JADE, JADEX, JAM, 3APL, JACK, entre outras. É importante, agora, que tais implementações sejam adequadas para o desenvolvimento de sistemas multiagente em larga escala. Com essa preocupação, foi criada a plataforma chamada BDI4JADE. Ela caracteriza-se principalmente por prover a arquitetura BDI (*Belief-Desire-Intention*) em uma camada adicional a uma plataforma já existente (JADE) e por ser totalmente escrita em Java. Por isso, é facilmente integrada a qualquer API, biblioteca, *framework* ou IDE utilizados na área industrial para desenvolvimento de software em Java. Este trabalho visa aprimorar o desenvolvimento com BDI4JADE. Quanto ao processo de depuração, não há ainda uma ferramenta adequada. O depurador comumente utilizado com BDI4JADE é o depurador do Eclipse. Ele apresenta muitas informações não relevantes do ponto de vista de uma arquitetura BDI. Assim, este trabalho visa prover uma visualização alternativa das informações, através de uma customização da visão de variáveis fornecida pelo depurador do Eclipse. Foi feito um projeto de visão de variáveis focada na arquitetura BDI e implementado através de um *plugin* para o Eclipse. Dessa forma, a proposta deste trabalho é oferecer uma ferramenta de depuração focada na arquitetura BDI, através da customização do depurador do Eclipse para BDI4JADE, o qual supre uma carência desta e, consequentemente, facilitará o desenvolvimento de sistemas multiagente em larga escala.

Palavras-chave: Agentes BDI. depurador. BDI4JADE. depurador do Eclipse.

ABSTRACT

Artificial Intelligence has shown that it has great potential for solving computational problems. In particular, the Multi-Agent Systems approach is interested in being applicable to several areas. Both academia and industry have invested in this, and there are already several platforms that implement multi-agent systems. We can mention JADE, JADEX, JAM, 3APL, JACK, among others. It is important now that such implementations are suitable for the development of multi-agent systems on a large scale. With this concern, the platform called BDI4JADE was created. It is mainly characterized by providing the Belief-Desire-Intention (BDI) architecture in an additional layer to an already existing platform (JADE) and for being totally written in Java. Therefore, it is easily integrated with any API, library, framework or IDE used in the industrial area for software development in Java. This work aims to improve the development with BDI4JADE. As for the debugging process, there is not yet a proper tool. The debugger commonly used with BDI4JADE is the Eclipse debugger. It presents many information not relevant from the point of view of a BDI architecture. Thus, this work aims to provide an alternative visualization of the information, through a customization of the view of variables provided by the Eclipse debugger. A variables view project focused on the BDI architecture was implemented and implemented through a plugin for Eclipse. In this way, the purpose of this work is to offer a debugging tool focused on the BDI architecture, by customizing the Eclipse debugger for BDI4JADE, which provides a lack of this and, consequently, will facilitate the development of large-scale multiagent systems.

Keywords: BDI agents. debugger. BDI4JADE. Eclipse debugger.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SMA Sistema Multiagente

FIPA Foundation For Intelligent, Physical Agents

JDE JACK Development Environment

BDI Belief-Desire-Intention

DSL Domain Specific Language

JCC JADEX Control Center

SUMÁRIO

1 INTRODUÇÃO

Através do uso de heurísticas e aprendizagem por reforço, a inteligência artificial forneceu um novo paradigma para enfrentar problemas computacionais. Nas últimas décadas, com base em modelos teóricos do raciocínio humano, sistemas inteligentes de software foram sendo desenvolvidos. Sistemas especialistas, *engines* lógicas, sistemas de tomada de decisão, mineradores de dados, simuladores científicos e muitos outras aplicações tornaram-se realidade a partir de então. Há várias abordagens diferentes para soluções de problemas dentro da área da inteligência artificial. Dentre elas, uma que vem sendo muito explorada é a chamada abordagem Multiagente.

Um Sistema Multiagente (SMA) (WOOLDRIDGE, 2009) consiste em um agrupamento cooperativo de unidades de resolução de problemas. Essas unidades de software, chamadas de Agentes Autônomos, são capazes de reproduzir uma inteligência que estabelece e atinge objetivos de forma independente a qualquer instrução humana.

A natureza distribuída dos SMAs permite a modelagem mais adequada de muitos problemas do mundo real, onde se tem diversas entidades, cada qual com suas características particulares, objetivos próprios e muitas vezes conflitantes e também necessidade de cooperação para a obtenção de uma meta. A abordagem multiagente permite decomposição (cada agente pode resolver um pedaço do problema), decentralização (muitos sistemas reais não possuem um topo), abstração (através de uma hierarquia de agentes, agentes de maior nível abstraem ou delegam detalhes para agentes de menor nível), organização (através da comunicação entre os agentes, pode-se estabelecer uma estratégia de atuação) e flexibilidade (agentes podem mudar de atuação em tempo de execução). Outras motivações para o uso de SMAs estão relacionadas com permitir a interoperabilidade entre sistemas legados e permitir uma maior interação humano-computador onde os dois participam como agentes do sistema.

SMAs possuem aplicações em muitos domínios e têm provocado muito interesse tanto na área acadêmica quanto na área industrial. Existem muitas ferramentas e metodologias para desenvolvimento de SMAs. Há diversas APIs e *frameworks*, como JADE (BELLIFEMINE; CLAIRE; GREENWOOD, 2007), dMARS (D'INVERNO et al., 1997) e JAM (STOLFO et al., 1997), além de um padrão internacional: o FIPA (Foundation For Intelligent, Physical Agents) (IEEE, 2005). Algumas delas, implementam a arquitetura BDI, modelo proposto por Bratman (BRATMAN, 1987), a qual é considerada uma das melhores formas de modelagem de agentes com raciocínio (GEORGEFF et al., 1999).

Exemplos de plataformas de agentes que implementam a arquitetura BDI incluem Jason (BORDINI; HüBNER; WOOLDRIDGE, 2007), JACK (JACK, 2015), JADEX (JADEX, 2016) e a plataforma 3APL (3APL, 2005). Em particular, estas quatro plataformas são baseadas na linguagem Java. Entretanto, embora a linguagem subjacente consista em uma linguagem de programação de propósito geral, os agentes são implementados nessas plataformas em uma nova linguagem de programação. Diferentemente, BDI4JADE (NUNES; LUCENA; LUCK, 2011) não utiliza uma nova linguagem. Ela implementa uma arquitetura BDI como uma camada de software acima da plataforma JADE, o qual baseia-se apenas em Java. Assim, *features* próprias do Java, como anotações e reflexão podem ser exploradas no desenvolvimento de sistemas mais complexos. Além disso, outras tecnologias importantes para o desenvolvimento de aplicações industriais de larga escala como, por exemplo, bibliotecas para gerenciamento de banco de dados, podem ser facilmente integradas (NUNES; LUCENA; LUCK, 2011)

Uma ferramenta importante para o desenvolvimento de aplicações de larga escala é o depurador. Um depurador é um programa voltado para encontrar defeitos em software. Ele fornece diversos utilitários para visualizar o fluxo do programa, o estado interno das variáveis em tempo de execução e até o seu uso de memória. BDI4JADE é capaz de utilizar o depurador presente nas IDEs Java. Porém, seria interessante poder utilizar um depurador customizado, já que a ferramenta padrão mostrará todas as variáveis do programa BDI4JADE. Como a plataforma é bastante extensa, há muita informação irrelevante para o desenvolvedor, que pode dificultar a tarefa de depuração.

A proposta deste trabalho é oferecer um ambiente de depuração com possibilidade de visualização do estado interno dos agentes onde sejam apresentadas as informações mais relevantes, principalmente as específicas da aplicação, considerando a programação de agentes na ferramenta BDI4JADE. A IDE escolhida para isso foi o Eclipse, a qual permite, através do desenvolvimento de um *plugin*, estender o seu depurador nativo.

A ferramenta desenvolvida neste trabalho tem boa utilidade para desenvolvedores de SMAs com agentes BDI, pois foca especialmente mas suas necessidades e fornece uma solução que poucas ferramentas possuem. Por trazer benefícios aos desenvolvedores que utilizam a ferramenta BDI4JADE, e por ser essa uma ferramenta voltada para o desenvolvimento industrial, este trabalho poderá ser utilizado posteriormente, não ficando apenas no âmbito acadêmico.

Essa monografia está dividida em cinco capítulos. No Capítulo 1 abordaremos os conceitos de agentes, sistemas multiagentes e arquiteturas BDI, fornecendo uma breve

fundamentação teórica para situar o leitor no contexto específico do problema a ser resolvido. Falaremos também da plataforma BDI4JADE que é uma implementação para arquiteturas BDI de agentes. No Capítulo 2 falaremos do depurador proposto para o BDI4JADE, abordando suas funcionalidades, as tecnologias utilizadas para o seu desenvolvimento e os detalhes de implementação. No Capítulo 3 falaremos dos resultados obtidos, apresentando casos de uso da aplicação. No Capítulo 4 falaremos de trabalhos relacionados, fazendo um comparativo com outros depuradores como Jason, JACK, entre outros. No Capítulo 5 apresentamos a conclusão, projetando possíveis trabalhos futuros.

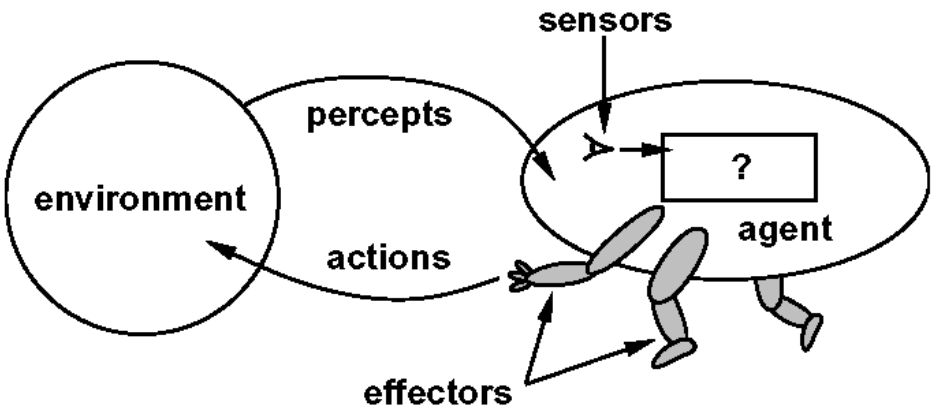
2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está divido em três partes. Na primeira parte, é discutido o que são os agentes inteligentes, entidade básica deste trabalho. Na segunda parte, é introduzida a arquitetura BDI de agentes, o qual é uma evolução do agente puro para uma organização baseada em uma teoria do raciocínio humano. A terceira parte descreve a plataforma utilizada para implementar agentes BDI.

2.1 Agentes Inteligentes

Russell e Norvig (RUSSELL; NORVIG, 2004) definem um agente como um sistema capaz de perceber através de sensores e agir em um dado ambiente através de atuadores.

Figura 2.1: Agente segundo definição de Russel e Norvig



Fonte: (ZAMBIASI, 2011)

Além disso, muitos autores entram em consenso no que se refere a algumas características principais que um sistema deve possuir para consistir em um agente:

- **Autonomia:** independência de outros agentes ou de seres humanos para o controle do seu estado interno e para tomada de decisões.
- **Reatividade:** capacidade de perceber o ambiente no qual está inserido e reagir em consequência.
- **Pró-atividade:** capacidade de tomar iniciativa de acordo com seus próprios objetivos quando julgar necessário.

- **Sociabilidade:** capacidade para troca de informações com outros agentes do sistema a fim de atingir seus próprios objetivos ou de ajudar outros agentes. Vale ressalvar que, pelo fato de serem considerados entidade autônomas, é possível que um certo agente não "deseje" se comunicar com um outro específico e ignore a mensagem.

Ser um agente significa ser uma entidade ativa, capaz de produzir suas próprias ações (Autonomia) de acordo com objetivos próprios. Para isso, é necessário que o agente consiga analisar o ambiente em que se situa e, eventualmente, interagir com outros agentes em negociações a fim de conquistar benefícios. Um agente é uma entidade de software diferente de um objeto. Enquanto um objeto é passivo, e recebe chamadas, um agente é ativo e tomas suas próprias decisões. Um agente é capaz de produzir conhecimento, e, por isso, lidar com situações imprevistas. Um objeto não é capaz de lidar com parcialidade dos dados.

O raciocínio é o aspecto mais importante que distingue um agente dito inteligente dos outros agentes. Afirmar que um agente tem raciocínio significa dizer que ele tem a capacidade de analisar e inferir baseando-se no seu conhecimento atual e nas suas experiências. Esse raciocínio pode ser:

- **Baseado em regras:** onde eles usam um conjunto de condições prévias para avaliar as condições no ambiente externo.
- **Baseado em conhecimento:** onde eles têm à disposição grandes conjuntos de dados sobre cenários anteriores e ações resultantes, dos quais eles deduzem seus movimentos futuros.

Um sistema pode ser monoagente (ex.: um gerenciador de emails) ou multiagente (ex.: um simulador de tráfego). Em um sistema multiagente, os agentes podem ser competitivos ou colaborativos. Agentes colaborativos buscam alcançar os seus próprios objetivos e também os dos outros agentes. Assim, uma solução que beneficia um agente ajuda também outros agentes. Um exemplo é um SMA que controla a linha de produção de uma fábrica. Agentes competitivos possuem conflitos de interesses, como, por exemplo, em um jogo onde há oponentes.

SMAs têm aplicações em muitos domínios. A seguir são descritas algumas áreas que fazem uso de Agentes Inteligentes:

- **Sistemas de tráfego:** agentes são utilizados para automatizar veículos, semáforos e outras entidades de tráfego.

- **Sensoriamento distribuído:** agentes podem ajudar a fornecer informações mais precisas quando cada agente é responsável por um sensor.
- **Controle e automação:** Agentes podem automatizar processos industriais e tomar decisões na presença de algo imprevisto, como mudanças no cronograma de produção, quebra de máquinas, atualização dos equipamentos, etc.
- **Simuladores:** agentes podem simular pessoas, corpos ou qualquer entidade que necessite de comportamento específico. Pode-se utilizar em filmes ou jogos para modelar atores, em simulações de corpos estelares ou outras simulações de eventos físicos, simulações de *traders* em mercado de ações, de emergências, etc.
- **Gerenciamento e Recuperação de informações:** Um agente pode ser um assistente pessoal de emails. Pode também auxiliar usuários que necessitam de ajuda em um site como um sistema especialista.
- **Tomada de decisões e negociação:** Agentes auxiliando humanos em tomadas de decisão, como, por exemplo, na área de *e-commerce* em compras e vendas de produtos. Um agente pode recomendar produtos, comparar ofertas em busca da melhor, negociar a compra com um determinado vendedor, fazer todas os procedimentos para o pagamento da compra.

2.2 Agentes BDI

Existem diferentes tipos de agentes inteligentes. Pode-se classificá-los segundo sua arquitetura. De forma mais geral, há três grandes categorias de agentes inteligentes:

- **Agentes com Arquiteturas Reativas:** são agentes com implementação mais simples, pois seguem um conjunto de regras para tomada imediata de ações. Eles não possuem representação do ambiente, nem guardam histórico de suas ações. Seu comportamento é baseado unicamente no estímulo que recebem a cada instante.
- **Agentes com Arquiteturas Deliberativas:** possuem modelo de raciocínio para tomada de decisões.
- **Arquiteturas Híbridas:** combinam as características das duas abordagens anteriores.

Uma arquitetura que pode ser, nas suas variações, tanto deliberativa como híbrida e que assume particular destaque na literatura da especialidade é a arquitetura BDI (“*Be-*

lief-Desire-Intention"). O nome BDI provém do fato de essa arquitetura estar baseada em um modelo que reconhece a primazia de crenças (*beliefs*), desejos (*desires*) e intenções (*intentions*) na ação racional (WOOLDRIDGE, 2000). Este modelo tem sua origem na teoria da ação racional desenvolvida por Bratman (BRATMAN, 1987). Essa teoria busca compreender o raciocínio prático em humanos. Por racionínio prático entende-se o problema de ponderar considerações conflitantes, onde as considerações relevantes são providas pelo que o agente deseja e sobre o que ele acredita. Assim, podemos distinguir duas atividades importantes do raciocínio prático: decidir o que se quer alcançar e decidir como alcançar. A primeira etapa é denominada deliberação e a segunda etapa raciocínio.

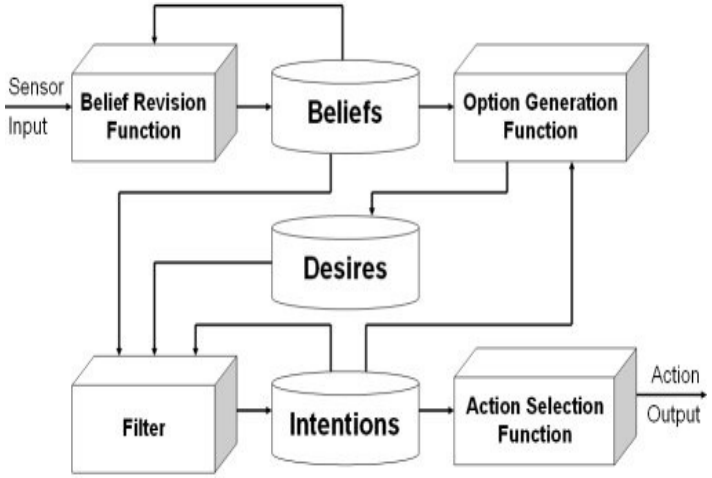
Este raciocínio prático é, como apresentado acima, descrito por três estados mentais:

- **Crenças (*Beliefs*) –** Tudo o que o agente conhece e acredita a respeito do ambiente. Podem ser informações que ele obtém através de sensores, ou interação com outros agentes. Tais informações podem ser parciais ou até estarem erradas.
- **Desejos (*Desires*) –** Possíveis objetivos ou estados a serem alcançados. Os desejos são o que motivam um agente a agir.
- **Intenções (*Intentions*) –** Dentre os diversos desejos, considerando as circunstâncias e restrições do ambiente e do agente, somente alguns podem ser alcançados em um dado momento. As intenções são as atuais metas buscadas pelo agente. São um subconjunto dos desejos.

Rao e Geogeff (RAO; GEORGEFF, 1995) adotaram o modelo BDI para agentes de software e desenvolveram a arquitetura BDI, apresentando uma teoria formal e um interpretador BDI. A Figura 2.2 apresenta a arquitetura de um interpretador BDI. Além dos três componentes anteriormente citados, a arquitetura comporta ainda os seguintes elementos:

- **Função de revisão de crença -** As informações a respeito do ambiente são atualizadas para cada agente.
- **Função geradora de opções -** Lista possíveis objetivos que um agente pode querer alcançar (desejos).
- **Filtro -** Seleciona, dentre todos as opções disponíveis, qual desejo será atualmente buscado (intenção).
- **Função de seleção de ação -** Decide qual algoritmo usar para alcançar determinada intenção.

Figura 2.2: Arquitetura BDI



Fonte: (WOOLDRIDGE, 1999)

Existem diversas linguagens e plataformas que implementam sistemas BDI. Um breve resumo das principais tecnologias existentes pode ser visto nas Tabela 2.1 e 2.2.

Tabela 2.1: Linguagens de programação BDI

Linguagens de Programação BDI	
Linguagem	**Descrição**
AgentSpeak	Linguagem puramente abstrata de paradigma lógico
JACK Agent Language	Extensão de Java orientada a agentes
3APL	Baseada em paradigma lógico
Goal	Linguagem declarativa

Tabela 2.2: Plataformas de agentes BDI

Algumas plataformas de agentes BDI	
Plataforma	**Descrição**
PRS	Primeiro sistema BDI.
dMars	Sucessor do PRS, baseado em C++
JACK	Uma das mais recentes extensões do PRS
JAM	Plataforma em Java, baseado em BDI e PRS
JADEX	Agentes são escritos em Java e XML
Jason	Interpretador da linguagem AgentSpeak
3APL	Interpretador da linguagem 3APL

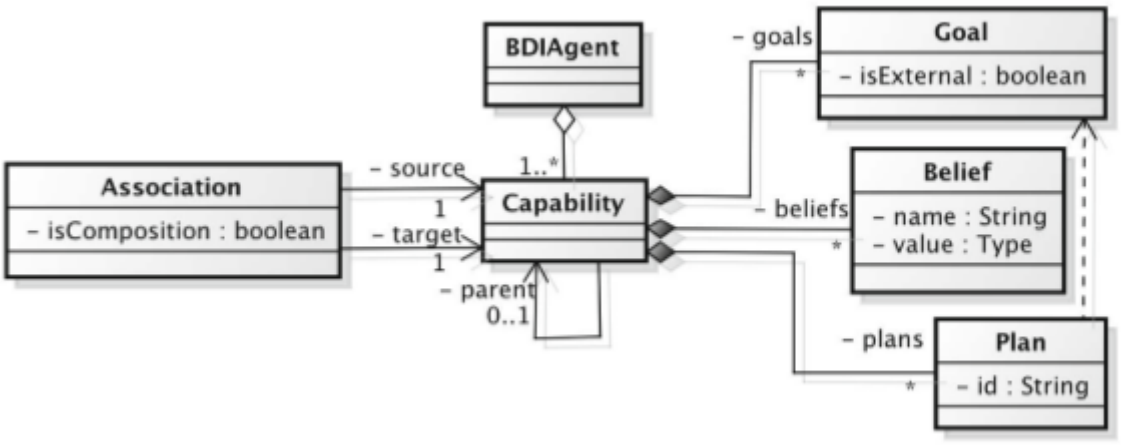
2.3 BDI4JADE

Diversas plataformas implementam sistemas multiagentes com arquitetura BDI. Entretanto, torna-se dificil a integração com bibliotecas e *frameworks* quando essas plataformas se utilizam de outras linguagens para definição dos agentes, tais como XML ou Prolog. Muitos *frameworks* necessitam fazer chamadas para as classes específicas da aplicação, e é impossível fazê-lo se não se tem classes JAVA ou semelhantes.

A motivação para a construção de BDI4JADE é facilitar a integração de SMAs de arquitetura BDI com ferramentas necessárias para o desenvolvimento de sistemas industriais de grande escala (NUNES; LUCENA; LUCK, 2011). Além disso, como sua implementação consiste puramente em JAVA, torna-se mais fácil fazer modificações no modelo dos agentes sem a necessidade de alterar como arquivos fonte são processados ou compilados.

BDI4JADE (NUNES; LUCENA; LUCK, 2011) é uma extensão da plataforma JADE a qual consiste em uma plataforma de agentes construída em JAVA e que não utiliza nenhuma DSL. BDI4JADE constrói uma camada sobre o JADE que implementa a arquitetura BDI de agentes inteligentes. Um diagrama com as principais classes pode ser visto na Figura 2.3.

Figura 2.3: Principais Classes



Fonte: (NUNES, 2014)

Os principais componentes de BDI4JADE são:

- **BDIAgent -** Agentes BDI são extensões da classe `Agent` do JADE. Como no JADE, cada *BDIAgent* tem sua própria thread de execução. Um *BDIAgent* é com-

posto de um conjunto de *Intentions* e de *Capabilities*. *Beliefs* e *plans* não são partes de um agente diretamente, mas de suas *capabilities*.

- **Capability -** Agentes são agregados de *capabilities*. Estas são módulos que implementam todo o ciclo de vida do agente em função de uma única tarefa alvo. Assim, por exemplo, um agente para varrer pode ter uma *capability* que tem como objetivo a movimentação do agente no ambiente e outra que tem como objetivo limpar a sujeira. Uma *capability* é composta de um conjunto de *beliefs* e de *plans*. Não há uma declaração explícita de *goals* em *capabilities*. Isso acontece em tempo de execução. *Capabilities* podem estar relacionadas com outras *capabilities* por associação, composição ou generalização/especialização.
- **Beliefs -** *Beliefs* podem armazenar qualquer tipo de informação e estão associadas a um nome. A classe `BeliefBase` fornece métodos para manipular *beliefs*, tais como adicionar, remover e atualizar. *Beliefs* possuem um valor e um nome. A classe `Belief` é abstrata e não determina nenhuma forma de armazenamento específica.
- **Goals -** *Goals* podem ser quaisquer classes JAVA, desde que implementem a interface `Goal`. Existe um conjunto de *goals* pré-definidos que podem ser usados em aplicações.
- **Intentions -** *Goals* são, internamente, tratados pela plataforma como *intentions*. Quando um *goal* é adicionado a um agente, uma *intention* é criada e associada com ele. *Intentions* possuem um estado, que pode ser: (i) *Achieved* - O *Goal* associado com a *Intention* foi alcançado; (ii) *No longer desired* - o *Goal* associado com a *Intention* não é mais buscado; (iii) *Plan Failed* - o agente está tentando alcançar o *Goal* associado com a *Intention*, mas a última execução do plano falhou; (iv) *Try to achieve* - o agente está tentando alcançar o *Goal* associado com a *Intention*, executando um plano para alcançá-lo; (v) *Unachievable* - todos os planos foram executados para tentar alcançar o *Goal* associado com a *Intention*, mas todos falharam.
- **Plans -** Planos a serem executados são instâncias da classe `Behavior` do JADE, e, por isso, são escalonados pelo JADE. Existem duas principais classes associadas com planos. `Plan` é uma classe que dá informações a respeito do plano. Ela informa o id do plano, a biblioteca de planos à qual ele pertence, quais *goals* ele é capaz de alcançar e quais mensagens ele pode processar. Planos podem ser executados por instâncias de `PlanBody`. Esta é uma classe que estende `Behavior` do JADE.

- **Messages -** Mensagens são enviadas e recebidas como no JADE. Adicionalmente, BDI4JADE fornece uma classe que implementa um receptor de mensagens, chamada `BDIAgentMsgReceiver`, que é um *behavior* que captura mensagens e verifica se o agente possui algum plano capaz de processá-la. Se sim, cria um *goal* do tipo `MessageGoal` contendo a mensagem recebida.
- **Events -** Há dois tipos de eventos: eventos de *beliefs* e de *goals*. *Belief listeners* são associados com bases de *beliefs* e sempre que uma *belief* é alterada, adicionada ou removida, a base de *beliefs* é notificada. *goal listeners* são associados com *intentions* e são usados para observar mudanças no estado da *intention*.

3 BDI4JADE DEBUGGER

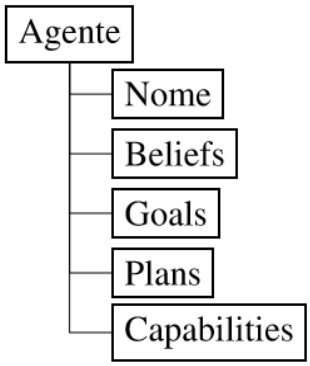
Neste capítulo é abordado o software desenvolvido. São apresentadas as suas funcionalidades, tecnologias utilizadas, detalhes de implementação e dificuldades encontradas no seu desenvolvimento.

3.1 Funcionalidades

Por não ser uma aplicação isolada, mas integrada ao Eclipse, a ferramenta desenvolvida possui todas as funcionalidades já existentes no depurador do Eclipse. O BDI4JADE Debugger oferece também uma visão de variáveis personalizada para a visualização de agentes do BDI4JADE. Sempre que o desenvolvedor estiver depurando um método de algum objeto que está dentro do contexto de um agente, são apresentadas as informações relevantes daquele agente em uma hierarquia adequada. Assim, para um dado agente, são mostradas suas informações conforme as Figuras 3.1, 3.2 e 3.3.

Dentro de um agente são mostrados o seu nome, suas *beliefs*, *goals*, *plans* e *capabilities*:

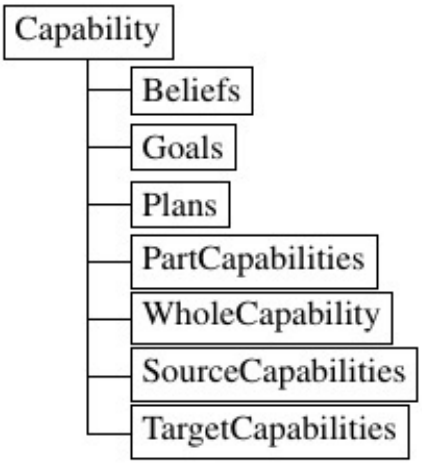
Figura 3.1: Estrutura para um agente



Fonte: Autor

Dentro de uma *capability* de um agente são mostradas além de suas *beliefs*, *goals* e *plans*, as suas relações com outras *capabilities*, ou seja, *partCapabilities*, *wholeCapabilities*, *sourceCapabilities* e *targetCapabilities*:

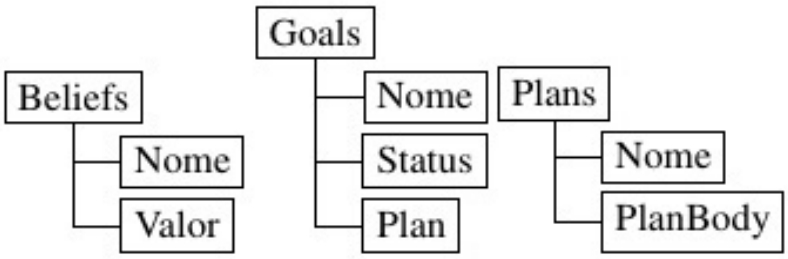
Figura 3.2: Estrutura para uma Capability de um Agente



Fonte: Autor

Dentro das *beliefs* são mostrados o seu nome e seu valor. Dentro de *goals* são mostrados o seu nome, o *plan* que está buscando ou tentou alcançá-lo e o seu *status*. Dentro de *plans* são mostrados o seu nome e o *planBody* (a instância da classe que sabe como executar o *plan*):

Figura 3.3: Estrutura para Beliefs, Goals e Plans



Fonte: Autor

3.2 Tecnologias Utilizadas

A arquitetura do Eclipse oferece a possibilidade de adicionar funcionalidade à plataforma através de *plugins*. Pode-se criar diferentes visões, componentes de interface, perspectivas, entre outros. Existe na IDE do Eclipse um tipo de projeto para *plugins*,

semelhante ao projeto java, onde são criados classes, arquivo XML descrevendo informações de nome, id, versão, dependências e toda estrutura de arquivos necessária para o *plugin*. Um novo *plugin* é uma extensão integrada ao sistema através de pontos de extensão. Esses pontos de extensão definem um contrato através de interfaces e marcações XML o qual deve ser implementado para a criação do *plugin*. A Figura 3.4 mostra um exemplo de uma extensão para criar uma nova visão para o Eclipse. É necessário definir alguns atributos como o nome, ícone da visão, id e uma classe a qual possuirá a implementação do comportamento da visão.

Figura 3.4: Definição de uma Extensão

```
<extension point="org.eclipse.ui.views">
   <category
      id="com.xyz.views.XYZviews"
      name="XYZ"/>
   <view
      id="com.xyz.views.XYZView"
      name="XYZ View"
      category="com.xyz.views.XYZviews"
      class="com.xyz.views.XYZView"
      icon="icons/XYZ.png"/>
</extension>
```

Fonte: (ECLIPSE, c)

Este trabalho utiliza dois pontos de extensão: `org.eclipse.ui.views` e `org.eclipse.debug.core.logicalStructureTypes`. Como foi visto acima, o primeiro é utilizado para a criação de uma nova visão no ambiente. Assim, foi criada uma visão chamada BDI4JADE variables e posicionada ao lado da visão de variáveis do depurador. Essa nova visão é, inicialmente, uma cópia da visão de variáveis do depurador. O segundo ponto foi utilizado para a definição de uma nova hierarquia de variáveis na visão BDI4JADE variables. Esse ponto de extensão permite definir uma nova hierarquia lógica de variáveis diferente da estrutura física dessas variáveis em memória.

3.3 Detalhes de Implementação

Para cada ponto de extensão, foi necessário criar uma classe que implementasse a interface requerida. A classe que implementou a visão foi uma extensão da classe `VariablesView` da plataforma de depuração do Eclipse. Isso foi o suficiente para

se obter uma visão de variáveis igual ao do depurador do Eclipse.

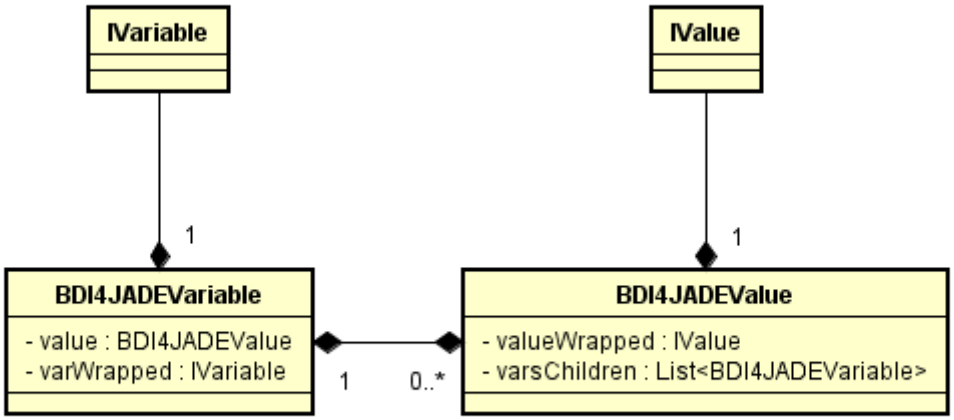
Para implementar a estrutura lógica, foi criada uma classe que possui a responsabilidade de, dado uma variável da visão de variáveis, retornar uma nova variável reestruturada. A cada *breakpoint* da depuração, o Eclipse fornece para a classe a variável que será mostrada. Essa variável está organizada em uma estrutura de árvore. Assim, se é uma variável referente a um objeto, ela conterá nos seus filhos todas as a propriedades do objeto. A classe então processa a árvore, construindo uma nova hierarquia que será retornada para ser apresentada.

3.3.1 Estrutura de dados

Todas as variáveis de uma aplicação no Eclipse, quando é depurada, são representadas por duas interfaces principais: `IVariable` e `IValue`. `IVariable` representa uma variável e `IValue` representa o valor de uma variável. `IValue` pode possuir tipos simples como inteiros e *strings* ou tipos complexos como *arrays* e objetos. Essas classes podem ser implementadas de diferentes maneiras, dependendo da linguagem sendo utilizada. Java, especificamente, implementa essas interfaces com várias classes: `JDIVariable`, `JDIValue`, `JDIField`, `JDIObjectReference`, entre outras.

A estrutura de dados do BDI4JADE Debugger consiste nas seguintes classes: `BDI4JADEVariable` e `BDI4JADEValue`. Elas são *wrappers* para `IVariable` e `IValue`, ou seja, classes contêiner de `IVariable` e `IValue`. Assim, é possível para o BDI4JADE Debugger utilizar toda a estrutura subjacente sem a necessidade de recriar classes para todos os tipos de variáveis e valores, e, ao mesmo tempo, controlar quem aparece na hierarquia da árvore apenas fazendo substituições entre os valores e variáveis contidos nas classes contêiner.

Figura 3.5: Estrutura de Dados do BDI4JADE Debugger



Fonte: Autor

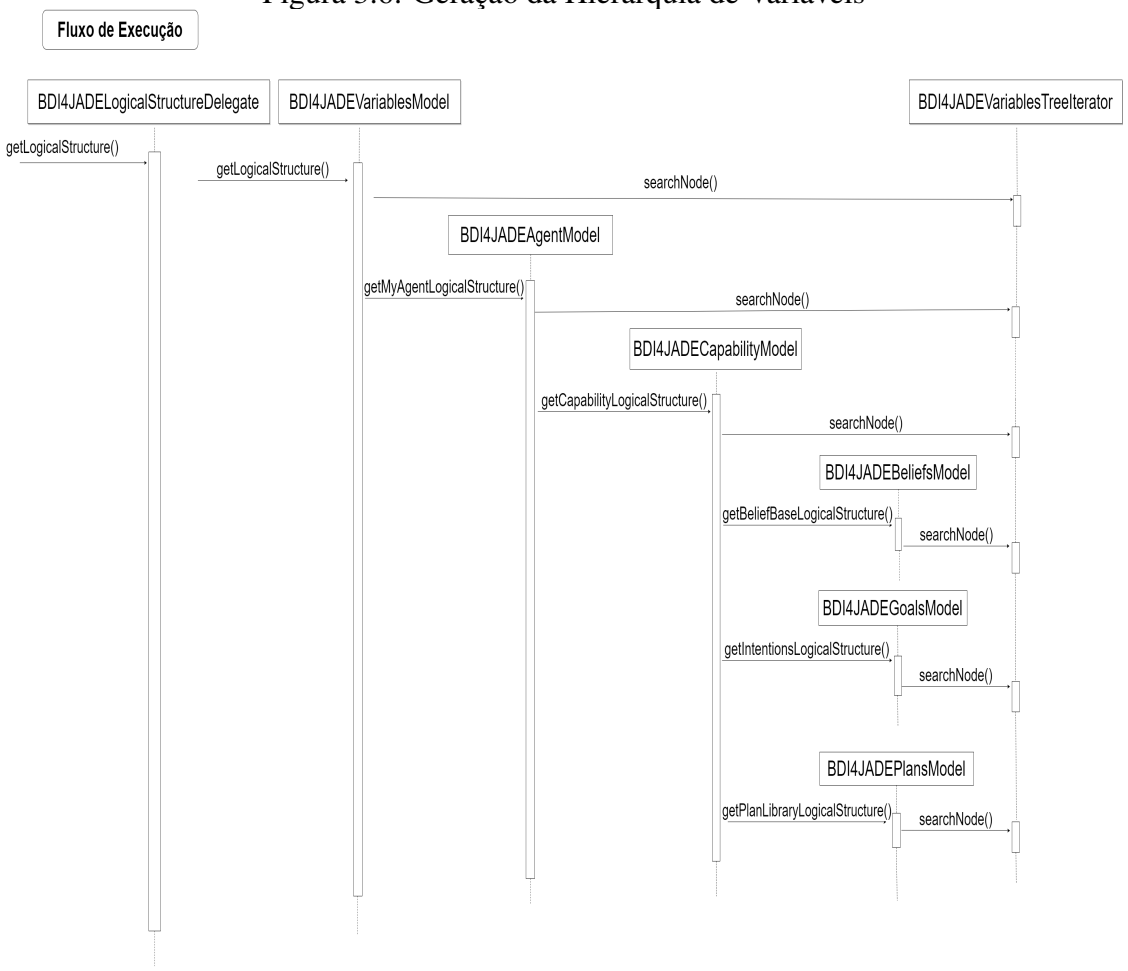
Um detalhe que vale ressaltar é que foi necessário ter cuidado com os ciclos de referências entre as variáveis. Quando o depurador construia as variáveis, criando os *wrappers* de cada uma, sua execução entrava em um laço infinito. Por isso, a cada dois níveis de variáveis foi estabelecida uma pausa na construção. Somente quando o usuário do depurador clicar em uma variável do nível mais profundo, é que serão expandidos os próximos dois níveis e assim sucessivamente.

3.3.2 Fluxo de Execução

A cada vez que é processado, o depurador envia para o método chamado *getLogicalStructure* da classe `BDI4JADELogicalStructureDelegate` o nó raiz da hierarquia de variáveis que irá mostrar para o usuário. Este nó raiz pode ser tanto um agente, como uma variável que não tem a ver com agentes BDI, ou uma variável de granularidade mais fina como uma *belief* que está dentro do contexto de um determinado agente. O `BDI4JADELogicalStructureDelegate`, por sua vez, envia essa variável para um *model* que irá retornar a variável com uma nova hierarquia. As classes do tipo *model* são classes responsáveis por conhecer a organização da estrutura de variáveis que devem ser retornadas. Se o nó raiz for um agente ou for algo dentro do contexto de um agente, o *model* divide a tarefa em outros *models* menores responsáveis por tratar pequenas porções

da hierarquia. Desse modo, existe um *model* específico para tratar a hierarquia das *beliefs*, outro para tratar *goals*, outro para tratar *plans*, outro para tratar *capabilities* e outro para tratar o agente. Cada um desses *models* chama uma classe utilitária para navegar dentro da árvore. A Figura 3.6 apresenta um diagrama de atividades que ilustra o fluxo de execução acima descrito.

Figura 3.6: Geração da Hierarquia de Variáveis



Fonte: Autor

3.3.3 Principais classes

Foram criadas, ao todo, dezoito classes. São seis as classes que podemos considerar como as principais: `BDI4JADELogicalStructureDelegate`, `BDI4JADEValue`, `BDI4JADEVariable`, `BDI4JADEVariablesModel` , `BDI4JADEVariables-View` e `BDI4JADEVariablesTreeIterator`. Abaixo, segue uma descrição de cada uma delas:

- `BDI4JADEVariablesView` - classe que implementa toda a parte gráfica da visão. É igual a `VariablesView` do depurador padrão do Eclipse.
- `BDI4JADELogicalStructureDelegate` - faz a comunicação com a plataforma do Eclipse e delega o processamento para a `BDI4JADEVariablesModel`. Esta é quem tem conhecimento da estrutura que deve ser retornada. Assim há uma separação de conceitos, onde a camada que lida com a hierarquia fica isolada da camada que lida com a interface do ponto de extensão.
- `BDI4JADEVariablesModel` - classe que tem o conhecimento da hierarquia de variáveis que deve ser mostrada. Ela é responsável por construir toda a hierarquia a partir de uma variável. Ela divide a sua tarefa em outros *models* capazes de lidar com pedaços menores da hierarquia. As classes que ela utiliza são as seguintes: `BDI4JADECapabilityModel`, `BDI4JADEAgentModel`, `BDI4JADEBeliefsModel`, `BDI4JADEPlansModel` e `BDI4JADEGoalsModel`.
- `BDI4JADEVariable` - representa uma variável do BDI4JADE. Esta classe é um contêiner de uma variável do depurador padrão do Eclipse. Ela contém um objeto que representa uma variável java e que implementa a interface `IVariable` (interface que define o comportamento genérico de qualquer variável do depurador do Eclipse). Através de `BDI4JADEVariable`, é possível alterar o nome da variável a qual é mostrada na visão de variáveis, alterar o valor dela, obter seu tipo, entre outros. Muitas das operações são implementadas pela variável contida em `BDI4JADEVariable`.
- `BDI4JADEValue` - representa um valor de uma variável do BDI4JADE. Esta classe também é um contêiner de um valor do depurador padrão do Eclipse. Ela contém um objeto que representa um valor e que implementa a interface `IValue` (interface que define o comportamento genérico de qualquer valor do depurador do Eclipse). Cada valor `BDI4JADEValue` possui variáveis `BDI4JADEVariable` filhas. Assim, no momento de buscar os filhos de um nó da hierarquia, o BDI4JADE debugger busca os filhos de um `BDI4JADEValue` em vez de buscar os filhos de um valor `IValue`. Isso é o que permite ao depurador alterar a estrutura da hierarquia, pois dentro de um objeto `IValue` as variáveis filhas não possuem acessibilidade pública para alteração, enquanto que em um objeto `BDI4JADEValue` elas possuem acessibilidade pública.
- `BDI4JADEVariablesTreeIterator` - classe utilitária para percorrer e bus-

car nodos dentro da hierarquia.

3.3.4 Manutenibilidade e Extensibilidade

O BDI4JADE Debugger não é uma aplicação com fins exclusivamente acadêmicos, mas será utilizada posteriormente, e, por esse motivo, poderá passar por alterações e evoluções. Portanto, é importante entender como a implementação está modularizada e de que forma é possível realizar nela modificações e extensões.

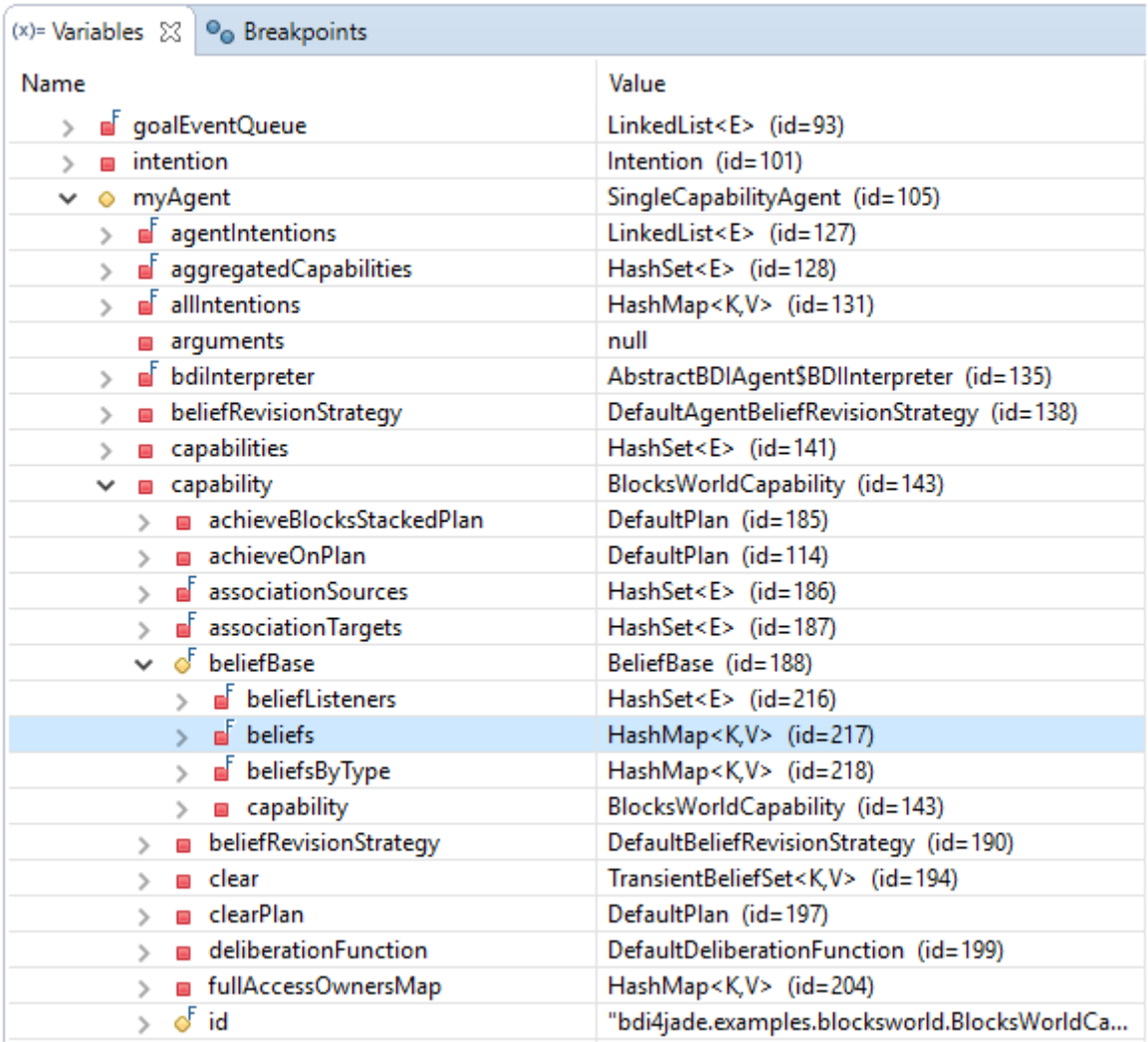
A modularidade do código se dá através dos diversos *models* os quais manipulam e dividem o conjunto de variáveis da hierarquia em pequenos subconjuntos. Assim, quando for necessário adicionar ou remover uma nova sub-hierarquia, basta adicionar ou remover o *model* correspondente.

Para se realizar uma alteração na hierarquia é necessário saber ainda alguns outros detalhes:

- Todas as variáveis que aparecem na visão do BDI4JADE são variáveis reutilizadas da visão de variáveis do Eclipse. Nunca são criadas novas variáveis. O que deve ser feito é uma alteração do nome ou do valor da variável da visão do Eclipse para a variável da visão do BDI4JADE, através dos métodos *setName* e *setValue* de `BDI4JADEVariable`. Assim, por exemplo, para a criação da variável *beliefs* na hierarquia do BDI4JADE, a variável *beliefsRevisionStrategy* é guardada dentro de um objeto `BDI4JADEVariable` e o seu nome é atribuído com *beliefs*.
- Mudanças nos nós filhos são necessárias. Por isso, cada valor `BDI4JADEValue` possui um *array* `List<BDI4JADEVariable> variables`. Esse *array* substitui os nós filhos da variável reutilizada da visão do Eclipse. Assim, por exemplo, para a variável *beliefs* da hierarquia do BDI4JADE, são buscados seus nós filhos através de um método do *model* responsável por manipular *beliefs*, e são inseridos no *array* de nós filhos da BDI4JADEVariable *beliefs*.
- É importante a utilização do método *resetChildren* da classe `BDI4JADEValue`. Ele substitui os nós filhos de uma variável pelos nós passados por parâmetro. Se nenhum nó é passado por parâmetro, ele remove todos os filhos. Isso é utilizado para eliminar variáveis irrelevantes como, por exemplo, `mobHelper` ou `msgQueue` dentro de um agente. É usado também para trazer variáveis de hierarquias mais profundas na árvore para hierarquias mais altas, eliminando hierarquias interme-

diárias. Para isso, basta passar como parâmetro os nós da hierarquia mais profunda para o método *resetChildren* da variável que está em um nível acima do novo nível desejado para eles. Um exemplo de uso aparece dentro da variável *capability*, conforme a Figura 3.7, onde é necessário remover *beliefBase* para mostrar *beliefs*.

Figura 3.7: Exemplo de uso para o método resetChildren



Fonte: Autor

3.4 Dificuldades Encontradas

Uma alternativa ao uso de pontos de extensão explorada no desenvolvimento deste trabalho é implementar a nova hierarquia de variáveis alterando diretamente a estrutura de dados utilizada pelo Eclipse. É possível obter o código que implementa a visão de

variáveis e reutilizá-lo. Nele é criada uma árvore e, com isso, é possível mudar tal código para que gere os nós de outra maneira. Essa é uma solução pouco viável pelos seguintes motivos:

- **Complexidade da estrutura de classes:** Pelo fato de o depurador ser uma ferramenta bastante complexa, com muitos comportamentos, e, talvez numa tentativa de criar uma IDE bastante configurável e estensível, o Eclipse possui uma enorme estrutura de classes, as quais muitas delas são bastante genéricas e outras muitas são classes intermediárias que apenas delegam funcionalidades.
- **Documentação:** É difícil encontrar a documentação necessária para fazer essas alterações. Há muita documentação sobre os pontos de extensão, porém quase nenhuma sobre a estrutura de classes do próprio depurador do Eclipse. Também não há nenhum outro exemplo de código de alguém que tenha feito algo parecido.

Outra dificuldade encontrada, porém de menor tamanho, foi a ocorrência de estouros de memória. Como é gerada uma hierarquia de variáveis que pode possuir muitos níveis de profundidade, é necessário colocar um limite na sua geração. Também pode haver ciclos de referências entre as variáveis, o que pode ocasionar um laço infinito na hora de construir a hierarquia. Para isso foi criado um atributo dentro de `BDI4JADEValue` chamado *canExpand* que controla se, no dado momento, é permitido ou não gerar os nós filhos da variável daquele valor. Ao gerar o nó raiz, ele gera apenas os seus filhos e os filhos dos seus filhos.

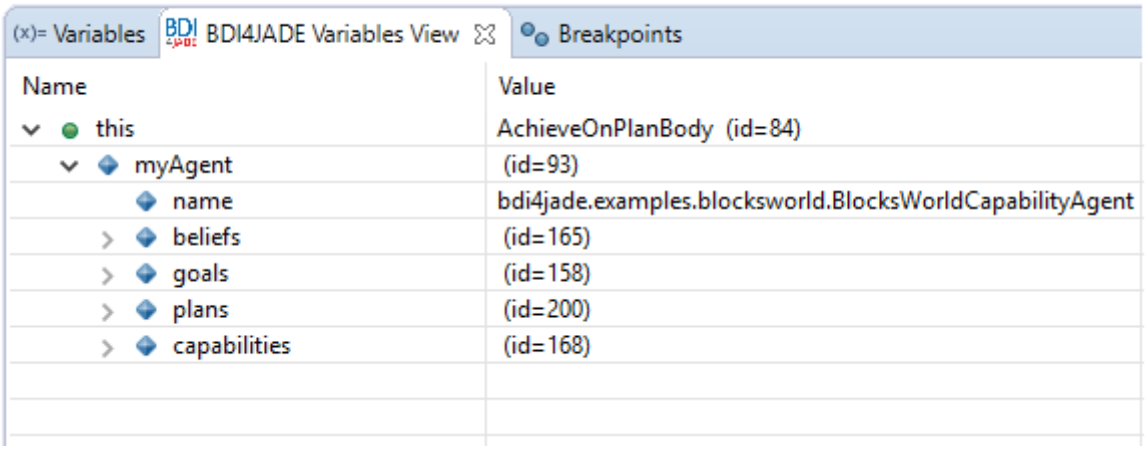
4 RESULTADOS

Neste capítulo são apresentadas algumas imagens dos resultados obtidos. Com a redução das informações apresentadas, a visão de variáveis do BDI4JADE mostrou ser muito mais simples e legível para o desenvolvedor. Para uma mesma execução de um programa que utiliza BDI4JADE, são mostradas uma imagem da visão do depurador do BDI4JADE e outra imagem da visão do depurador do Eclipse. Primeiramente é apresentado como ficou a visualização do topo da hierarquia de variáveis, ou seja, é visto como ficou a visualização do um agente como um todo. Depois, é apresentado o segundo nível da hierarquia de variáveis. Para simplificação, é mostrada apenas a visualização de *beliefs*, porém o mesmo ocorre para a visualização de *plans* e *goals*. Após, é mostrada a visualização de *capabilities*.

4.1 Visualização do Agente

A Figura 4.1 mostra como é a visão de um agente no BDI4JADE debugger. Para o agente só é mostrado o seu nome, *beliefs*, *plans*, *goals* e *capabilities*. No caso específico do que é mostrado na Figura 4.1, o escopo de variáveis é o de um `AchievePlanBody`. Sempre que a variável `this` pertence ao escopo de um agente, este é mostrado. Nenhuma outra informação é mostrada além das informações do agente.

Figura 4.1: Visão do BDI4JADE debugger para um agente

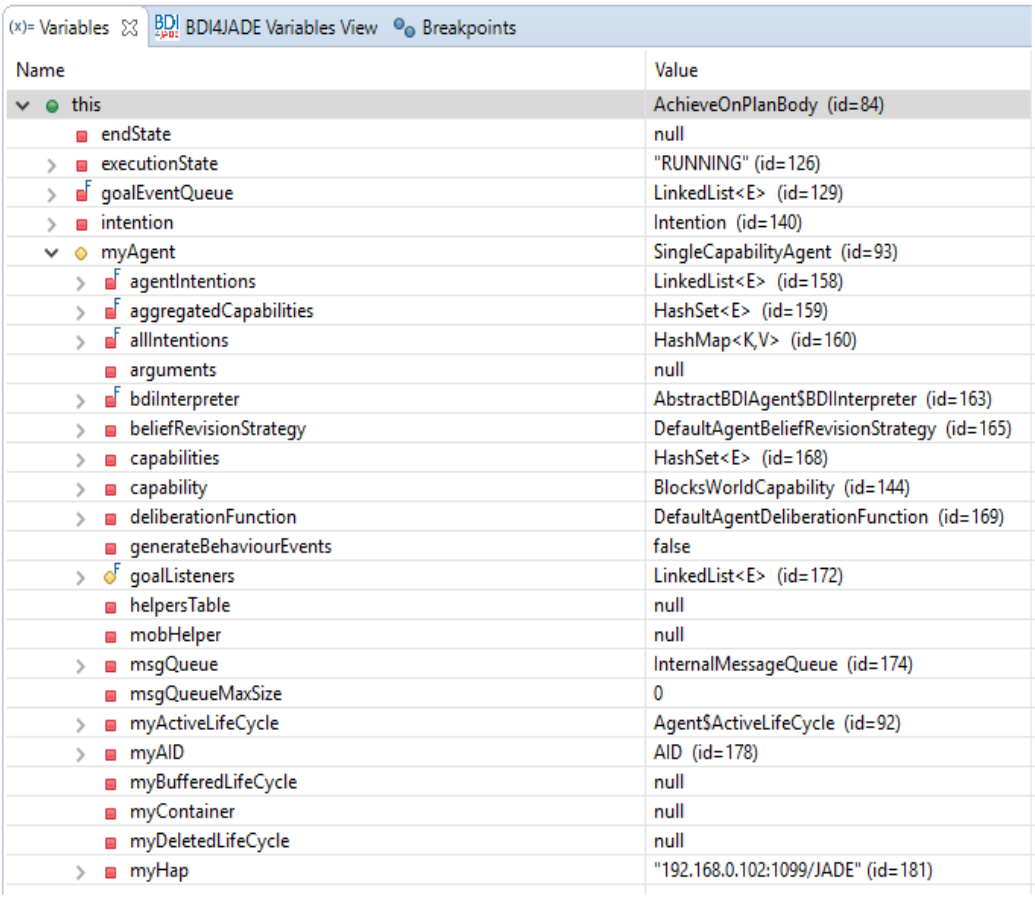


Fonte: Autor

Nota-se, comparando a Figura 4.1 com a Figura 4.2, que a apresentação de um agente na visão do BDI4JADE debugger é bem mais intuitiva. Na visão do Eclipse apare-

cem muitas informações irrelevantes como `helpersTable`, `mobHelper`, `arguments` e `myContainer`. O desenvolvedor focado na arquitetura fornecida pelo BDI4JADE (que inclui variáveis herdadas do JADE) normalmente não precisará visualizar essas informações e, caso necessite em algum momento, poderá visualizar diretamente na visão do Eclipse.

Figura 4.2: Visão do depurador do Eclipse para um agente

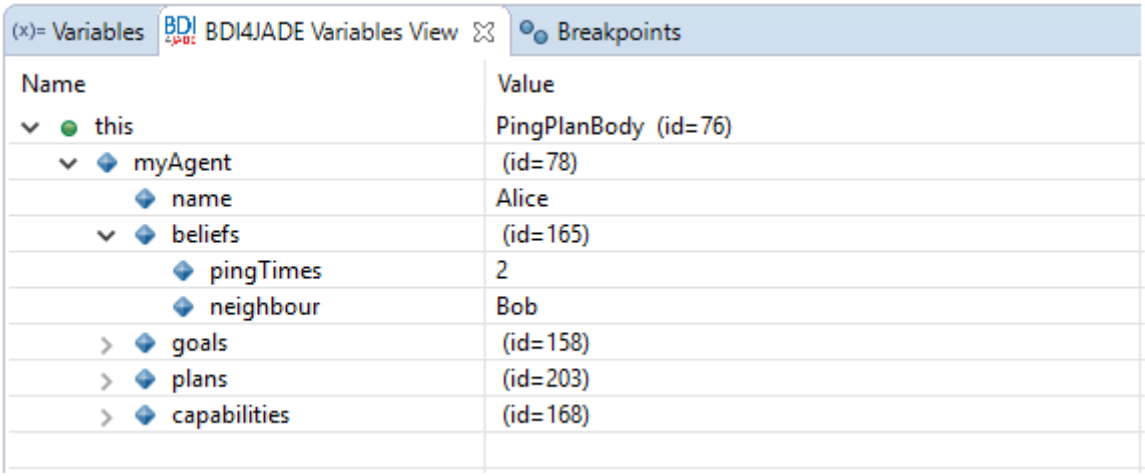


Fonte: Autor

4.2 Visualização das Beliefs

Abaixo seguem as visões apresentando as *beliefs* de um agente. Na visão do BDI4JADE, dentro do agente, são mostradas cada uma das suas *beliefs*. Para cada uma é mostrado o seu nome e valor. Essa mesma estrutura aparece dentro de uma *capability*. Não há diferença entre os dois conjuntos de *beliefs*, exceto que as *beliefs* que aparecem para o agente reunem todas as *beliefs* de todas as *capabilities* do agente, enquanto que as *beliefs* que aparecem para a *capability* são somente as da própria *capability*.

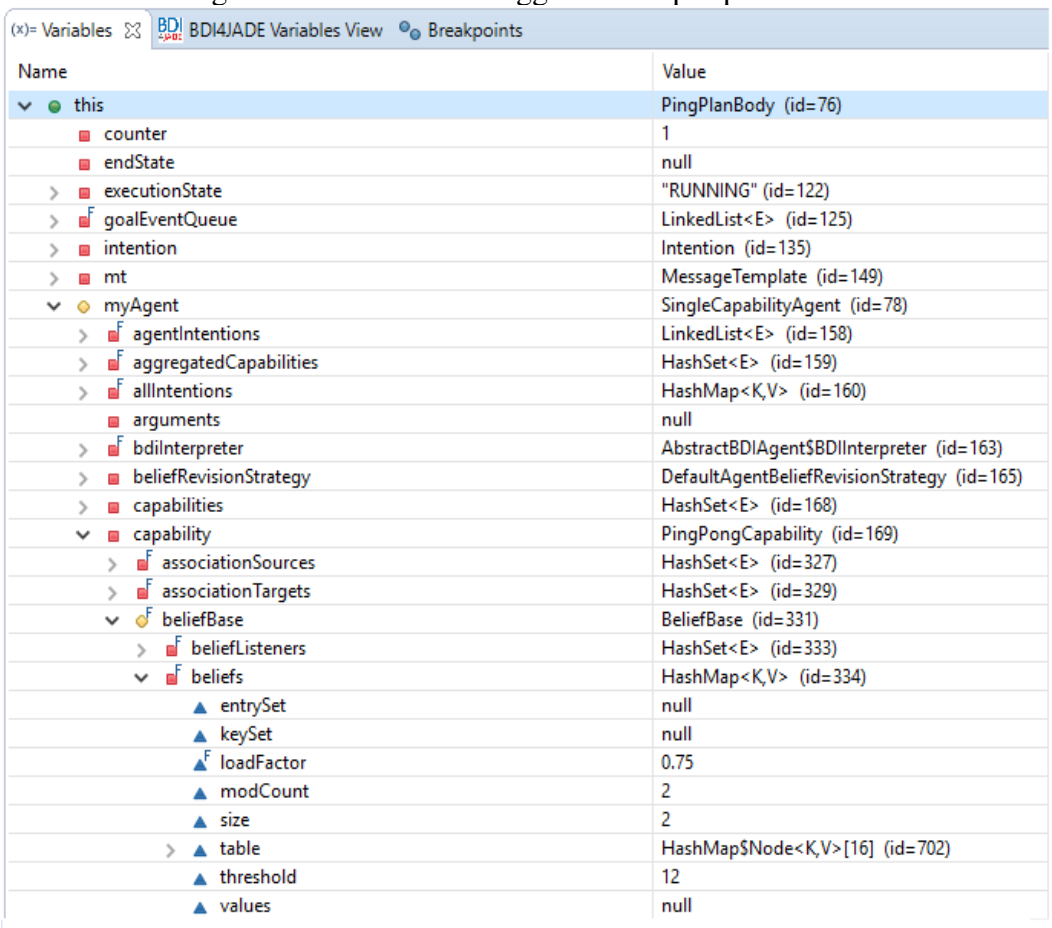
Figura 4.3: Visão do BDI4JADE debugger para beliefs



Fonte: Autor

Conforme aparece na Figura 4.4, a visão do depurador do Eclipse mostra variáveis da estrutura interna de listas, como `keyset`, `size` e `table`, que não interessam ao desenvolvedor.

Figura 4.4: Visão do debugger do Eclipse para beliefs

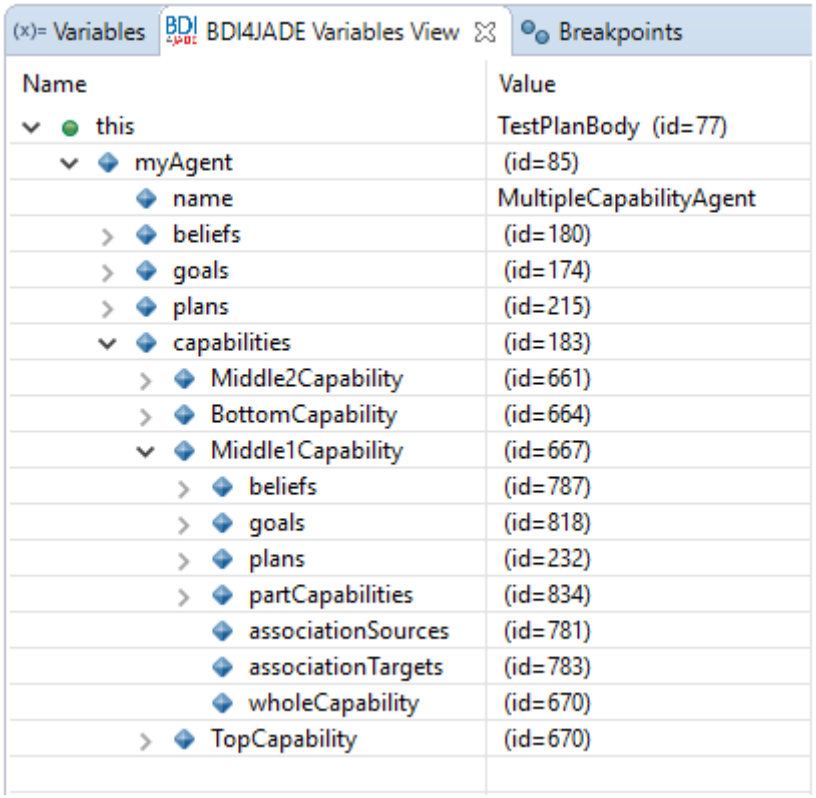


Fonte: Autor

4.3 Visualização da Capability

A seguir, são mostradas as visões para uma *capability*. A estrutura de uma *capability* na visão do BDI4JADE é semelhante à estrutura de um agente. *Beliefs*, *goals* e *plans* são iguais estruturalmente. As outras variáveis da *capability* são conjuntos de outras *capabilities*. Quando uma *capability* não possui outras *capabilities* relacionadas a ela, essas variáveis são mostradas vazias, ou seja, sem a possibilidade de expandi-las.

Figura 4.5: Visão do BDI4JADE debugger para capability

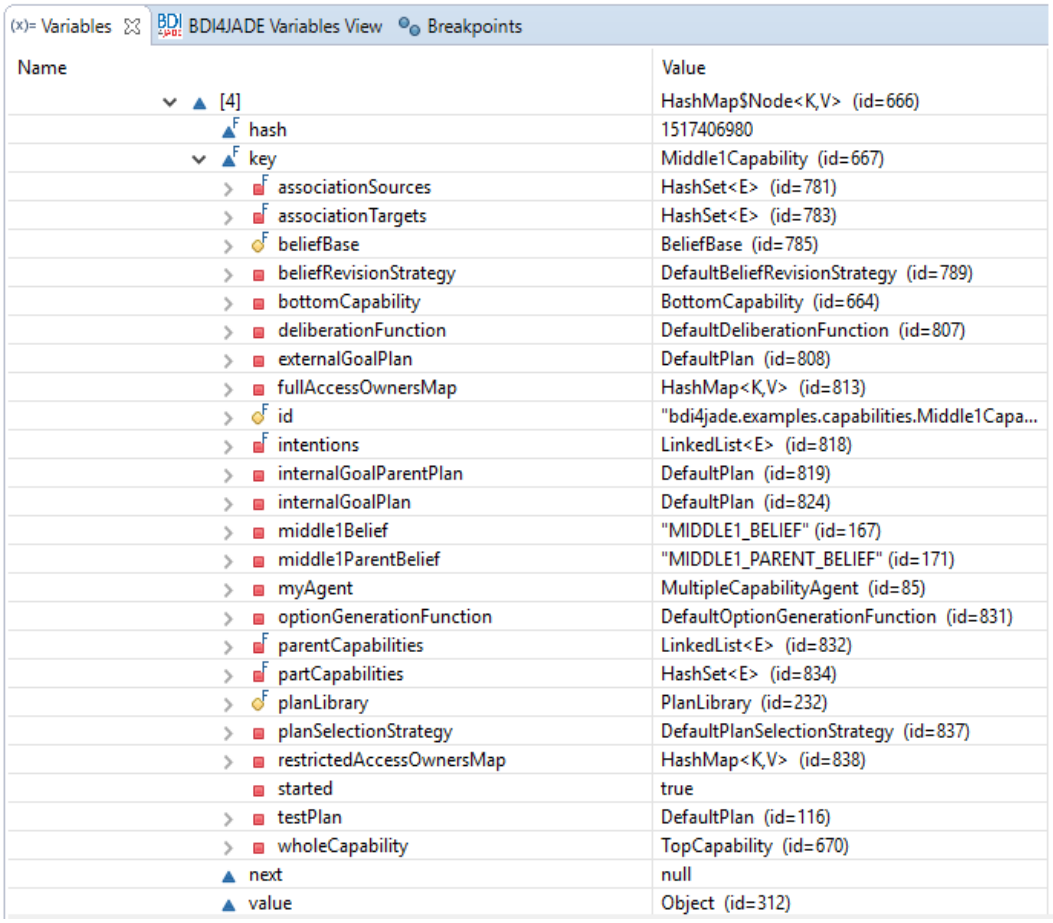


Fonte: Autor

Conforme pode ser visto na Figura 4.6, aqui também a visão do BDI4JADE reduziu significativamente a quantidade de informações apresentadas e reduziu bastante o esforço de leitura necessário ao desenvolvedor. Se, por exemplo, o desenvolvedor precisar descobrir, na visão do depurador do Eclipse mostrada na Figura abaixo, quais são as *beliefs* da *capability*, ele terá que pesquisar dentro de `beliefBase` e terá que ler todas as variáveis no mesmo nível de hierarquia de `beliefBase`, já que neste nível aparecem também *beliefs* como `middle1belief` e `middle1ParentBelief`.

Outra problema é que para chegar até uma determinada *capability*, como, por exemplo, a `Middle1Capability` da Figura 4.6, é necessário pesquisar dentro de `key` de cada índice de uma variável `table` já que o nome da *capability* não aparece explicitamente em um *label*.

Figura 4.6: Visão do debugger do Eclipse para capability



Fonte: Autor

Dessa forma, percebe-se que as visualizações fornecidas simplificam o processo de depuração de um agente BDI. Assim, o desenvolvedor pode focar naquilo que está sendo desenvolvido. Porém, caso ele queira verificar outras variáveis do agente, associadas as plataformas BDI4JADE ou JADE, as visualizações providas pelo Eclipse continuam disponíveis.

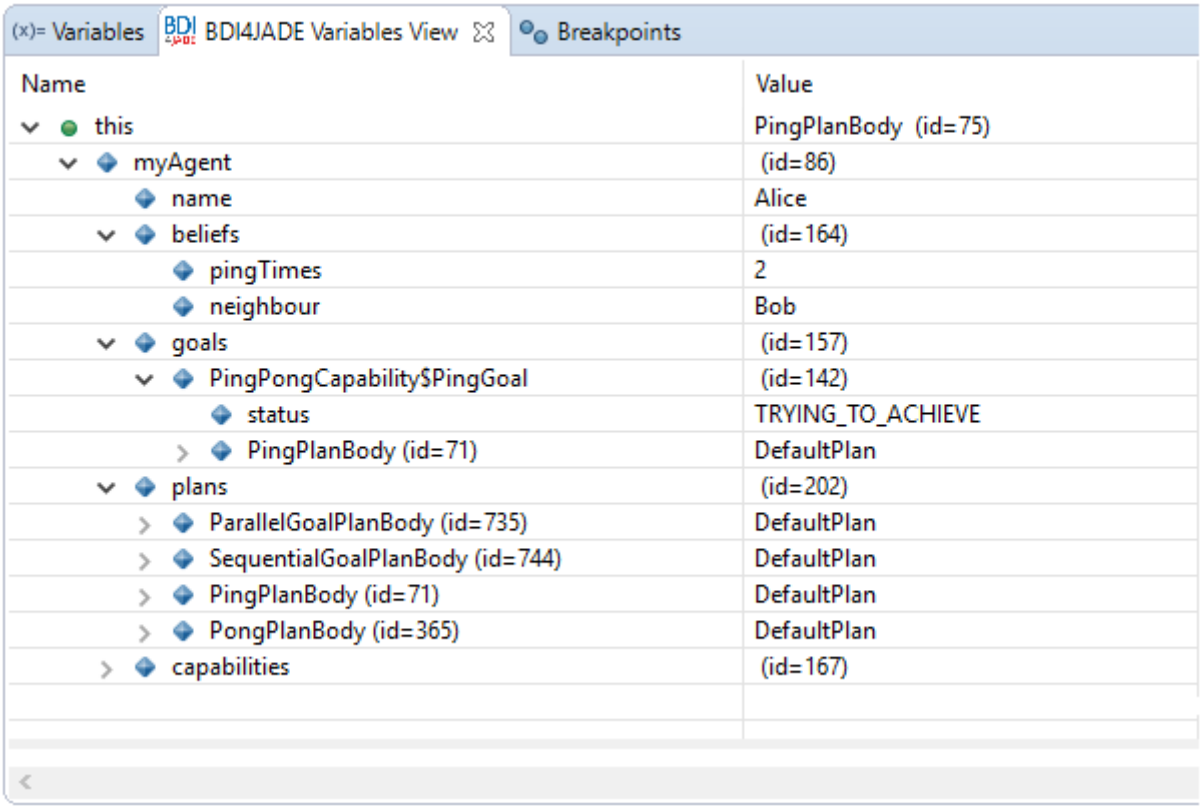
4.4 Exemplo de Uso

Segue um exemplo ilustrativo de uso do BDI4JADE Debugger em uma aplicação. A aplicação utilizada é um dos exemplos disponíveis no BDI4JADE e consiste em dois

agentes, Alice e Bob, que jogam uma partida de Ping-Pong. Nas Figuras abaixo, pode-se ver o estado interno de cada um dos agentes ao longo do jogo.

O agente Alice possui duas *beliefs*: *pingTimes* que informa quantas vezes o agente enviará uma jogada e *neighbour* que identifica o agente oponente. Seu objetivo, representado por *PingGoal*, é jogar tantas vezes quanto definido em *pingTimes*. O agente está tentando atingir o objetivo (*status trying to achieve*) através de *PingPlanBody*.

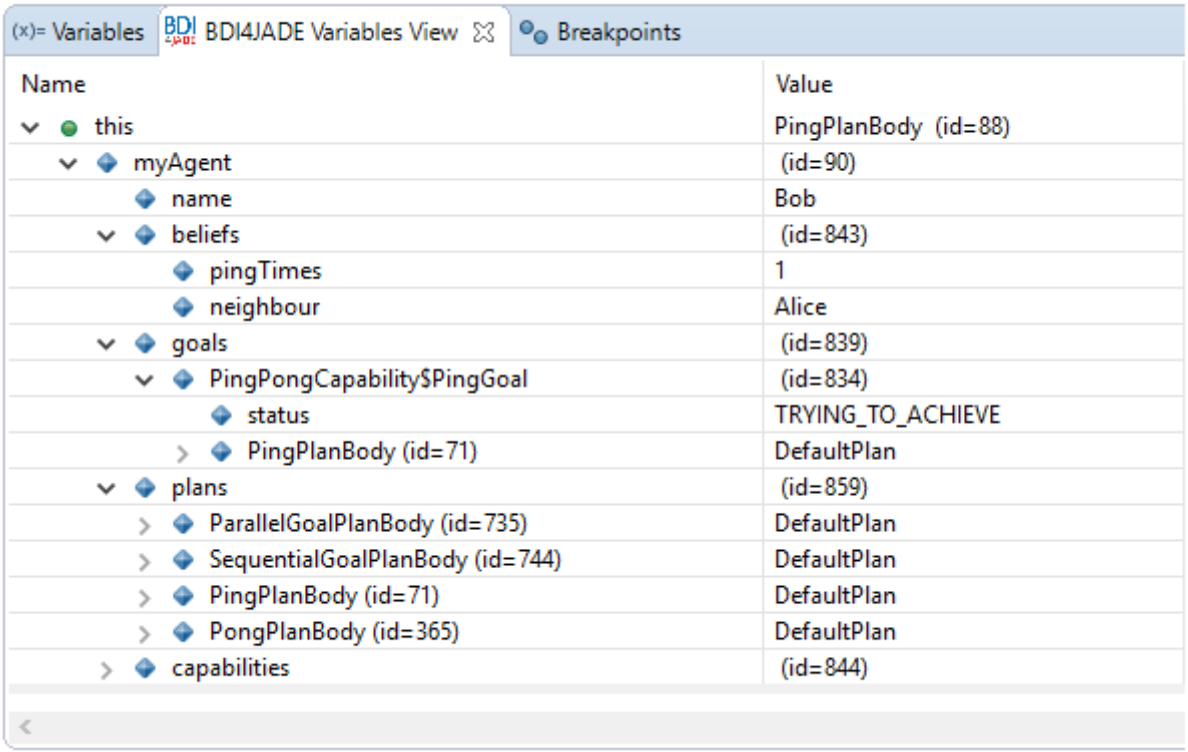
Figura 4.7: Estado interno de agente Alice



Fonte: Autor

O agente Bob possui a mesma estrutura interna que Alice e diferentes valores para as suas *beliefs*. Vale notar que ele, e também Alice, possui vários *plans* à sua disposição e que são mostrados na variável *plans* da visão do depurador. No entanto, ele utiliza apenas *PingPlanBody*.

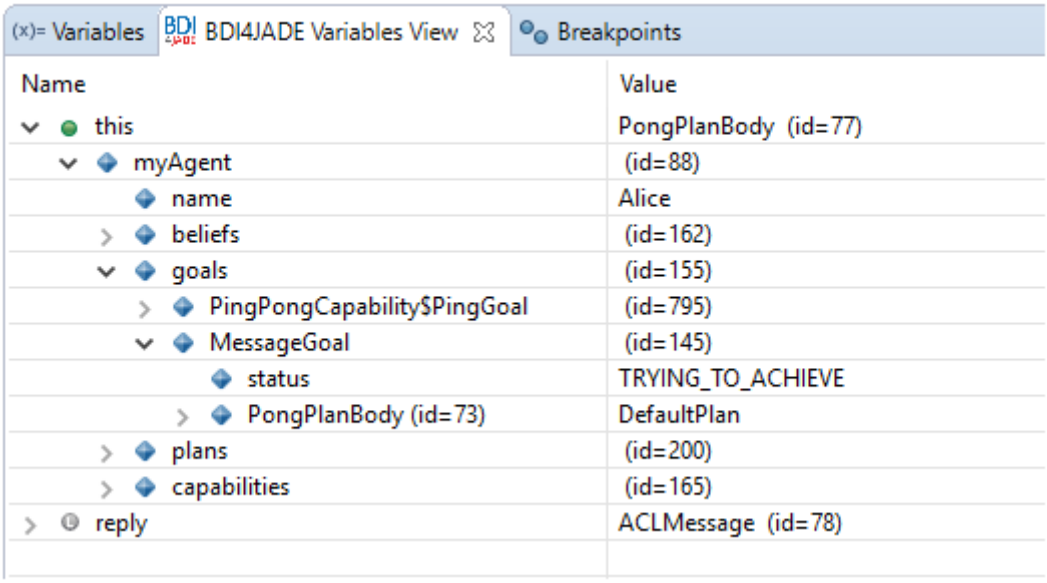
Figura 4.8: Estado interno de agente Bob



Fonte: Autor

Quando um agente envia uma jogada, é criado um *goal* específico de mensagens *MessageGoal*. Ele contém uma mensagem "Ping"ou "Pong"e é executado por um *pingPlanBody* ou *pongPlanBody*. O *goal* é alcançado após o envio da mensagem.

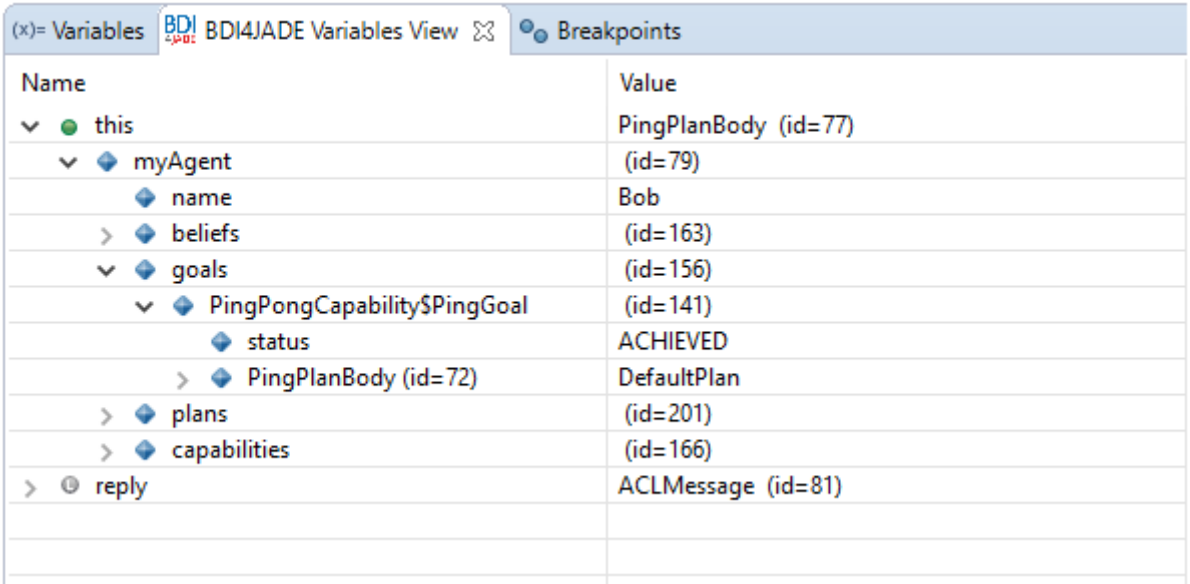
Figura 4.9: Estado interno de agente Alice ao enviar mensagem



Fonte: Autor

Ao terminar o jogo, o *goal* tem seu *status* alterado para *achieved*. Nesse momento, o agente Bob não possui mais um *goal* do tipo *MessageGoal*, pois este não é mais uma *intention* do agente.

Figura 4.10: Estado interno de agente Bob ao fim do jogo



Fonte: Autor

5 TRABALHOS RELACIONADOS

Como já foi dito acima, há outras plataformas para desenvolvimento de SMAs de agentes BDI. Neste capítulo são apresentadas os depuradores das plataformas mais usuais que implementam arquitetura BDI. É mostrada uma definição sucinta de cada uma, quais são as IDEs, *plugins*, depuradores e ferramentas disponíveis para o desenvolvimento. Com relação aos depuradores disponíveis para cada uma, é feita uma análise sob o aspecto dos componentes da arquitetura BDI. Por fim, é feito um estudo comparativo entre esses depuradores e o BDI4JADE debugger.

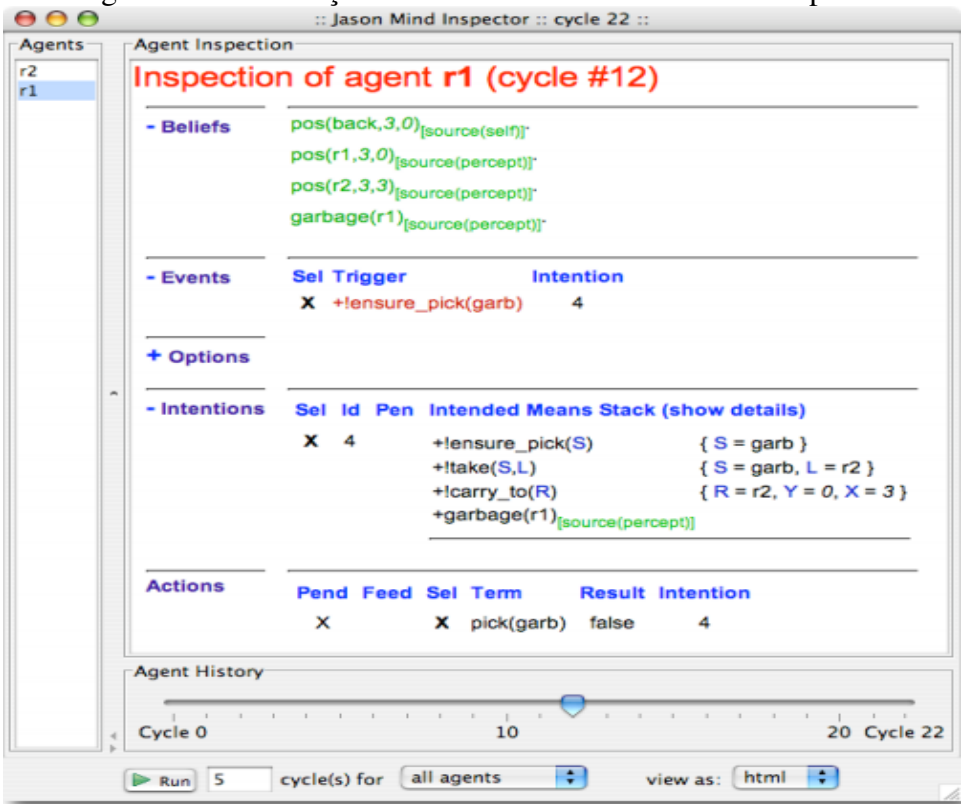
5.1 Jason Debugger

Jason é um interpretador baseado em JAVA para uma versão extendida da linguagem AgentSpeak (BORDINI; HüBNER, 2007). A linguagem de programação AgentSpeak(L) foi proposta por Anand S. Rao (ANAND, 1996) e consiste em uma extensão de programação em lógica para arquitetura de agentes BDI (BORDINI; HüBNER, 2007). Essa linguagem é baseada em lógica de primeira ordem, com eventos e ações.

Através de uma combinação de classes JAVA e arquivos com linguagem declarativa, Jason tem uma maneira simples para definir um sistema multiagente. Quem quer programar para Jason pode utilizar uma IDE que é um *plugin* do jEdit. O jEdit é um editor de texto para programadores que permite a adição de *plugins* customizados (JEDIT, 2015). Além disso, Jason possui um *plugin* para Eclipse.

Para depuração de programas, é possível utilizar o depurador tanto através do jEdit quanto através do Eclipse. Além disso, através do jEdit, é possível também utilizar o Sniffer do JADE para visualizar as mensagens trocadas entre os agentes. Quanto à visualização de elementos da arquitetura BDI, o jEdit possui uma ferramenta chamada Mind Inspector. Nela, é possível visualizar o estado dos agentes em tempo de execução. A ferramenta mostra, para cada agente, informações a respeito das *beliefs*, *intentions*, *options*, *actions*, *events* e *annotations*. É possível executar ciclos passo-a-passo acompanhando simultaneamente o SMA. Pode-se executar o ciclo de um agente individualmente ou de todos simultaneamente. Porém, o Mind Inspector não permite a inserção de *breakpoints* no código. Há uma poluição visual devido à linguagem do paradigma lógico que prejudica a clareza das informações. Não é possível alterar os valores de um agente ou *belief* em tempo de execução.

Figura 5.1: Informações visualizadas através do Mind Inspector



Fonte: (TUTORIAL, 2016)

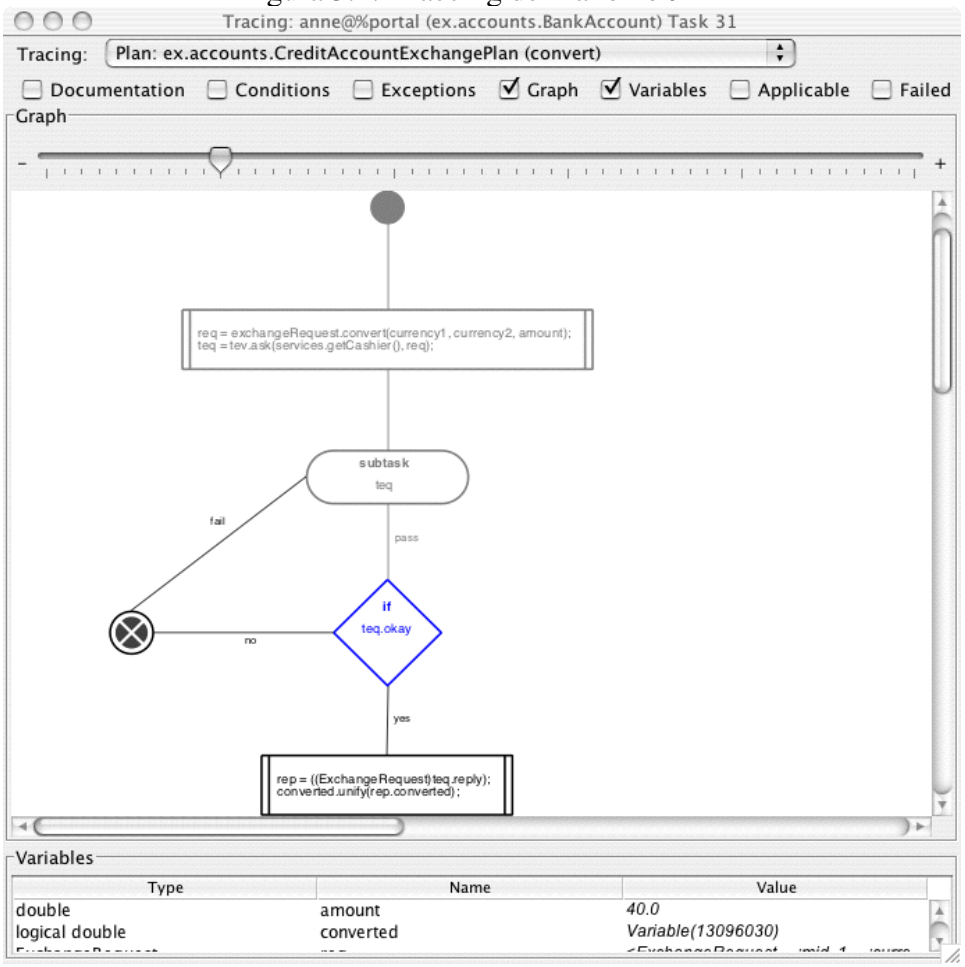
5.2 JACK Debugger

JACK Intelligent Agents (JACK) é uma extensão da linguagem JAVA orientada a agentes (JACK, 2005a). Ela estende a sintaxe do JAVA e fornece conceitos relacionados a agentes como componentes primitivos da linguagem, tais como: agentes, *capabilities*, eventos, mensagens, planos, bases de conhecimento e gerenciamento de concorrência (SARDINA, 2007).

JACK possui sua própria IDE, o JDE (JACK Development Environment). O JDE posui ferramentas gráficas para o desenvolvimento de planos e modelagem de sistemas de vários agentes. Há também uma opção para fazer *tracing* de agentes. Nela, uma janela gráfica apresenta um grafo com a representação de um componente sendo executado, conforme a Figura 5.2. Os componentes possíveis são planos, eventos, tarefas, *capabilities*, agentes, entre outros. Os valores de parâmetros e variáveis podem ser vistos na parte inferior da janela. Pode-se ver ainda a documentação e exceções. O JDE também fornece mais duas ferramentas para depuração: Audit Logging e Generic Debugging. Audit Log-

ging serve para visualizar as mensagens trocadas entre os agentes. Generic Debugging permite visualizar *prints* do agente a partir de comandos enviados em tempo de execução. Esta ferramenta é destinada principalmente para depuração remota. As opções de depuração apresentam muitas informações dos agentes. Porém, é interessante poder visualizar graficamente a execução dos agentes.

Figura 5.2: Traccing de Plano no JDE



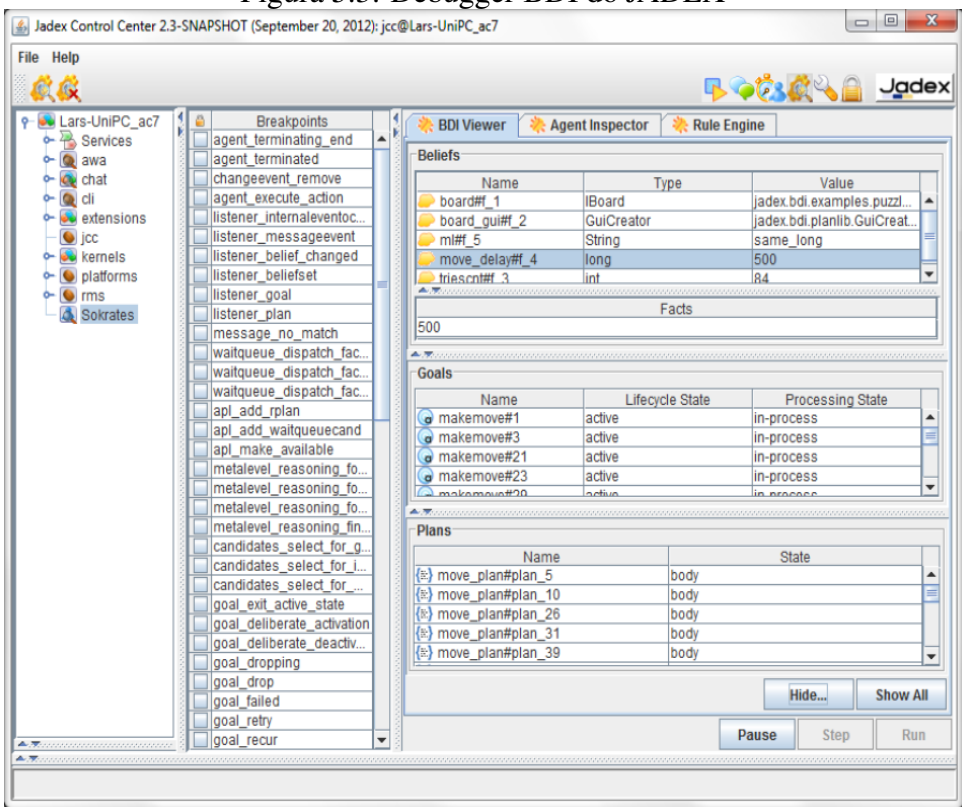
Fonte: (JACK, 2005b)

5.3 JADEX Debugger

JADEX é uma extensão da plataforma JADE que implementa a arquitetura BDI de agentes. A última versão da plataforma é inteiramente em Java. Ela oferece conceito de *capabilities*, dentre outros. Como um agente JADEX é também um agente JADE, as ferramentas de execução fornecidas pelo JADE (Sniffer e Dummy) também podem ser usadas com agentes JADEX.

JADEX possui uma aplicação chamada JCC (JADEX Control Center) a qual gerencia um SMA feito em JADEX. Ele possui um depurador próprio. Para um agente BDI, o depurador mostra suas *beliefs*, *plans* e *goals*. Ele possui uma aba para visualização do estado interno de um agente com maior detalhamento, podendo ver toda a estrutura do agente. Também permite a definição de *breakpoints*. Além disso, ele possui uma visão que contém uma representação visual das regras do agente.

Figura 5.3: Debugger BDI do JADEX



Fonte: (JADEX, 2016)

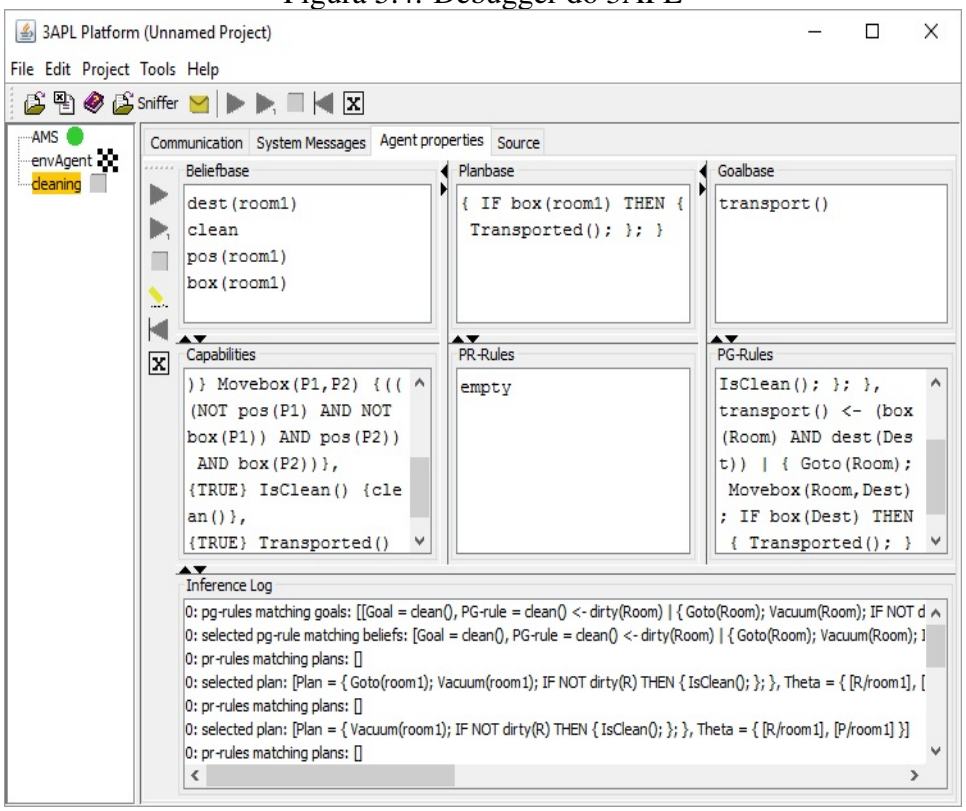
5.4 3APL Debugger

3APL consiste em uma linguagem de programação para agentes racionais baseada na arquitetura BDI. Ela mistura linguagem lógica e procedural. Os principais componentes da linguagem são *beliefs*, *plans*, *goals*, *capabilities* e *practical reasoning rules*.

3APL vem também com uma plataforma própria. A plataforma possui um pequeno *canvas* gráfico para visualização do ambiente de agentes, um *sniffer* onde é possível examinar as mensagens trocadas entre os agentes e uma visão com as propriedades de cada agente. Conforme a Figura 5.4, nessa visão são apresentadas as crenças, planos,

objetivos e *capabilities* do agente, além de regras, e o usuário pode executar os agentes passo-a-passo, individualmente ou todos simultaneamente. É possível também modificar os valores em tempo de execução.

Figura 5.4: Debugger do 3APL



Fonte: O Autor

5.5 Comparativo

Nesta seção, é feito uma comparação entre os diversos sistemas descritos na seção anterior e o BDI4JADE. Os critérios de comparação usados foram os seguintes:

- **Clareza das informações, legibilidade:** é fácil de entender?
- **Facilidade de acesso:** consigo encontrar na interface a informação que procuro com facilidade?
- **Rastreabilidade no código:** consigo encontrar no código, através de *breakpoints*, o que levou um agente a ter um determinado estado?
- **Relevância das informações:** que tipos de informacao são apresentadas? São focadas na arquitetura bdi? Há poluição de muitas informações?
- **Isolamento:** consigo isolar as informações só de um determinado agente ou *capa-*

bility?

- **Dinâmico:** posso alterar valores em tempo de execução?
- **Visualização de capabilities:** existe a possibilidade de visualizar as *capabilities*?
- **Visualização gráfica:** existe alguma visualização gráfica?

Primeiramente, é apresentada uma tabela analisando cada depurador individualmente. Depois, é mostrada uma tabela comparando os depuradores. Alguns poucos critérios não foram verificados devido a falta de uma licença do software e por não aparecer na documentação.

Critérios quanto à depuração de agentes BDI para Jason (Mind Inspector)	
Clareza das Informações	Ruim (lógica de predicados)
Facilidade de acesso	Boa
Rastreabilidade	Ruim (Não tem breakpoints)
Relevância das Informações	Boa
Isolamento	Possui
Dinâmico/Estático	Estático
Visualização de Capabilities	Não Possui
Visualização de Gráfica	Não possui

Critérios quanto à depuração de agentes BDI para JACK	
Clareza das Informações	Ruim (por ser um fluxograma)
Facilidade de acesso	Ruim (por ser um fluxograma)
Rastreabilidade	Ruim (por ser um fluxograma)
Relevância das Informações	Ruim (muitas informações)
Isolamento	Possui
Dinâmico/Estático	—
Visualização de Capabilities	Possui
Visualização de Gráfica	Possui

Critérios quanto à depuração de agentes BDI para JADEX	
Clareza das Informações	Boa
Facilidade de acesso	—
Rastreabilidade	Boa (possui breakpoints)
Relevância das Informações	Boa (possui view focada em um agente)
Isolamento	Possui
Dinâmico/Estático	—
Visualização de Capabilities	—
Visualização de Gráfica	Possui

Critérios quanto à depuração de agentes BDI para 3APL	
Clareza das Informações	Ruim (lógica de predicados)
Facilidade de acesso	Boa
Rastreabilidade	Boa
Relevância das Informações	Boa
Isolamento	Possui
Dinâmico/Estático	Dinâmico
Visualização de Capabilities	Possui
Visualização de Gráfica	Não possui

Critérios quanto à depuração de agentes BDI para BDI4JADE	
Clareza das Informações	Boa
Facilidade de acesso	Boa
Rastreabilidade	Boa
Relevância das Informações	Boa
Isolamento	Possui
Dinâmico/Estático	Dinâmico
Visualização de Capabilities	Possui
Visualização de Gráfica	Não possui

Comparativo					
	BDI4JADE	Jason	JACK	JADEX	3APL
Clareza das informações	Boa	Ruim	Ruim	Boa	Ruim
Facilidade de acesso	Boa	Boa	Ruim	—	Boa
Rastreabilidade no código	Boa	Ruim	Ruim	Boa	Boa
Relevância das informações	Boa	Boa	Ruim	Boa	Boa
Isolamento	Possui	Possui	Possui	Possui	Possui
Dinâmico	Dinâmico	Estático	—	—	Dinâmico
Visualização de capabilities	Possui	Não Possui	Possui	—	Possui
Visualização gráfica	Não Possui	Não Possui	Possui	Possui	Não Possui

Verifica-se que nos dois principais critérios a que este trabalho se propõe contribuir, clareza e relevância das informações, BDI4JADE é igual ou melhor que todos os outros depuradores. Além disso, com exceção de JADEX, BDI4JADE se apresenta mais completo neste conjunto de critérios.

6 CONCLUSÃO

Sistemas multiagentes estão provocando muito interesse tanto academicamente quanto industrialmente. Já existem diversas bibliotecas e *frameworks* para o desenvolvimento de SMAs. Muitos deles implementam a arquitetura BDI, a qual é bem consolidada. Com a preocupação de uma maior facilidade para o desenvolvimento em larga escala, foi criada uma ferramenta escrita totalmente em Java, chamada BDI4JADE. Esta ferramenta possuia uma carência quanto ao processo de depuração, pois não possuia uma visão de variáveis focada nos aspectos da arquitetura BDI.

Este trabalho apresentou um projeto de visão de variáveis onde foram eliminadas informações irrelevantes e apresentadas apenas as informações de um agente quanto à sua organização BDI. Foi proposta uma hierarquia de variáveis onde aparecem as *beliefs*, *goals*, *plans* e *capabilities* do agente. Foi implementada como um *plugin* para a IDE Eclipse. Verificou-se claramente que houve uma melhoria na facilidade de entendimento do estado interno de um agente. Comparando a visão de variáveis padrão do Eclipse com a visão de variáveis do BDI4JADE, nota-se que há, em recorrentes situações (sempre que o depurador está dentro do contexto de um agente), uma diminuição drástica no número de variáveis apresentadas na interface. Isso facilita o entendimento rápido do que está acontecendo, pois elimina toda a poluição da tela.

Realizou-se também um estudo comparativo entre depuradores de plataformas semelhantes que oferecem a arquitetura BDI. Foram estudas quatro das mais populares. Verificou-se que algumas possuem informações pouco claras e irrelevantes. Outras possuem dificuldades de associar a informação vista com o local no código onde aquela informação é definida, devido à falta de *breakpoints* e de integração do software de depuração com a IDE de desenvolvimento. Com exceção de JADEX, todas elas pecam em algum destes aspectos. Assim, temos que BDI4JADE aparece como uma ferramenta mais completa nesses aspectos.

Como trabalhos futuros, são feitas as seguintes sugestões:

- Facilitar a compatibilidade do *plugin* com futuras versões do BDI4JADE.
- Otimizar o desempenho.
- Verificar a possibilidade de, através das opções de preferências da interface do Eclipse, mudar a hierarquia de variáveis apresentada na visão do depurador configurando *detail formatters* (ECLIPSE, a) e *logical structure providers* (ECLIPSE, b).

REFERÊNCIAS

3APL. *3APL Official Site*. 2005. Disponível em <http://www.cs.uu.nl/3apl-m/>. Acesso em: Junho 2016.

ANAND, S. R. Agentspeak(l): Bdi agents speak out in a logical computable language. In: **Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World(MAAMAW'96)**. Eindhoven, The Netherlands: Springer-Verlag, 1996. p. 42–55.

BELLIFEMINE, F. L.; CLAIRE, G.; GREENWOOD, D. **Developing Multi-Agent Systems with JADE**. [S.l.]: JohnWiley Sons, 2007.

BORDINI, R. H.; HüBNER, J. F. **Jason, a Java-based interpreter for an extended version of AgentSpeak**. 2007. Disponível em <http://jason.sourceforge.net/Jason.pdf>. Acesso em: Maio 2016.

BORDINI, R. H.; HüBNER, J. F.; WOOLDRIDGE, M. J. **Programming Multi-Agent Systems in AgentSpeak Using Jason**. [S.l.]: JohnWiley Sons, 2007.

BRATMAN, M. E. **Intention, Plans, and Practical Reason**. [S.l.]: Harvard University Press, 1987.

D'INVERNO, M. et al. M. j. - a formal specification of dmars. In: **Agent Theories, Architectures, and Languages**. [S.l.: s.n.], 1997. p. 155–176.

ECLIPSE. **Eclipse debug details formatter**. Disponível em <http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fdebug%2Fref-detail_formatters.htm>. Acesso em: Novembro 2016.

ECLIPSE. **Eclipse debug logical structures**. Disponível em <http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fdebug%2Fref-logical_structures.htm>. Acesso em: Novembro 2016.

ECLIPSE. **Eclipse Views Extension Point**. Disponível em <http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ui_views.html>. Acesso em: Maio 2016.

GEORGEFF et al. The belief-desire-intention model of agency. In: **Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98)**. [S.l.]: Springer-Verlag, 1999. p. 1–10.

IEEE. **Foundation for Intelligent Physical Agents**. 2005. Disponível em <http://www.fipa.org/>. Acesso em: Maio 2016.

JACK. **JACK Inteliggent Agents - Agent Manual**. 2005. Disponível em <http://www.aosgrp.com/documentation/jack/Agent_Manual.pdf>. Acesso em: Maio 2016.

JACK. **JACK Inteliggent Agents - Tracing and Logging Manual**. 2005. Disponível em <http://www.aosgrp.com/documentation/jack/JACK_Tracing_Manual_WEB/index.html#TracingfromtheJDE>. Acesso em: Junho 2016.

JACK. **JACK Official Site**. 2015. Disponível em <http://aosgrp.com/products/jack/>. Acesso em: Junho 2016.

JADEX. **Jadex Official Site**. 2016. Disponível em <https://www.activecomponents.org/#/project/news>. Acesso em: Junho 2016.

JEDIT. **jEdit Official Website**. 2015. Disponível em <http://www.jedit.org/>. Acesso em: Maio 2016.

NUNES, I. **Improving the Design and Modularity of BDI Agents with Capability Relationships**. 2014. Disponível em: <http://inf.ufrgs.br/~ingridnunes/publications/emas-2014.pdf>. Acesso em: Junho 2016.

NUNES, I.; LUCENA, C. J.; LUCK, M. **BDI4JADE: a BDI layer on top of JADE**. 2011. Disponível em: <http://www.dcs.kcl.ac.uk/staff/mml/publications/assets/promas-2011.pdf>. Acesso em: Maio 2016.

RAO, A. S.; GEORGEFF, M. P. Bdi-agents: from theory to practice. In: **Proceedings of the First Intl. Conference on Multiagent Systems**. [S.l.: s.n.], 1995.

RUSSELL, S.; NORVIG, P. **Inteligência Artificial**. [S.l.]: Elsevier, 2004.

SARDINA, S. **BDI Programming**. 2007. Disponível em <http://goanna.cs.rmit.edu.au/~ssardina/courses/Roma07PhDcourse/lect09/handout_jackbdi.pdf>. Acesso em: Maio 2016.

STOLFO, S. et al. Jam: Java agents for meta-learning over distributed databases. In: THE THIRD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY DATA MINING., 1997. **Proceedings...** [S.l.]: AAAI Press, 1997.

TUTORIAL. **Getting started with Jason**. 2016. Disponível em <http://jason.sourceforge.net/mini-tutorial/getting-started/>. Acesso em: Maio 2016.

WOOLDRIDGE, M. J. **Intelligent Agents**. [S.l.: s.n.], 1999. 27–77 p.

WOOLDRIDGE, M. J. **Reasoning about Rational Agents**. [S.l.]: MIT Press, 2000.

WOOLDRIDGE, M. J. **An Introduction to MultiAgent Systems**. [S.l.]: JohnWiley Sons, 2009.

ZAMBIASI, S. P. **Agentes Inteligentes**. 2011. Disponível em <http://gsigma.ufsc.br/~popov/wiki/index.php/Agentes_Inteligentes>. Acesso em: Novembro 2016.