UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MICHAEL CHARLES ANDREW CURADO FLEURY DE VIDIGAL

AGENTES POLIMÓRFICOS DINÂMICOS INTELIGENTES: TEORIA E APLICAÇÃO

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Área de Concentração: Automação e Sistemas Elétricos Industriais.

Orientador: Prof. Leonardo de Mello Honório

Co-orientador: Prof. Luiz Edival de Sousa

Julho de 2007

ITAJUBÁ - MG

À minha avó Maria Curado Fleury (in memoriam).

À minha família, em especial à minha mãe Jeanne, que com toda a luta, esforço e amor dedicado é a maior responsável por minha formação pessoal e profissional e que, mesmo a uma grande distância, é pra mim uma grande fonte de energia. Agradeço ao meu pai pelo apoio em minha formação e ao meu irmão Arthur pela sua amizade.

À minha noiva Sâmia, pela compreensão e incentivo, sendo minha maior inspiração e motivação, e os seus pais Habib e Marisa por me acolherem como um filho.

Aos amigos, principalmente aos irmãos da República Toca-Gado, sendo minha segunda família desde 1999 e aos grandes amigos da república Quivara.

Aos meus professores, em especial ao meu Orientador Leonardo de Mello Honório pelo convite, oportunidade, orientação, por me ensinar a ser pesquisador e ter acreditado no meu trabalho. Ao professor Luiz Edival, por toda ajuda e co-orientação.

Aos membros do CRTI e a todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho.

"Quem não tem gratidão, não tem caráter" (Jorge Kajuru)[1].

[1] Citação verbal frequentemente utilizada pelo Jornalista e Cronista Esportivo Jorge Reis da Costa, conhecido como Jorge Kajuru.

“Não é do trabalho que nasce a civilização: ela nasce do tempo livre e do jogo.”

Alexandre Koyré.

“Estamos desabituados de uma tal maneira a fazer as coisas com calma, que assim que dispomos de uma hora livre a enchemos de tantos compromissos ou tarefas, que o tempo acaba sempre faltando. Tempo e espaço, ou seja, as duas categorias mais importantes da nossa vida, reduziram-se de tal forma, que dispor deles, isto é ter tempo e espaço passou a ser um luxo. [...] Mas em que consistirá o luxo na sociedade pós-industrial? Se vive de forma luxuosa, quem possui bens que são escassos pode-se perguntar: o que será escasso no futuro próximo? Segundo Enzensberger, seis coisas serão escassas: o tempo, a autonomia, o espaço, a tranqüilidade, o silêncio e o ambiente ecologicamente saudável. A esses bens cada vez mais "luxuosos" porque cada vez mais raros, eu somaria também a convivialidade e a beleza. Como pode ver, trata-se de bens cuja disponibilidade depende mais da sensibilidade, da formação e da cultura do que do dinheiro.”

De Masi, Domenico, O Ócio Criativo. Rio de Janeiro: Sextante, 2000.

RESUMO

Quando múltiplos agentes com características diferentes são considerados para a solução de uma série de problemas, alguns fatores como a especificação do número de agentes a ser utilizado, qual função específica que deverá ser utilizada por cada um e ainda o planejamento das ações para a resolução do problema, não é de forma alguma trivial. Diferentes situações podem exigir conjuntos específicos de agentes com conhecimento dedicado à resolução do problema. Para lidar com este cenário, este trabalho de dissertação sugere uma inovação na teoria de agentes inteligentes, um novo conceito chamado Agente Polimórfico Dinâmico Inteligente (APDI). Este agente juntamente com a arquitetura do sistema proposto neste trabalho tem por objetivo buscar um conjunto mínimo de agentes e recursos, (cada qual com suas próprias características) para se resolver um determinado problema utilizando técnicas de Planejamento Inteligente. Através de atualização on-line, a arquitetura de APDI permite a procura pelos agentes existentes no sistema que estão disponíveis, e também obter informações dos seus respectivos domínios. Com este conhecimento, o sistema utiliza uma heurística pré-definida para auxiliar dinamicamente o planejador, obtendo assim, um plano com um número reduzido de recursos. Após a determinação deste conjunto mínimo, os agentes se unem a outra entidade, o APDI, que receberá o plano gerado pelo planejador e será capaz de executar cada função específica de cada um dos agentes através de polimorfismo, realizando, assim a tarefa inicialmente atribuída. Outra funcionalidade importante desta metodologia é quando ocorre uma parada no sistema durante a execução do plano, e algum agente ou recurso do sistema se torna indisponível. Para resolver isto, é realizado um replanejamento para se buscar um novo conjunto de agentes que continue a execução da tarefa interrompida.

ABSTRACT

When multiple agents with different characteristics are considered to solve a set of problems, it is very challenging to specify the number of agents, the specific function of each one to be used, and the schedule of these actions in order to solve these problems. Different situations may demand different sets of agents with specific knowledge regarding how to solve each problem. To deal with scenarios like this, the present article suggests an innovation at the Intelligent Agent Theory, a new concept called Intelligent Dynamic Polymorphic Agent (IDPA). The IDPA has the goal of trying to find the minimum set of resources and agents (each one with its own characteristics) to plan and execute specific problems using AI planning. By on-line actualization, the IDPA architecture allows the quest for available agents present in the system and being feed by information about their respective domains. Using this knowledge, the system uses a predefined heuristic model to dynamically assist the planner obtaining a plan with a reduced number of resources. It also identifies and dispatches the necessary agents in order to carry out the plan. After this identification, the agents merge with the entity called APDI, which, through polymorphism, is capable of accomplish the initial task Another benefit of this methodology is that, due to its dynamic behavior, it is possible to change the original plan by the advent of an unexpected problem, even if the plan is being already executed.

SUMÁRIO

RESUMO ..... **V**

ABSTRACT ..... **VI**

SUMÁRIO ..... **VII**

ÍNDICE DE FIGURAS ..... **IX**

ÍNDICE DE TABELAS ..... **X**

1. INTRODUÇÃO **1**

2. AGENTES INTELIGENTES **5**

2.1. INTRODUÇÃO 5
2.2. DEFINIÇÃO 5
2.3. PROPRIEDADES DOS AGENTES INTELIGENTES 6
2.4. ESTRUTURA DOS AGENTES INTELIGENTES 7
2.5. SISTEMAS MULTIAGENTES 8
2.6. LINGUAGENS DE COMUNICAÇÃO ENTRE AGENTES 9
2.7. RESOLUÇÃO DISTRIBUÍDA DE PROBLEMAS 9

3. PLANEJAMENTO EM INTELIGÊNCIA ARTIFICIAL **13**

3.1. INTRODUÇÃO 13
3.2. PLANEJAMENTO DE DOMÍNIOS ESPECÍFICOS X PLANEJAMENTO DE DOMÍNIOS INDEPENDENTES 16
3.3. A LINGUAGEM PDDL 17
3.4. PLANEJADORES UTILIZADOS 24
3.5. FERRAMENTAS CASE PARA PDDL E TREEVIEW PDDL 34

4. METADADOS **41**

4.1. INTRODUÇÃO 41
4.2. ASSEMBLIES 42
4.3. CIL – COMMON INTERMEDIATE LANGUAGE 42
4.4. CLR - COMMON LANGUAGE RUNTIME 43

4.5. METADADOS 44

5. PROGRAMAÇÃO ORIENTADA A ASPECTO 47

5.1. INTRODUÇÃO 47
5.2. APLICAÇÃO DE AOP NO TRABALHO PROPOSTO 51
5.3. PORQUE UTILIZAR AOP? 54
5.4. UTILIZAÇÃO DE REFLECTION NO TRABALHO PROPOSTO 56

6. AGENTES POLIMÓRFICOS DINÂMICOS INTELIGENTES 60

6.1. INTRODUÇÃO 60
6.2. AGENTE MONITOR 60
6.3. HIPERDADOS 61
6.4. AGENTE PROBLEMA 62
6.5. AGENTE PLANEJADOR 63
6.6. ASPECTO FUNCIONAL 64
6.7. AGENTE POLIMÓRFICO DINÂMICO INTELIGENTE (APDI) 64
6.8. AGENTE REGULAR 66

7. APLICAÇÃO E RESULTADOS 68

7.1. AMBIENTE DE APLICAÇÃO 68
7.2. O SOFTWARE APDI 77
7.3. APLICAÇÕES POSSÍVEIS DA METODOLOGIA 79

8. CONCLUSÃO E TRABALHOS FUTUROS 81

8.1. CONCLUSÃO 81
8.2. TRABALHOS FUTUROS 82

REFERÊNCIAS BIBLIOGRÁFICAS 83

APÊNDICE A – GRAFOS DE PLANEJAMENTO 88

APÊNDICE B - DESCRIÇÃO DOS AGENTES REGULARES UTILIZADOS 93

APÊNDICE C – PROBLEMA 4 BLOCOS NO FORMATO PDDL 100

APÊNDICE D – TESTES REALIZADOS 102

ÍNDICE DE FIGURAS

Figura 1 – Agente interage com o ambiente através de sensores e atuadores 8
Figura 2 – Busca da Solução em árvore 16
Figura 3 – Árvore de busca do planejador 22
Figura 4 - Plano Gerado 23
Figura 5 – Busca Progressiva no espaço de estados 25
Figura 6 – Busca Regressiva no espaço de estados 26
Figura 7 – Alocação de Agentes 31
Figura 8 – Busca direcionada 33
Figura 9 – Vitaplan: Seleção de domínios, problemas e configuração do planejador 34
Figura 10 – Ação "Viajar" no editor Gráfico do Vitaplan 35
Figura 11 – Imagem da tela do software TreeView PDDL 37
Figura 12 – Tela de Configuração de Simulação do TreeView PDDL 39
Figura 13 – Compilação Convencional 43
Figura 14 – Compilação com o CLR 44
Figura 15 – Exemplo de Metadado 45
Figura 16 – Chamada a função utilizando um framework 46
Figura 17 – Entrelaçamento de Código e a Solução AOP 48
Figura 18 – Usando Modelagem Orientada a Aspecto 53
Figura 19 – Resolvedor Dedicado 55
Figura 20 – Replanejamento on-line 56
Figura 21 – Formação do APDI 65
Figura 22 – Ambiente de Aplicação 68
Figura 23 – Diagrama de Seqüência do plano gerado 73
Figura 24 – Tela do Software do APDI 78
Figura 25 – Grafo de planejamento para o problema "ter bolo e comer bolo também" até o nível S_2. 89

ÍNDICE DE TABELAS

Tabela 1 – Matriz de Propriedades Variantes _____ *11*
Tabela 2 – Desempenho dos planejadores _____ *30*
Tabela 3 – Testes para ordem crescente de entrada de blocos _____ *103*
Tabela 4 – Diferentes conjuntos de Agentes para a resolução de um mesmo problema _____ *105*
Tabela 5 – Testes para ordem aleatória de entrada de blocos _____ *106*
Tabela 6 – Testes para ordem decrescente de entrada de blocos _____ *107*

1. Introdução

De forma simplificada um agente é um componente pré-modelado que, dada uma entrada, executa um processamento e gera um resultado. A partir deste conceito genérico, a teoria de agentes é aplicada em um amplo espectro de problemas; otimização de sistemas [1], robótica [2], procedimentos de negócios [3] dentre outros. Esta variedade de aplicações de agentes se deve, em grande parte, pela capacidade de se definir de forma eficaz as informações de entrada e o resultado gerado pelo processamento do agente. Esta pré-definição de variáveis e funcionalidades gera, entre outros benefícios, grande portabilidade, confiabilidade e modularidade a este paradigma. Porém os agentes, mesmo os com capacidade de análise e geração de conhecimento, são estruturas de funcionalidade fixa. Logo, mesmo que os agentes sejam capazes de alterar sua capacidade de interação mútua ou com o ambiente, o que cada elemento faz é necessariamente o que foi projetado para fazer. Para ampliar este paradigma o presente trabalho sugere uma nova metodologia denominada de Agentes Polimórficos Dinâmicos Inteligentes (APDI).

A idéia de polimorfismo não é nova. Dentro da teoria de análise e programação orientada a objetos, que surgiu nos anos 80, o polimorfismo é um conceito utilizado para se evitar redundância de implementação e aumentar a reutilização de elementos nucleares. Dentro deste conceito, componentes básicos são implementados e utilizados por elementos mais complexos através de herança. Os novos elementos derivados herdam as propriedades e métodos de diversas classes básicas possibilitando, desta forma, possuir novas funcionalidades. Apesar de ser flexível, o polimorfismo é um conceito utilizado geralmente de forma estática, ou seja, é implementado somente durante o processo de modelagem do problema e uma vez estando operante, as classes derivadas ficam estáticas em relação a novas propriedades e métodos.

Um conceito mais flexível, a Programação Orientada a Aspectos (AOP) [4], surgiu no final dos anos 90 como uma necessidade de se resolver problemas relacionados ao entrelaçamento de métodos e funcionalidades em sistemas de alta complexidade. A presença de funcionalidades que cruzam várias classes deixa o sistema com baixa reutilização e difícil modularização. Logo, a idéia de aspecto vem a ser um nível a mais de abstração, ou seja, a visualização de partes do problema como sendo genéricas, quase ao nível de algoritmo, e que podem ser atribuídas a um dado método ou objeto que possuam uma assinatura específica durante o tempo de execução.

Para se entender bem este conceito é necessário se aprofundar no conhecimento de como uma linguagem de alto nível trata as chamadas aos métodos. Cada função presente em um objeto possui uma assinatura e é referente a informações como o nome do método, o tipo de dado de retorno, quantos e quais os tipos de dados dos parâmetros de entrada dentre outras. Este conjunto de informações presente na assinatura é denominado metadado. Quando uma chamada é realizada a um dado método o que é enviado é seu respectivo metadado. O objeto invocado que recebe a mensagem, a retorna após ter realizado o processamento e preenchido a saída. Logo, com os metadados definidos através de um padrão, é possível criar novas chamadas a diferentes tipos de métodos presentes em diferentes tipos de objetos em tempo de execução, sem que estas tenham sido previamente implementadas durante o projeto do agente. Isso permite selecionar, durante o tempo de execução, o objeto que será responsável pelo cumprimento de um dado algoritmo. Este procedimento permite grande reutilização de código, portabilidade e modularidade. Porém, apesar de toda a flexibilidade apresentada onde, através de AOP, é possível realizar a chamada de diversos objetos a partir de um único algoritmo, o algoritmo em si continua sendo estático. É neste ponto onde se encaixa o conceito do Agente Polimórfico Dinâmico.

Com a possibilidade de acesso às informações de um dado componente, é possível implementar um agente que identifique a presença de outros, que importe metadados externos e crie funções virtuais derivadas que irão invocar os métodos

básicos. Por poder identificar outros agentes e assumir sua assinatura de funções, a estrutura proposta é polimórfica e, por realizar isto em tempo de execução é dinâmica. Porém o agente apresentado é de pouca utilidade se não for capaz de reconhecer o conhecimento adquirido e, frente a um problema, não conseguir identificar se é possível ou não resolvê-lo.

Desta forma, para completar a metodologia, cada metadado, ou seja, cada função de um dado agente deve ter instruções de utilização específicas, denominadas de hiperdados, descrevendo sua funcionalidade, suas entradas e suas saídas. Um planejador (sistema onde, dado um estado inicial se utiliza de busca heurística para formular um plano de ações para assim atingir um determinado estado objetivo), através dos hiperdados, consegue ler os dados que um problema fornece como entrada, verificar todas as funcionalidades presentes no APD e apresentar um ou mais planos de ação. Este planejamento de ações dá uma característica de inteligecia ao Agente, e a combinação desta funcionalidade com as outras funcionalidades presentes no APD cria o que foi definido neste trabalho de dissertação como sendo Agente Polimórfico Dinâmico Inteligente (APDI).

Para apresentar esta proposta, como descrito no corpo desta introdução, é necessária uma base teórica de alguns conceitos bem distintos. Sendo assim, o trabalho está estruturado da seguinte forma; No capítulo 2 está a conceituação de agentes inteligentes, juntamente com discussões e revisões bibliográficas pertinentes ao assunto. No capítulo 3 é apresentado um detalhamento de Planejamento em Inteligência Artificial, bem como a apresentação das características e ferramentas de linguagem utilizadas. No capítulo 4 é apresentado o conceito de metadados e sua utilidade para este trabalho. No capítulo 5 é apresentada programação orientada a aspectos, fundamental para o dinamismo do agente proposto nesta dissertação. No capítulo 6 é apresentada a metodologia de APDI proposta neste trabalho juntamente com sua arquitetura. No capítulo 7 é apresentado um exemplo de utilização, ou seja, um domínio criado para validar a metodologia e os resultados obtidos da aplicação. Por fim, no capítulo 8 é

apresentada a conclusão com relação à metodologia e resultados, além das propostas de trabalho futuro.

2. Agentes Inteligentes

2.1. Introdução

Agentes inteligentes de software são atualmente objetos de pesquisa em muitos campos tais como psicologia, sociologia e inteligência artificial [5].

Atualmente existem várias definições de agentes, algumas até mesmo contraditórias. A verdade é que não existe nenhuma definição oficial ou universalmente aceita de Agentes Inteligentes, e este capítulo tratará de Agentes Inteligentes com referência a alguns autores já consagrados e bem aceitos no mundo acadêmico, como Russel & Norvig [6] e Frankling & Graesser [7], com o objetivo de mostrar seus principais conceitos e características.

2.2. Definição

De modo a tentar minimizar a discordância sobre o conceito de agentes, em [7] é feita uma crítica a diversas definições encontradas na literatura, buscando assim uma taxonomia clara para agentes autônomos.

Definiu-se agente como sendo “sistemas situados e parte de um ambiente, no qual percebem e agem, durante um período de tempo, de modo a seguirem seu cronograma próprio, provocando, assim, percepções futuras”.

Analisando cada item da definição, podemos entender que o ambiente a que ele se refere pode ser um ambiente físico ou eletrônico (Virtual, Internet), a percepção e ação podem ser feitas por sensores e atuadores ou rotinas de software que funcionam como sensores e atuadores. Suas ações modificam o ambiente e são executadas durante um determinado período de tempo.

Segundo a mesma referência, agentes podem possuir as seguintes propriedades: reativo, autônomo, orientado a objetivo, temporariamente contínuo, comunicativo, adaptativo, móvel, flexível, e com caráter.

2.3. Propriedades dos Agentes Inteligentes

Ainda segundo [7] todo agente, pela definição apresentada, deve possuir, pelo menos, as quatro primeiras propriedades apresentadas: Reativo, Autônomo, Orientado a Objetivo e Temporariamente Contínuo.

Desta forma, o modelo proposto neste trabalho se enquadra nos seguintes itens de definição:

- Reatividade (responde, em uma fração de tempo, às mudanças no ambiente);

- Autonomia (exerce controle sobre suas próprias ações);

- Orientação a objetivo (não age simplesmente em resposta ao ambiente);

- Continuidade temporal (é um processo executável contínuo);

- Comunicação (se comunica com outros agentes);

- Adaptação (muda seu comportamento devido a uma necessidade).

Uma importante diferença a ser observada nas definições da literatura para a utilizada neste trabalho é a necessidade do agente proposto estar sobre um

framework[1] de mensagens. Este *framework*, apesar de não ser parte integrante do agente, é responsável pela troca de informações entre os agentes e pela execução de métodos de novas entidades criadas a partir dos agentes originais. Este assunto será tratado no capítulo 4.

2.4. Estrutura dos Agentes Inteligentes

Para entender melhor a estrutura dos Agentes Inteligentes e tomando ainda por base a definição de agente apresentada por [7], um agente humano será utilizado como exemplo.

Um agente humano tem seus olhos, ouvidos, nariz, entre outros órgãos como sensores, e também tem suas mãos, boca, pernas como atuadores. Portanto, um agente robótico ou de software possui, no lugar dos órgãos sensores, câmeras, sensores de infravermelho, *encoders*, eventos de *softwares* entre outros tipos de sensores. Já no lugar dos órgãos atuadores, podem ter motores, pistões, ou uma rotina de *software*, entre outros tipos de atuadores.

A Figura 1 ilustra de forma genérica um Agente Inteligente segundo [6].

[1] Um framework usa um conjunto de classes e interfaces e mostra como decompor a família de problemas, O conjunto de classes deve ser flexível e extensível para permitir a construção de várias aplicações com pouco esforço, especificando apenas as particularidades de cada aplicação.

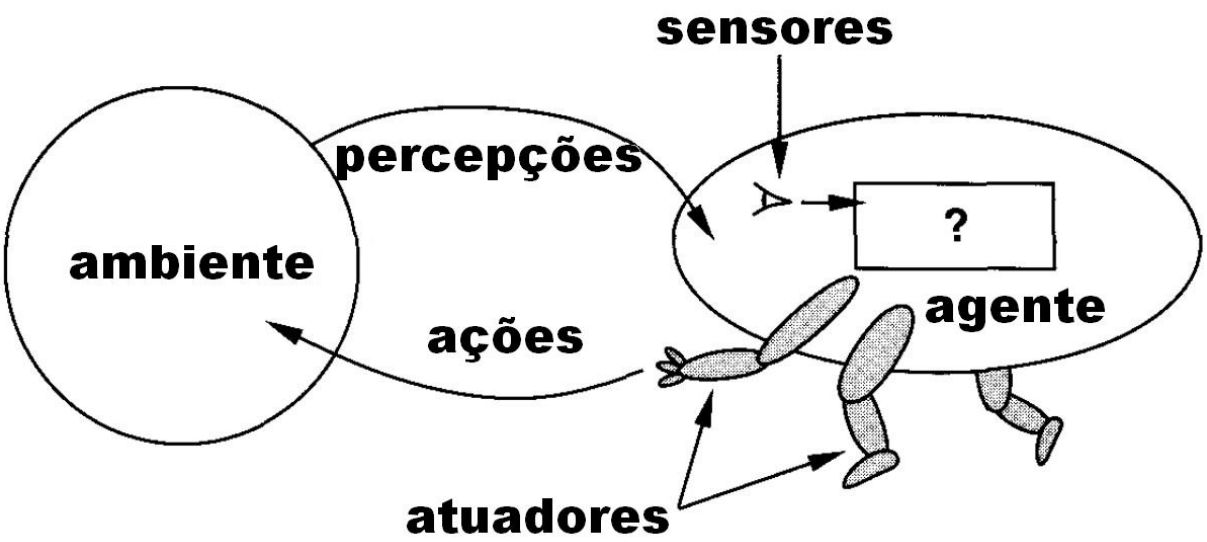

Figura 1 – Agente interage com o ambiente através de sensores e atuadores

2.5. Sistemas Multiagentes

São sistemas onde vários agentes são utilizados e/ou se interagem para a resolução de um problema e são conhecidos como MAS (Multi-Agent Systems).

Estes sistemas são compostos de agentes individuais que trabalham juntos para resolver problemas complexos, como exemplo um sistema de apoio à decisão, onde um diagnóstico final pode ser gerado, baseado no diagnóstico de cada agente individualmente, ou como no caso deste trabalho, onde determinados problemas não podem ser resolvidos por um agente específico. Tipicamente estes agentes são estáticos, isto é, não são modificados ao longo do tempo.

De acordo com Scherer & Schlageter [8], o principal ponto de pesquisa em MAS está focado na coordenação. Estes autores continuam afirmando que as abordagens MAS atuais possuem um aspecto inerentemente estático. MAS são projetados para seguir uma abordagem *top-down*, onde, normalmente, não é permitido a adição ou remoção dinâmica de agentes.

2.6. Linguagens de comunicação entre agentes

Uma necessidade básica para que os agentes cooperem e se comuniquem é que eles devem falar uma linguagem comum seguindo um protocolo padrão, caso contrário a troca de informações se torna nebulosa ou até inexistente.

A forma de comunicação ideal, assim como a definição de Agentes Inteligentes, não possui uma definição formal tão pouco um padrão universal, desta forma diversos autores sugerem alguns mecanismos como a Linguagem Formal Derivada do Inglês sugerida por [6].

Neste trabalho, pode-se dizer que a comunicação dos agentes será feita através de um framework de mensagens, que será inteiramente explanado no capítulo 4.

2.7. Resolução distribuída de problemas

Alguns autores diferenciam a Resolução Distribuída de Problemas (DPS Distributed Problem Solving) dos Sistemas Multiagentes (MAS – Multi Agent Systems). No DPS, é considerado que a tarefa de resolver um problema particular pode ser dividida em um número de módulos (ou nodos) que cooperam uns com os outros, dividindo e compartilhando conhecimento sobre o problema e sobre o processo de obter uma solução. Já as pesquisas em MAS estão envolvidas com o comportamento de uma coleção de agentes autônomos visando resolver um dado problema.

Em Edmund & Jeffrey [9], a diferença e a relação entre DPS e MAS é amplamente discutida, e são mostradas três dos principais pontos de vista com relação aos dois sistemas.

No primeiro ponto de vista, o DPS é visto como sendo um subconjunto dos sistemas MAS. Nesta situação, um sistema MAS assegura algumas propriedades como metas comuns e projeto centralizado.

Em um segundo ponto de vista os sistemas MAS fornecem a base para os sistemas DPS. Sendo assim, um sistema MAS procura trabalhar com a individualidade e o interesse próprio de cada agente para definir como serão feitas as interações com outros agentes. O sistema DPS procura então, verificar como estas interações podem ser exploradas de forma a resolver algum problema mais global.

Na terceira proposta, os sistemas MAS são vistos como sendo complementares aos sistemas DPS. Neste caso, um sistema MAS procura responder de que forma algumas propriedades coletivas podem ser percebidas em um ambiente particular. Enquanto, sistemas DPS procuram responder como uma coleção particular de agentes pode obter algum desempenho eficiente se as propriedades do ambiente são dinâmicas e não controláveis.

Ainda segundo os autores, a questão de qual é a melhor visão, é um campo aberto a discussões e respostas. Se esta discussão é importante ou não, também é uma questão de opinião, mas a habilidade de se diferenciar as características de cada um dos dois sistemas na construção de um sistema pode significar uma grande vantagem.

Em [10], por exemplo, uma discussão relacionada a sistemas Multi Agentes é feita onde os MAS são classificados como “Sistemas onde cada agente tem informações ou habilidades incompletas para a resolução do problema, portanto com campo de visão limitado. Não possuem controle global do sistema, os dados são descentralizados e a computação é assíncrona”. Além da diferença conceitual, nota-se que no artigo a Resolução Distribuída de Problemas (DPS) não é sequer citada durante toda a discussão, ou seja, para muitos a diferença entre os dois sistemas é irrelevante e o termo MAS é explicitamente bem mais utilizado na comunidade científica, muitas vezes de forma referencialmente errônea.

Partindo do ponto, onde a diferença entre os dois sistemas é uma questão de opinião, voltando a [9], uma diferenciação interessante é proposta baseada no terceiro ponto de vista mencionado anteriormente onde os dois sistemas são complementares.

MAS: Corresponde a uma área de pesquisa que se foca nas propriedades internas de um sistema de agentes, onde as propriedades individuais de cada agente podem variar. Portanto, MAS se concentra em como agentes com preferências individuais irão se interagir para chegar ao objetivo final. MAS sempre se preocupa em como as propriedades coletivas podem ser alcançadas, se as propriedades de cada agente podem variar indefinidamente ou não.

DPS: O foco é em extrair propriedades externas como robustez e eficiência, quando o ambiente tem suas propriedades e condições variáveis, mas os agentes têm propriedades fixas. DPS sempre se preocupa em como um determinado conjunto de agentes atinge algum nível de desempenho coletivo, sendo que as propriedades do ambiente podem variar indefinidamente ou não.

A tabela a seguir, extraída de [9], ilustra melhor estas diferenças.

Tabela 1 – Matriz de Propriedades Variantes

	Propriedade dos Agentes	**Propriedades do Ambiente**	**Propriedades do sistema**
MAS	Variáveis	Fixas	Fixas (interna)
DPS	Fixas	Fixas	Fixas (externa)

Considerando esta última visão, o sistema proposto neste trabalho poderia ser classificado como um DPS, levando em consideração que cada agente que entra ou sai do sistema, identificados pelo agente monitor, assim como o próprio agente monitor, possui suas propriedades fixas e são utilizados em conjunto para a resolução de um problema. Em contrapartida, o ambiente (problemas propostos) possui suas propriedades variáveis. Por outro lado, o Agente Polimórfico Dinâmico

Inteligente (APDI), que é o objetivo deste trabalho, possui suas propriedades variáveis de acordo com mudanças nas propriedades do ambiente e coopera com outros agentes como o Agente Monitor que coordena a maioria das ações dos demais agentes (Agente Planejador, Agente Problema e APDI – estes agentes são detalhados no Capítulo 6). Este sistema de agentes em cooperação e com uma coordenação pode ser considerado um MAS.

Portanto, devido à questão de interpretação, e ainda segundo as conclusões de [9], ("a distinção entre os dois sistemas se perdeu com o tempo e é hoje uma questão cultural e de discussão"). Com isso, o sistema proposto neste trabalho está sendo considerado um Sistema Multiagente com Resolução Distribuída de Problemas.

3. Planejamento em Inteligência Artificial

3.1. Introdução

Uma revisão bibliográfica e histórica da Inteligência artificial mostra opiniões bastante diferentes e até mesmo contraditórias a respeito da definição formal, ou do conceito de "Inteligência Artificial". Entrar neste mérito não é o objetivo deste tópico.

Os Sistemas Inteligentes são, de forma geral, sistemas que possuem alguma habilidade ou conhecimento para realizar determinadas tarefas. Estes sistemas geralmente decidem por si próprios como executar um plano de ações dados os objetivos e as características do ambiente ou domínio em questão.

O Planejamento é a área da Inteligência Artificial que estuda a elaboração de um plano de ações, dados os objetivos e as características do ambiente ou domínio em questão. Portanto, pode-se fazer uma analogia com o parágrafo acima, onde o Planejamento em Inteligência Artificial é um sistema onde dado "O que" fazer, consegue determinar "Como" e "Quando" fazer, ou seja soluciona um problema.

Segundo Gaffner Héctor [11] o desenvolvimento de um "Solucionador Geral de Problemas", ou GPS (*General Problem Solver)*, tem sido um dos principais objetivos na Inteligência Artificial. Um GPS é um programa que aceita descrições em alto nível de problemas e automaticamente computa uma [12].

Após a introdução do primeiro planejador por Newell e Simon [11], o Planejamento em Inteligência Artificial foi se modificando, se tornando mais matemático (uma grande variedade de problemas de planejamento foi definida e estudada). O sistema STRIPS [13], que surgiu na década de 70, foi efetivamente o

primeiro sistema de planejamento que se caracterizava por procurar uma seqüência de ações que permitia transformar o estado inicial do mundo em um estado no qual o objetivo era satisfeito. Foi também o primeiro planejador que alcançou um relativo sucesso. A partir do marco gerado pelo STRIPS, o planejamento passou a ser tratado como um problema de busca no espaço de estados principalmente. Desde então, as pesquisas foram sendo dedicadas à elaboração de algoritmos de planejamento mais eficientes.

Atualmente, uma competição internacional de Planejamento (International Planning Competitionn – IPC) [14] é realizada a cada dois anos. Esta competição conseguiu gerar uma linguagem padrão de planejamento, permitindo, desta forma, uma comparação mais específica entre as diversas técnicas de planejamento. O efeito gerado na comunidade científica, causado por esta competição, é o desenvolvimento de novas ferramentas de planejamento bem como o incentivo para o aumento do desempenho dos planejadores.

O Planejamento em Inteligência Artificial estuda linguagens, modelos e algoritmos para descrever e solucionar problemas que envolvem a seleção de ações para atingir um determinado objetivo. No Planejamento Clássico, que é a forma mais simples de planejamento, as ações são consideradas determinísticas, porém a própria Competição Internacional de Planejamento citada anteriormente já possui uma divisão entre as partes determinísticas e não-determinísticas do Planejamento em IA. Mas em ambos os casos a tarefa de um planejador é computar uma solução ou plano de ações.

São várias as metodologias apresentadas na literatura para o desenvolvimento de um planejador, como exemplo, nesta parte do trabalho pode ser citado um algoritmo de [6] por sua simplicidade. Este algoritmo é ilustrado pela Figura 2.

De forma sucinta o algoritmo se baseia em uma busca em árvore onde cada estado corrente é analisado para verificar as condições presentes. Estas condições são cruzadas com as informações contidas nas pré-condições e as possíveis ações

são adicionadas a uma árvore. Associada com cada ação está uma pós-condição, ou seja, os efeitos: a informação do que muda no estado corrente se uma dada função for realizada.

Novamente o estado corrente é analisado, novas ações adicionadas à árvore como filhas e este processo continua até que um nível máximo na árvore seja atingido ou que o estado desejado seja alcançado.

Estratégias de busca em profundidade ou amplitude podem ser selecionadas. A Figura 2 ilustra o processo descrito.

Supondo que o estado corrente do ambiente é representado pelas condições A, B e C. São analisadas todas as ações existentes, para se verificar quais destas ações poderiam ser executadas a partir deste conjunto de condições (A, B e C), ou seja, quais ações têm suas pré-condições satisfeitas pelo estado corrente do ambiente. A “Ação 1” portanto é identificada como a única ação possível de ser executada. Após a execução da “Ação 1”, é gerado um novo conjunto de condições (X, Y e Z). Este novo conjunto de condições satisfaz as pré-condições de duas outras ações “Ação 2” e “Ação 3”, que se executadas gerariam um novo estado corrente do ambiente. Este processo se repete até que seja encontrado um estado onde todas as condições determinadas pelo objetivo do problema sejam satisfeitas ou até que o espaço de busca seja totalmente percorrido.

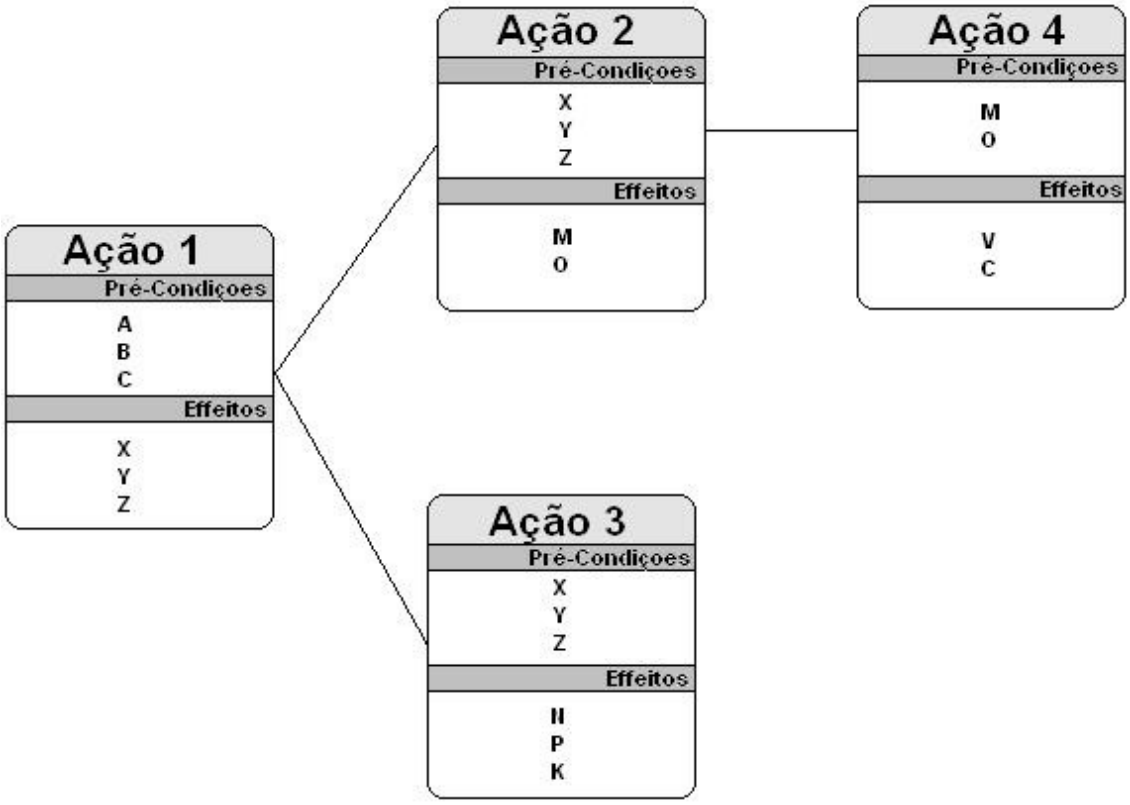

Figura 2 – Busca da Solução em árvore

3.2. Planejamento de Domínios Específicos x Planejamento de Domínios Independentes

A grande motivação de se criar um Solucionador Geral de Problemas é em parte, ainda segundo [11], devido ao fato de que nós Humanos somos de certa forma "Solucionadores de Problemas". Porém não somos Solucionadores "Gerais" de Problemas, uma vez que só conseguimos resolver determinado problema após adquirir o conhecimento específico necessário. Após a década de 60, considerada por muitos como o período "negro" da Inteligência Artificial, começaram a surgir os primeiros sistemas especialistas, talvez motivados pelo fato de que se somos especialistas em algo, fazemos de forma correta e otimizada. Um simples exemplo

pode ilustrar isso, se pensarmos em um Doutor especialista em Inteligência Artificial, que não consegue preparar um simples bolo de chocolate.

Analisado este histórico, os primeiros planejadores, efetivamente utilizados eram planejadores de Domínios específicos, ou seja, eram sistemas especialistas, que solucionavam problemas limitados a um determinado domínio.

Com o surgimento de linguagens padronizadas para descrição de domínios e problemas de planejamento, como é o caso de STRIPS [13], e ADL [15], o desenvolvimento de planejadores independente de domínio foi encorajado. Com o surgimento da primeira Competição Internacional de Planejamento em 1998 - (IPC 98), surgiu também a linguagem PDDL que será apresentada em seguida. Esta linguagem fez com que todos os desenvolvedores de planejadores a utilizassem como linguagem de entrada para o algoritmo de planejamento (planejador).

3.3. A Linguagem PDDL

Essa seção descreve a linguagem PDDL *(Planning Domain Definition Language)* [16] que permite a descrição de domínio e problemas de planejamento em uma forma padronizada.

A Linguagem PDDL, foi criada em 1998 para a 1ª Competição Internacional de Planejamento. Tem como base a linguagem STRIPS além de incorporar outras extensões e funcionalidades.

O objetivo da criação da linguagem foi além da padronização, uma maior facilidade de se comparar o desempenho dos planejadores além de aumentar a interoperabilidade entre eles. Com isso Alfonso Gerevini e Derek Long criaram a descrição BNF (*Backus Naur Form*) da linguagem [17], e uma série de problemas e domínios. Os competidores tiveram então que desenvolver um interpretador (*parser*) da linguagem para que seus respectivos algoritmos pudessem buscar uma solução.

De forma simplificada os componentes da linguagem PDDL são apresentados a seguir:

- Objetos: Todos os elementos do Domínio/Mundo que são pertinentes ao problema;
- Predicados: Propriedades e relações entre os objetos. Podem ser verdadeiros ou falsos;
- Estado Inicial: É a descrição do Mundo em seu estado inicial. São os Objetos instanciados;
- Descrição dos Objetivos: O estado desejado do Mundo, ou os predicados que devem ser verdadeiros para se alcançar o Objetivo;
- Ações/Operadores: Descrição dos meios de como modificar o Mundo;

Um pequeno exemplo da linguagem pode ser analisado no domínio mostrado a seguir no Exemplo 1:

```
(1)  (define (domain Viagem)
(2)  (:requirements  :strips :typing)
(3)  (:types  cidade - object )
(4)  (:predicates        (Existe-Estrada-Entre  ?x - cidade  ?y - cidade )
                         (Estou-em  ?x - cidade ) )
(5)  (:action Viajar
(6)          :parameters (?a - cidade ?b - cidade )
(7)          :precondition  (and        (Estou-em ?a )
                                        (Existe-Estrada-Entre ?a ?b ))
(8)          :effect                    (and   (Estou-em ?b )
                                        (not (Estou-em ?a ) ) ) ))
```

Exemplo 1 – Domínio “Viagem”

Em (1) é onde se define o nome do domínio a ser trabalhado, que no caso do exemplo é “Viagem”.

Em (2), o campo “(:requirements” indica quais as funcionalidades que o planejador precisa ter para resolver um problema relacionado ao domínio de trabalho.

Em (3), o campo ”(:types” informa quais os tipos de objetos possíveis no domínio. Um tipo pode ter sub-tipos, como, por exemplo, pode-se criar um tipo chamado “lugar” e um outro tipo chamado “cidade”, onde é informado que cidade também é um “lugar”. Maiores particularidades da linguagem podem ser encontrados em [16] e [17].

Em (4), o campo “(:predicates” é onde são declarados todos os predicados que eventualmente poderão ser utilizados na criação do domínio. Os predicados indicam propriedades de objetos ou relação entre eles e não precisam necessariamente serem “tipados”, ou seja, desta forma qualquer objeto pode ser usado no predicado.

Após todos estes componentes da linguagem, em (5), são declaradas todas as ações/operadores “(:action” do domínio. Com a finalidade de simplicidade, este domínio tem apenas uma ação que tem o nome de “Viajar”.

Em (6), no campo “(:parameters” da ação são apresentados os parâmetros variáveis que serão utilizados para realizar a ação.

Em (7), o campo “(:precondition” mostra todos os predicados, também chamados de “átomos”, e todas as condições que precisam ser verdadeiras para que a ação seja realizada. O campo “(:effect”, em (8), cria ou apaga os átomos de acordo com a lógica dos operadores utilizados.

Em português esta ação poderia ser traduzida da seguinte forma: “Para viajar da cidade ?a para a cidade ?b, é necessário estar em ?a, e existir uma estrada que leva de ?a até ?b. Caso sim, não estarei mais em ?a e estarei agora em ?b”.

A seguir, no Exemplo 2 uma demontração de um problema em PDDL para o domínio anterior:

(1) (define (problem ViagemGoianiaItapaci)

(2) (:domain Viagem)

(3) (:objects

Itapaci - cidade

Goiania - cidade

Jaragua - cidade)

(4) (:init

(Existe-Estrada-Entre Goiania Jaragua)

(Existe-Estrada-Entre Jaragua Itapaci)

(Estou-em Goiania))

(5) (:goal (Estou-em Itapaci)))

Exemplo 2 – Problema exemplo do Domínio “Viagem”

No arquivo problema em PDDL, é onde são instanciados os objetos e predicados que o planejador necessitará pra montar o espaço de busca.

Em (1), no campo “(:define”, é declarado um nome qualquer para o problema que neste caso é “ViagemGoiâniItapaci”. Em (2), no camo “(:domain é indicado o nome do domínio a ser utilizado no problema que no caso é “Viagem”, nome utilizado para o domínio do Exemplo 1.

Em (3), no campo “(:objects” são definidos os objetos do problema, ou seja, as variáveis na forma instanciada. Quando pertinente o tipo de tais objetos deve ser declarado logo em seguida. No caso todos os objetos declarados são do tipo “cidade”;

Em (4), no campo “(:init” é onde os predicados do sistema são instanciados, ou seja, é onde são definidos todos os estados iniciais do problema em questão. Finalmente em (5), “(:goal”, é onde são definidos os estados finais do problema ou os objetivos do problema que neste caso é apenas um: “Estar na cidade de Itapaci”.

Analogamente à descrição do domínio, em português este problema pode ser traduzido da seguinte forma: “Quero estar em Itapaci. Dado que existem três cidades, Itapaci, Goiânia e Jaraguá, sabendo que estou em Goiânia, e que existe estrada entre Goiânia e Jaraguá e entre Jaraguá e Itapaci, como fazer para chegar em Itapaci?”

Os exemplos ilustrados anteriormente mostram de forma fiel, a sintaxe e elementos básicos exigidos pela linguagem PDDL. Apesar da simplicidade, os exemplos, mostram de uma forma didática o nível de descrição da linguagem PDDL.

Os planejadores trabalham com adição e exclusão de átomos. São chamados de “átomos” os predicados na forma instanciada, ou seja, os predicados (declarados no arquivo de domínio) contendo objetos que foram anteriormente declarados no campo “(:objects” do arquivo PDDL de problema . Tudo que o componente da linguagem PDDL “effect” faz é excluir ou adicionar átomos. À medida que o planejador vai analisando as ações, vai montando uma árvore de busca onde cada nó da árvore corresponde a um estado do mundo, caso determinada ação seja executada. Portanto, para o problema exemplificado, uma visualização do trabalho do planejador pode ser vista na Figura 3, onde em (1) tem-se o estado inicial do sistema indicando a existência de estrada entre as cidades de Goiânia e Jaraguá e entre Jaraguá e Itapací, além da indicação que o sujeito encontra-se em Goiânia. Em (2) o planejador realiza o primeiro passo de sua busca. Como para este domínio, devido à didática, só existe uma ação possível, o planejador tentará aplicar todos os parâmetros existentes (Objetos instanciados) nesta ação gerando um ramo para cada combinação de parâmetros. Como o sujeito encontra-se em Goiânia, o único parâmetro que pode ser testado juntamente com “Goiânia” é o parâmetro “Jaraguá”. Isto é possível devido à modelagem do domínio, que no campo “(:preconditions” é

definido que para de viajar de ‘A’ a ‘B’, é preciso estar em ‘A’ e é preciso existir uma estrada que liga ‘A’ a ‘B’. Após a realização desta ação, um novo estado é gerado, onde o átomo que indica a presença do sujeito em Goiânia (Estou-em Goiania) é deletado, e o átomo representando que o sujeito está em Jaraguá (Estou-em Jaraguá) é criado.

De forma análoga, do nó (2) o planejador irá analisar todos os ramos gerados na árvore de busca. Agora podem ser gerados 2 ramos distintos (3) e (4). A solução é encontrada em (3). Isto ocorre, pois a cada novo estado gerado, o planejador faz uma comparação com o estado objetivo, se as condições do estado final forem satisfeitas, significa que foi encontrada uma solução.

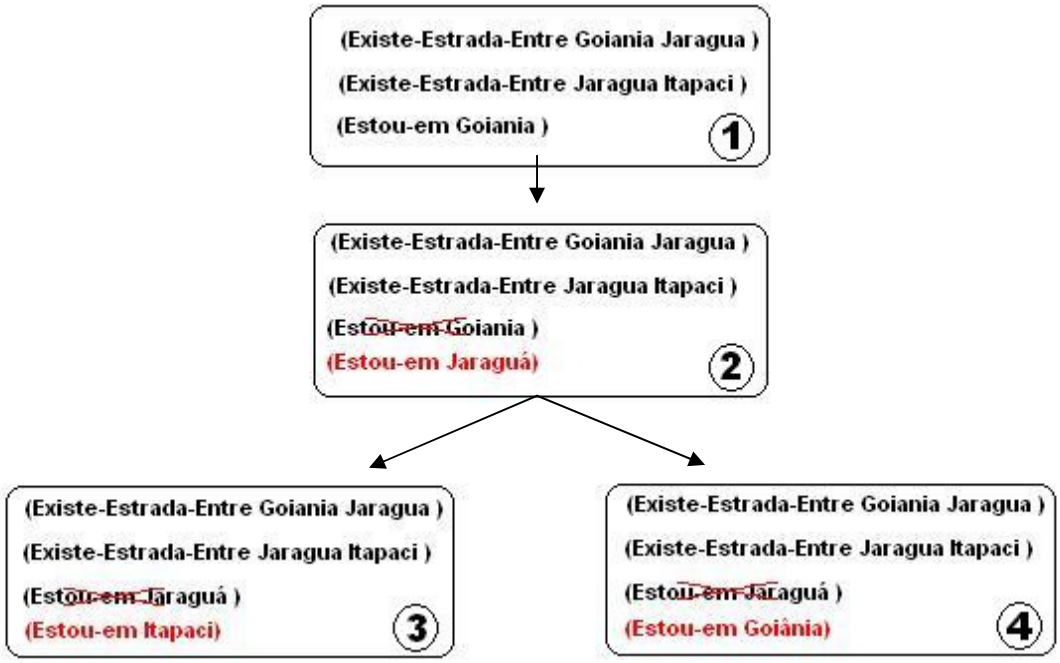

Figura 3 – Árvore de busca do planejador

É importante ressaltar que a Figura 3 apenas ilustra uma técnica de busca no espaço de estados. Os algoritmos utilizados pelos planejadores possuem uma

complexidade muito maior, incorporando várias técnicas e inteligência como limitar a profundidade ou largura da busca, evitar estados repetidos, etc.

A Figura 4 mostra um plano gerado pelo planejador FF [20] para o domínio e problema apresentados anteriormente, ou seja o plano pode ser executado em dois passos:

Passo 0: Viajar de Goiânia a Jaraguá.

Passo 1: Viajar de Jaraguá a Itapaci

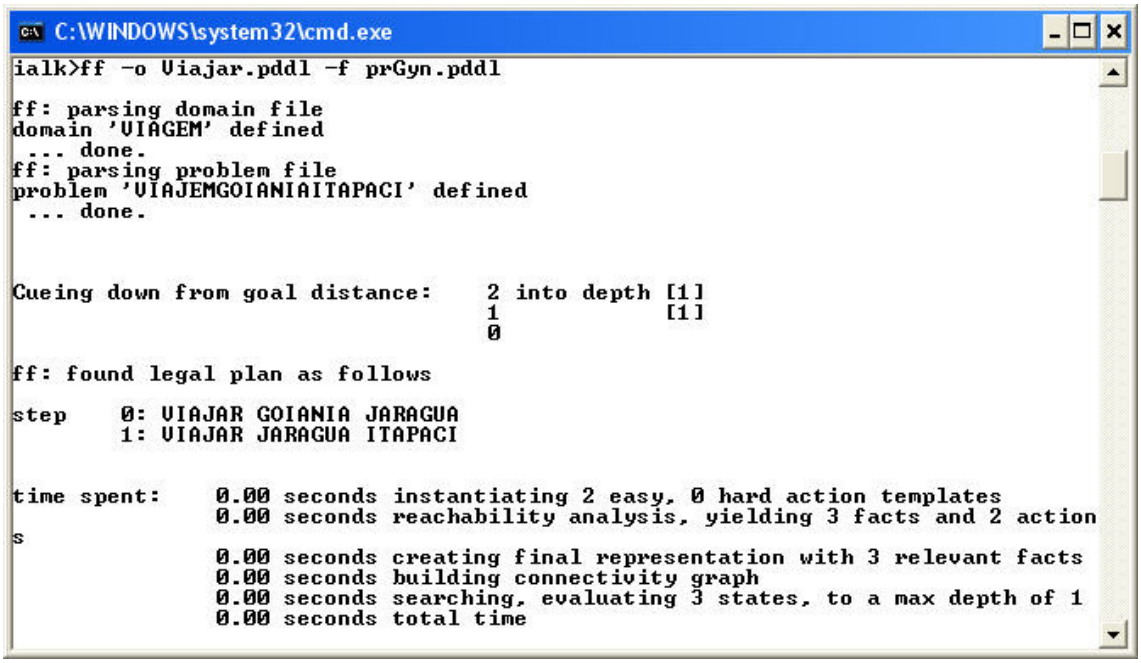

Figura 4 - Plano Gerado

A linguagem PDDL se encontra hoje na versão 3.0 [17], incorporando vários elementos e funcionalidades como otimização (minimização e maximização), ações com durações temporais, funções métricas (predicados não booleanos), probabilidade entre outras coisas.

Porém os planejadores mais novos, principalmente os que atingiram uma boa colocação nas últimas competições de planejamento, não são tão facilmente acessíveis para utilização. Muitas vezes não são disponibilizados na Internet, e alguns quando o são, é em forma de código fonte que precisa ser compilado por um compilador específico de uma distribuição específica do Sistema Operacional.

Mas muitos planejadores confiáveis e rápidos, porém um pouco antigos, são facilmente achados e serão comentados no item 3.4 deste capítulo. Cada planejador trabalha com uma versão específica do PDDL e praticamente nenhum trabalha atendendo a todos os requisitos da linguagem. Todos têm suas limitações, e por isso se torna uma tarefa difícil o fato de se definir qual versão da linguagem foi utilizada neste trabalho.

3.4. Planejadores utilizados

Dentro das necessidades dos domínios utilizados neste trabalho, e dentro de um estudo feito de Planejamento em Inteligência Artificial, foram analisados os seguintes planejadores: AltAlt [18], Blackbox [19], FF [20], GPCSP [21], LPG-TD [22], STAN [23] e VHPOP [24]. Destes, dois receberam uma maior atenção por motivos de aplicação aos domínios utilizados neste trabalho, que foram o FF e LPG-TD e que serão apresentados a seguir.

O planejador FF, acrônimo para *Fast Foward*, utiliza o algoritmo de busca *Subida de Encosta* reforçado por uma heurística (*reinforced Hill-Climbing*). O algoritmo *Subida de Encosta* reforçado [25] efetua uma busca local pelo espaço de estados, orientada por uma função heurística. O espaço de estado em qualquer problema de planejamento não trivial é muito grande, ou seja computacionalmente difícil. Por esta razão, planejadores que fazem busca com esta técnica são dependentes de boas funções heurísticas para guiar a busca.

Um planejador de espaço de estado progressivo executa busca a partir do estado inicial (busca de encadeamento progressivo) até encontrar um estado no qual o objetivo especificado é satisfeito. A Figura 5 ilustra uma busca progressiva no espaço de estados. Na figura, os estados são representados pela letra 'S', sendo S_0 o estado inicial, S_g o estado objetivo e S_n os demais estados possíveis. As ações são representadas pela letra 'a'. Através da aplicação de ações (a_1, a_2, a_n), novos estados vão sendo gerados, e a árvore de busca vai sendo montada até uma solução ser encontrada.

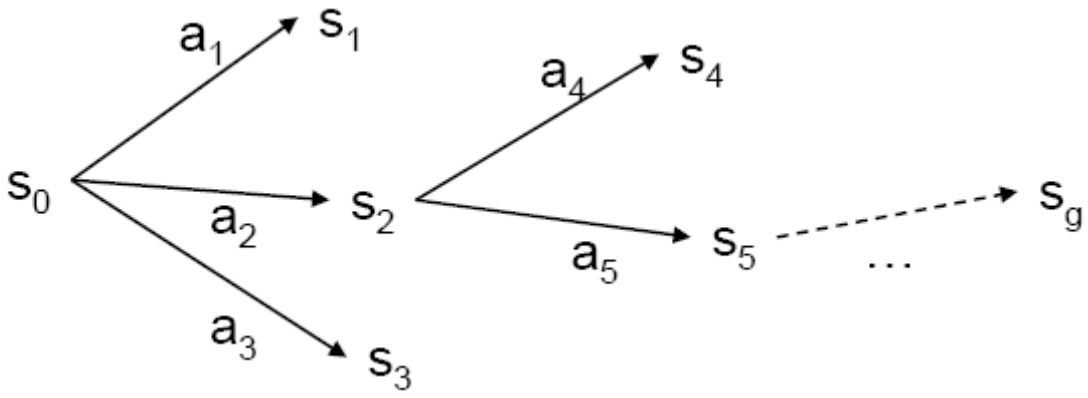

Figura 5 – Busca Progressiva no espaço de estados

A busca no espaço de estado pode ser feita de maneira regressiva (busca de encadeamento regressivo). Ao invés de fazer a busca adiante, a partir do estado inicial, faz-se regressivamente a partir do estado objetivo. A Figura 6 ilustra a busca regressiva no espaço de estados, onde a partir do estado objetivo g_0, e da aplicação das ações de forma reversa, novos estados vão sendo gerados até que o estado inicial S_0 seja encontrado.

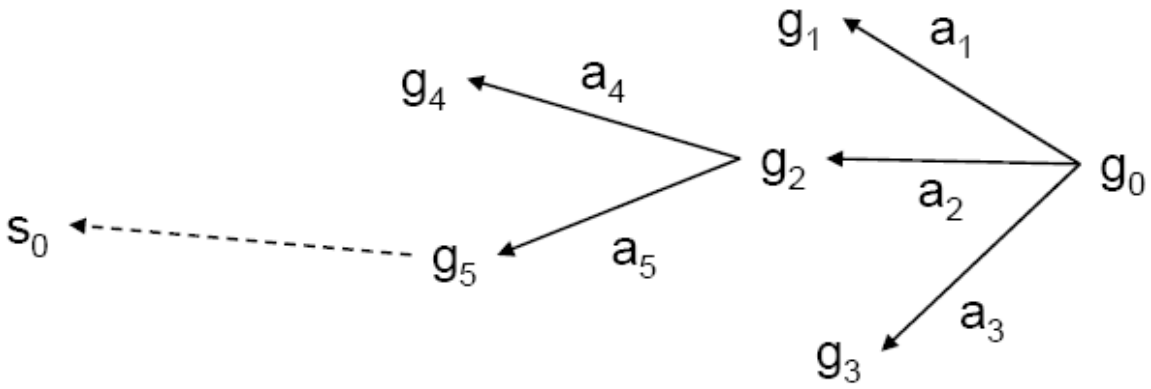

Figura 6 – Busca Regressiva no espaço de estados

No planejador FF os problemas são atacados por uma busca progressiva no espaço de estados, e esta busca é guiada por uma função heurística que é automaticamente extraída da descrição do arquivo de domínio e problema [25]. Esta extração é feita através de grafos de planejamento (APÊNDICE A). Com o grafo de planejamento gerado através dos arquivos PDDL, é montada uma busca "relaxada", ou seja, são desconsiderados alguns fatores pertinentes às ações, como por exemplo, as precondições. Com esta busca "relaxada", a solução é facilmente encontrada ou é detectado que o problema não tem solução. O resultado da função heurística do planejador FF é obtido computando o comprimento da linearização desta solução para a tarefa relaxada. Com isso, cada estado criado durante a busca em árvore, recebe um valor de custo estimado pela heurística. Como a busca é feita em largura pelo FF, o estado que possuir um menor custo é o escolhido para a expansão do nó. Isso é feito até que a solução seja encontrada, ou até que estados comecem a se repetir.

O FF é um planejador consagrado nas competições de planejamento ganhando vários prêmios como "Desempenho Destaque" na 2ª Competição de Planejamento, "Melhor Desempenho no segmento STRIPS e numérico" na 3ª Competição. O FF possui versões novas que competiram em 2006, na 5ª Competição Internacional de Planejamento não tão consagradas como as primeiras.

A versão do FF utilizada neste trabalho é a versão 2.3 utilizada na 3ª Competição Internacional de Planejamento. Esta versão possui funcionalidades e suporte a requerimentos que foram fundamentais para a confecção dos domínios utilizados neste trabalho. Esta versão não dá suporte às funcionalidades métricas (funções, predicados com valores numéricos) do PDDL, porém permite o uso de precondições disjuntivas ("ou", "não" e "implica"), efeitos condicionais ("quando x, então y") e precondições universais ("para todo").

O planejador LPG-TD que também já é consagrado pela comunidade de Planejamento, difere do FF primeiramente no seu método de busca que é uma busca local em grafos de planejamento (APÊNDICE A), técnica que possui desempenho semelhante à busca de espaço de estados progressiva. Porém, ao invés de se montar uma árvore de busca progressiva no espaço de estados, a busca é realizada no grafo de planejamento relaxado, através das técnicas mencionadas em [22].

Porém, o principal fator que difere os dois planejadores é a forma de parametrização da função heurística de busca, de onde pode tirar algumas vantagens e outras desvantagens.

Com certo conhecimento do domínio de trabalho e do problema é possível parametrizar a função heurística do LPG de forma a se conseguir resultados otimizados, velozes ou quantitativos. Este planejador permite isso, ou seja, se o objetivo é várias soluções diferentes, ou uma solução rápida independente da qualidade, ou ainda uma solução ótima independente do tempo de processamento, é totalmente possível escolher. Porém, a forma natural de parametrização da função heurística do LPG-TD é através de uma variável aleatória, ou seja, se este planejador realizar dois planejamentos consecutivos para o mesmo problema, serão encontrados dois planos distintos.

Existe, a possibilidade de se plantar uma semente para a aleatoriedade da heurística. Desta forma, os planos para os mesmos problemas serão sempre iguais,

porém, como será mostrado nos testes a seguir, a qualidade e o tempo de planejamento nem sempre são satisfatórios.

De forma comparativa, a grande funcionalidade do LPG-TD que o FF não possui é a capacidade de realizar planos de ordem parcial, ou seja, um mesmo passo pode conter várias ações distintas. Outra funcionalidade importante do LPG-TD é a capacidade de lidar com predicados derivados (Axiomas), que são predicados que não dependem de ações para serem adicionados ou apagados quando na forma de átomos.

A desvantagem do LPG-TD em relação ao FF fica por conta do não suporte aos efeitos condicionais, ou seja, não se pode usar o operador "*when*" no efeito do plano. Esta limitação juntamente com a velocidade foram os principais fatores que influenciaram para a escolha do planejador utilizado.

A

Tabela 2 ilustra de forma sucinta a comparação entre os dois planejadores. Para efeito de comparação de desempenho de planejadores, a descrição completa do Domínio utilizado não é interessante ainda nesta parte do trabalho, mesmo porque é um domínio diferente do que será descrito no capítulo 7. O domínio testado foi o Mundo dos Blocos Hannoi com Agentes Garra, Esteira e Magazine e sem restrições de permissão de retirada e alocação de blocos. Este domínio foi criado durante as fases de teste deste trabalho, antes da implementação propriamente dita do sistema. Foram feitos inúmeros testes com as finalidades de avaliar os planejadores e auxiliar na escolha do domínio para a validação da metodologia. Com isso foi definido que apenas um resumo destes testes, representado pela

Tabela 2, seria apresentado nesta dissertação.

O problema consiste em empilhar 8 blocos de tamanhos diferentes de forma que os blocos menores ficariam em cima dos maiores. Para a realização da tarefa têm se a disposição Robôs manipuladores, esteiras transportadoras e magazines de armazenamento de blocos, sem restrição de alocação de blocos.

O planejador LPG-TD foi utilizado no modo “speed”, ou seja, com prioridade na velocidade de planejamento, uma vez que as opções “qualidade” e “quantidade” de planos foram testadas, mas devido ao extenso tempo de processamento (cerca de meia hora para a primeira linha da tabela), não foram mais utilizadas. O Planejador FF foi utilizado sem quaisquer parâmetros adicionais.

O formato do problema descrito na tabela descreve o número de garras, esteiras e magazines respectivamente. Como exemplo será tomada a primeira linha de dados da tabela na primeira coluna: **4G2E4M**, onde “**4G**” significa a utilização de 4 “garras” ou robôs manipuladores, “**2E**” significa a utilização de 2 esteiras transportadoras e “**4M**” indica a utilização de 4 magazines armazenadores.

É importante ressaltar que, como discutido anteriormente, o planejador FF expressa seus resultados em forma de passos, sendo uma ação por passo, enquanto o planejador LPG-TD, por ser um planejador de ordem parcial, expressa seu plano também em forma de passos, porém cada passo pode ter uma ou mais ações simultaneamente, não interessando a ordem seqüencial das mesmas. Com isso, os números sem parênteses indicam o número de passos para alcançar o objetivo do problema, enquanto os números entre parênteses (somente LPG-TD) indicam a quantidade de ações necessárias para atingir a solução.

Tabela 2 – Desempenho dos planejadores

Problema	**Número de Passos da Soluçao**	
	LPG-TD (Speed)	**FF**
4G2E4M	104-(148)	57
4G2E2M	39-(64)	55
3G2E0M	59-(84)	83
2G2E0M	191-(212)	53
4G1E0M	32-(63)	51
3G1E0M	140-(170)	229

Mesmo alcançando alguns resultados qualitativamente melhores que o FF, o tempo de planejamento do LPG-TD era na maior parte dos casos cerca de duas vezes superior ao do FF.

Uma dificuldade encontrada durante a fase de testes deste trabalho foi com relação ao número de ações geradas pelos planejadores. Após a geração de um plano de ações, alguns agentes eram retirados do sistema, para verificar o comportamento dos planejadores. Foram constatados casos onde o planejador gerava uma resposta incoerente com o número de ações menor do que o caso anterior. Pode-se notar este efeito comparando-se as linhas 3 e 4 da

Tabela 2 onde na segunda linha o domínio conta com uma Garra a menos e deveria necessitar de mais ações para atingir o objetivo. Porém, o plano é gerado com 30 passos a menos. De forma análoga o mesmo problema ocorria quando a adição de agentes aumentava o número de ações necessárias para resolver um problema. Isto se deve às limitações dos planejadores, que tendo que percorrer uma extensa árvore de busca, nem sempre conseguem atingir uma solução ótima. Quanto mais vão sendo adicionados recursos e agentes, maior se torna a árvore de busca a ser formada e com isso nem sempre o melhor caminho para a solução pode ser encontrado.

Outro problema está no fato de que o planejador não consegue diferenciar os agentes simplesmente através de ações. Ele não tenta minimizar o número de agentes utilizados, pois o objetivo é encontrar um menor número de ações para se resolver o problema. Com os novos recursos da linguagem PDDL como otimização [26], custo de ações [27], preferências [28], entre outras, poderia se definir custos e prioridades para cada agente, o que levaria a uma melhora qualitativa na escolha dos agentes. Porém, para agentes que possuem funcionalidades similares, este problema de escolha de agentes não pode ser solucionado de forma explícita e trivial utilizando apenas PDDL. Para exemplificar este problema a Figura 7 será levada em consideração:

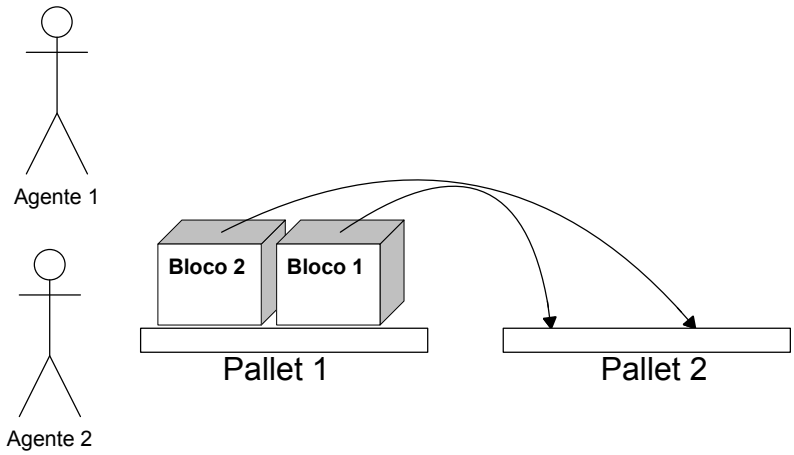

Figura 7 – Alocação de Agentes

Nesta ilustração, suponhe-se que os *Agentes 1 e 2* possuam uma única ação que é a de ***TransportarBloco(PosiçãoInicial, PosiçãoFinal)***. O Objetivo é transportar os *Blocos 1 e 2* do *Pallet 1* até o *Pallet 2*. A solução é encontrada em dois passos, porém dependendo do caminho que o planejador escolher, cada passo pode ser executado por um agente diferente. Sendo o processo seqüencial, dois agentes diferentes foram alocados para realizar uma tarefa que apenas um agente conseguiria realizar.

Para evitar os dois problemas apresentados, foi desenvolvida neste trabalho, uma característica de busca direcionada (Figura 8) onde o sistema irá buscar o menor número de agentes que tenham a capacidade de resolver o problema. Para explicar o funcionamento desta heurística, serão utilizados três componentes da arquitetura do APDI que não foram ainda discutidos até esta etapa da dissertação, mas o serão no capítulo 6. Tal heurística funciona da seguinte forma. É feita a divisão dos agentes em grupos onde o grau de importância de cada grupo é selecionado segundo um sorteio ou de acordo com as necessidades do domínio. Na Figura 8 pode-se notar três diferentes grupos (1, 2 e 3) com diferentes graus de importância representados por pesos (X, Y e Z). Para a primeira tentativa de solução, O AM (que será apresentado no capítulo 6 item 6.2) monta um mínimo de agentes capaz de solucionar a mais simples das tarefas possíveis e apresenta este conjunto juntamente com o problema apresentado pelo Agente Problema. Caso o problema não seja solucionado com estes agentes, uma varredura, realizada na ordem de importância dos grupos, é feita onde a cada iteração um novo agente é adicionado. Esta varredura continua até que uma solução seja encontrada ou até não ter mais agentes disponíveis. Desta forma, apesar de não poder afirmar que a solução encontrada é ótima, pode-se garantir um número menor de agentes.

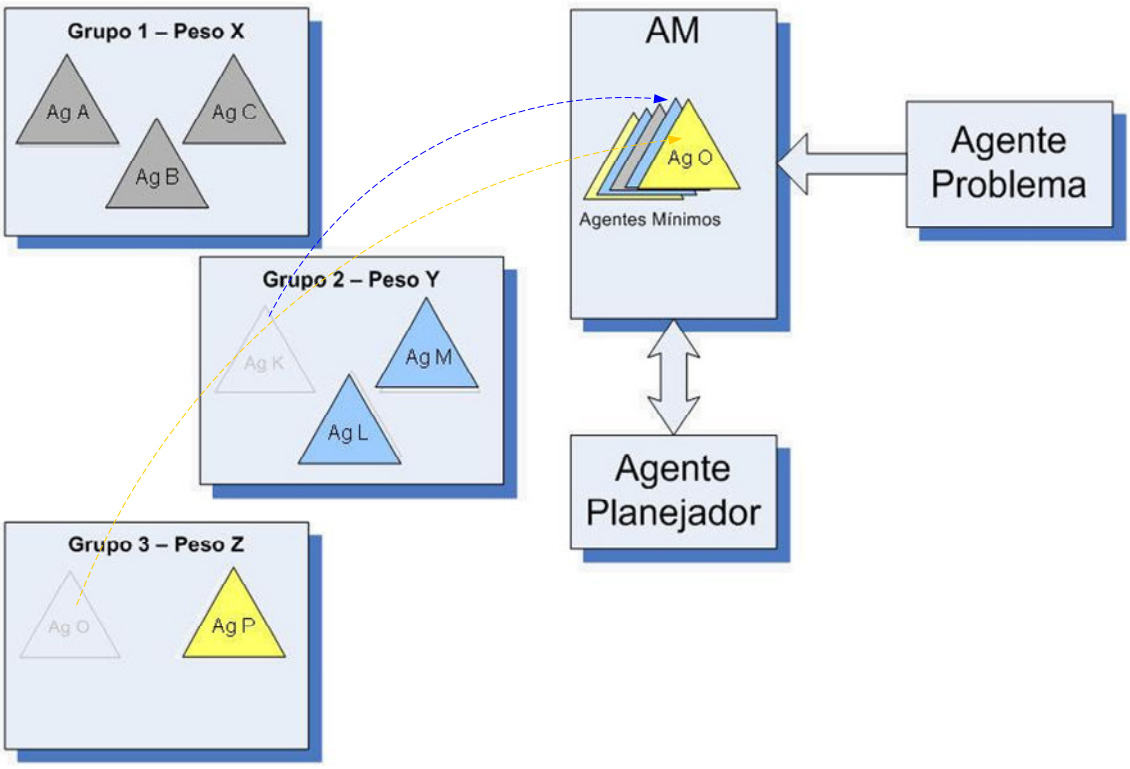

Figura 8 – Busca direcionada

A Figura 8 ilustra um estado específico de uma busca direcionada, onde, além dos agentes mínimos para se resolver uma tarefa trivial, foram adicionados ao conjunto de agentes o "Agente k" pertencente ao Grupo 2 e o "Agente O" pertencente ao Grupo 3.

O custo dos planejamentos consecutivos pode ser questionável, uma vez que para um domínio mais extenso, o tempo que seria gasto com estas tentativas poderia inviabilizar o sistema. Porém, é importante rever o funcionamento da extração da função heurística por parte do planejador FF. Durante a extração desta heurística [25], cujo processo é bem mais rápido do que a própria busca em si, é possível (na maioria das vezes) se determinar a não-possibilidade de se encontrar uma solução, conforme também mostrado no APÊNDICE A. No entanto, é óbvio que o processo interativo em bem mais lento do que um simples planejamento. Esta diferença de tempo foi medida e pode ser encontrada no APÊNDICE D.

O sistema proposto neste trabalho, o Agente Polimórfico Dinâmico Inteligente, funciona com qualquer planejador PDDL, feitas as devidas adaptações, porém os testes e resultados a serem apresentados neste trabalho estarão utilizando o planejador FF, principalmente pela sua velocidade de planejamento, qualidade do plano e pelos resultados constantes.

3.5. Ferramentas CASE Para PDDL e TREEVIEW PDDL

Atualmente são poucas as ferramentas CASE (*Computer Aided Software Engineering*) para o auxílio à elaboração de domínios e problemas em PDDL. A forma mais utilizada ainda é o editor de texto.

Em [29], é proposto um ambiente completo, chamado VitaPlan (Figura 9) para confecção de domínios de forma visual e integração com o planejador HAP do mesmo autor.

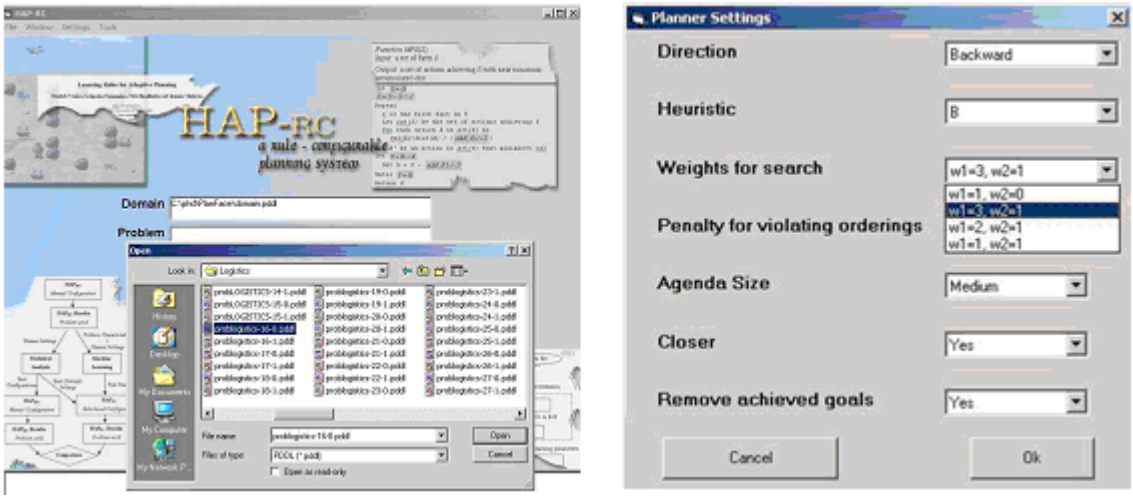

Figura 9 – Vitaplan: Seleção de domínios, problemas e configuração do planejador

Esta ferramenta visual permite ao usuário utilizar o sistema de planejamento e ajustar as regras e heurísticas necessárias. O VitaPlan contém ainda uma outra ferramenta visual, baseada no PDDL que permite que o usuário crie novos domínios

e problemas de planejamento de um modo gráfico (Figura 10) e permite, da mesma forma, extrair representações visuais de domínios e problemas de planejamento já existentes. A ferramenta contém ainda um módulo que simula a execução do plano e ilustra as mudanças no mundo em questão que segue a seqüência de ações do plano gerado.

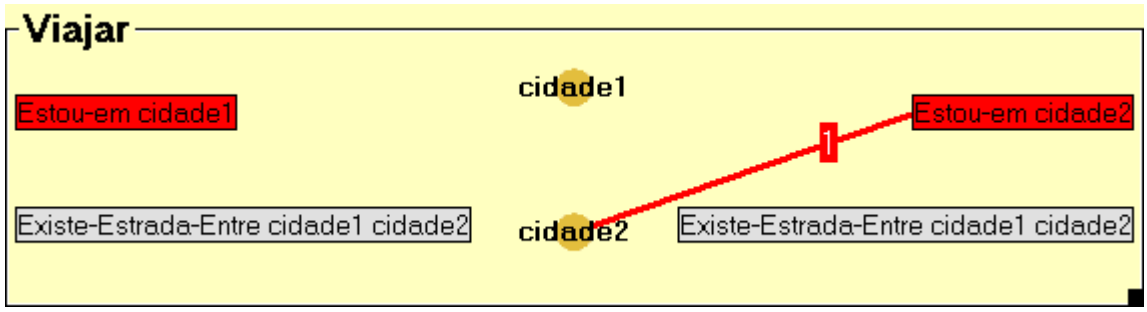

Figura 10 – Ação “Viajar” no editor Gráfico do Vitaplan

A Figura 10 mostra a representação do operador “Viajar” no editor gráfico de domínios da ferramenta VitaPlan. Do lado esquerdo está a lista de predicados da precondição da ação, no centro estão os parâmetros e à direita a lista de efeitos. O problema PDDL é representado de forma análoga ao operador. À esquerda ficariam os estados iniciais do problema, no centro os objetos e à direita os objetivos.

A iniciativa é interessante, porém o programa é bem limitado no que diz respeito à linguagem PDDL, sendo restrito a domínios muito simples e somente com o uso dos operadores “and” e “not”. Este último é inferido comparando-se a lista de predicados da esquerda com a lista de predicados da direita.

Outro fator que talvez desencoraje o uso de uma ferramenta como essa é que com a prática de criação de domínios e problemas, a preferência dos usuários ainda é o editor de texto. O editor de texto mais usado pra se modelar domínios PDDL é o EMACS [30], que na verdade é utilizado para a edição de vários tipos de linguagens e possui também um modo PDDL [31], que destaca as palavras reservadas da linguagem além de auxiliar a abertura e fechamento de parênteses.

Porém uma grande dificuldade de se modelar domínios surge quando uma lista de predicados ou efeitos começa a ficar um pouco extensa, com operadores dentro de operadores formando uma árvore muitas vezes difícil de se interpretar visualmente com apenas parênteses.

Tendo em vista esta dificuldade e juntamente com a necessidade de se trabalhar com domínios e problemas PDDL em forma de classes e objetos, um resultado secundário deste trabalho foi o software TreeView PDDL ilustrado a seguir.

O objetivo principal do TreeView PDDL é organizar a estrutura dos componentes da linguagem PDDL em classes e posteriormente serializá-las no formato XML *(Extensible Markup Language*) [32], e como objetivo secundário ser uma boa ferramenta CASE para PDDL. O software trabalha com os componentes da linguagem PDDL em forma de árvore, com o usuário podendo expandir e recolher os ramos da árvore. Além de auxiliar na modelagem lógica dos domínios e problemas, o software evita que erros comuns, principalmente digitação errônea, mas também erros como parâmetros que não se aplicam a determinados predicados (parâmetros de tipos incompatíveis) ocorram.

Como ilustrado na Figura 11, no quadro A, o usuário utiliza o mouse para expandir, adicionar ou remover itens do domínio. É importante ressaltar uma pequena diferença interpretativa que se pode ter à primeira vista no software: O conceito de “Lista de Precondições” e “Lista de Efeitos”.

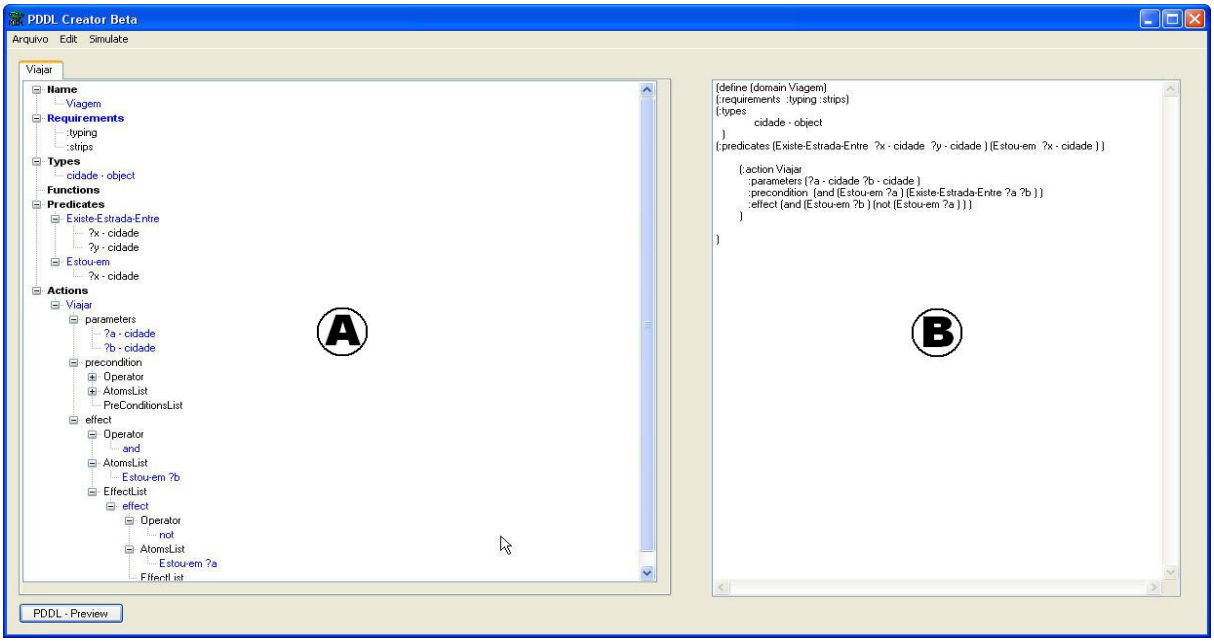

Figura 11 – Imagem da tela do software TreeView PDDL

Se a precondição "*precondition*" do domínio em PDDL precisa ser verdadeira para a realização da ação, e como precondição existem, por exemplo dois átomos com um operador "*and*", pode-se afirmar que o resultado da operação "*and*" entre os dois átomos em questão, é uma pré-condição para que a precondição "*precondition*" do domínio PDDL seja satisfeita. Se além dos dois átomos, o operador "*and*" tiver um terceiro elemento na operação isto continuaria sendo verdadeiro. Porém, se este terceiro elemento, ao invés de ser representado por um terceiro átomo, for representado, por exemplo, pelo resultado de uma operação "*or*" entre outros 5 átomos, o que interessa é o resultado da operação "*or*" que será verdadeira ou falsa, e assim recursivamente. Por esta razão, o conceito de "Lista de Precondições" foi adotado e da mesma forma a classe "*precondition*" escrita em linguagem C# [33] foi implementada.

De forma resumida, o "*precondition*" da ação em PDDL, pode ser considerado como o resultado *booleano* (Verdadeiro ou Falso) de uma operação entre átomos e uma "Lista de Precondições". E de forma análoga, uma "Lista de Precondições" é o resultado *booleano* de uma operação entre átomos e outra "Lista de Precondições",

transformando em uma árvore todas as operações que formam o "*precondition*" da ação.

Da mesma forma, este conceito foi aplicado também para o efeito da ação (*effect*) em PDDL e para o objetivo (goal) do problema em PDDL.

No quadro B do programa, mostrado na Figura 11, o usuário pode visualizar como está ficando o domínio ou problema em texto, já na sintaxe da linguagem PDDL.

Alguns domínios pelo tamanho da árvore de precondição que possuem, podem se tornar trabalhosos de se modelar e ajustar. Este software auxilia no desenvolvimento de domínios neste caso.

O software salva os domínios e problemas em dois formatos distintos:

- O formato ".pddl" que é o formato em texto puro, na sintaxe PDDL e que pode ser lido e interpretado por qualquer planejador que suporte PDDL.
- O formato ".spddl" ou "serialized PDDL", que é o formato PDDL estruturado em forma de classes e serializado em XML, e que é utilizado pelo Agente Polimórfico Dinâmico Inteligente Proposto neste trabalho.

Para a abertura dos arquivos de domínio e problema PDDL somente existe a opção do formato ".spddl", ou seja o programa só lê e edita os domínios e problemas criados por ele mesmo. O motivo desta limitação é que o "parser", ou seja o código que traduziria a linguagem textual em estrutura de classes não foi implementado. Do ponto de vista da usabilidade didática isso é uma limitação, porém, a elaboração e implementação do "parser" é demorada e estaria fugindo do foco e objetivo inicial que era serializar a estrutura dos componentes da linguagem PDDL.

Outra funcionalidade do programa é a integração com planejadores. Como se pode ver na Figura 12, na tela de configuração de simulação, o usuário indica o local

dos arquivos ".pddl" de domínio e problema, o caminho para o planejador que deseja utilizar e os parâmetro que deseja atribuir, uma vez que cada planejador tem seus parâmetros específicos.

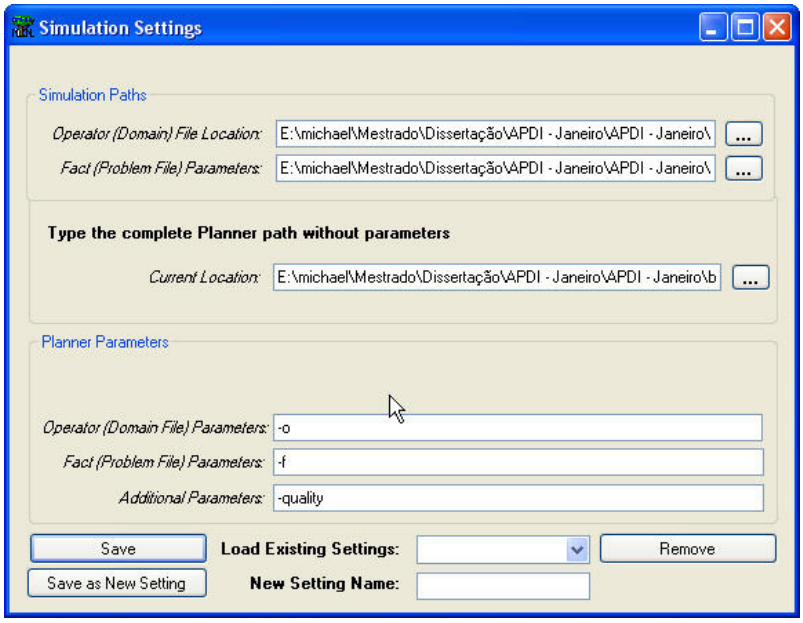

Figura 12 – Tela de Configuração de Simulação do TreeView PDDL

O usuário possui a opção de salvar suas configurações de simulação para posterior uso, facilitando os testes com variados domínios e planejadores sem ter que reconfigurar a simulação cada vez que desejar fazer um teste diferente.

Acessando o menu "Simulation", o usuário tem a opção de clicar em "Start Siumulation" onde o planejador configurado irá iniciar em thread DOS, separada do software TReeView PDDL e tentará achar um plano.

Como ferramenta CASE, o software é interessante para a didática em disciplinas relacionadas à planejamento em Inteligência Artificial. Com ele, o aprendizado ficaria inicialmente focado no entendimento da lógica e não tanto à sintaxe da linguagem. Para usuários e modeladores de PDDL avançados, o

programa não se mostra tão atraente, uma vez que a inércia de já se trabalhar com o ambiente textual é muito difícil de ser quebrada.

Porém, como dito anteriormente, o objetivo inicial do software que era serializar a estrutura dos componentes da linguagem PDDL foi cumprido e sua utilidade será melhor detalhada no Capitulo 6.

4. Metadados

4.1. Introdução

Para um melhor entendimento dos conceitos a serem apresentados, é necessário um perfeito entendimento de metadado e da forma que uma linguagem de alto nível realiza os procedimentos de execução de métodos e como isso é aplicado ao conceito de Programação Orientada a Objetos.

Somente alguns ambientes de programação baseados em framework de aplicação permitem a utilização de metadados. É o caso do "Framework .Net" da empresa Microsoft [34] e do Java Virtual Machine da empresa Sun [35].

O Microsoft .NET Framework, plataforma utilizada no desenvolvimento deste trabalho, é um componente que pode ser agregado ao sistema operacional. Foi projetado com três objetivos em mente. Primeiro, tornar aplicativos mais estáveis, enquanto prover uma aplicação com um grande grau de segurança. Segundo, simplificar o desenvolvimento de aplicativos para a Web e Web Services que não operam em equipamentos tradicionais, como em dispositivos móveis. Último, prover um conjunto de bibliotecas que podem ser usadas em múltiplas linguagens, diferente da plataforma Java que atualmente só trabalha com a linguagem Java.

O .NET Framework pode ser considerado um "sistema operacional" dentro de outro, ou seja, todas as requisições de programas gerados em sua plataforma são gerenciadas pelo próprio Framework. Quando pedidos de acesso à memória, requisição de hardware entre outras coisas são feitos, o Framework intermedia estes pedidos, fazendo com que, desta forma, somente ele próprio converse com o Sistema Operacional, sejam eles Windows, Linux, etc. Com este recurso, é possível realizar certas funcionalidades que eram mais complicadas de implementar anteriormente, e necessitavam do uso de ponteiros, rotinas em linguagem de máquina, etc.

O framework é composto de assemblies (DLL ou EXE) que dão suporte a qualquer linguagem que rode dentro dela, tornando seu desenvolvimento mais fácil. É uma biblioteca de classes que espelham as funcionalidades do sistema operacional, ou seja, acabando com as chamadas a API do Windows, tornando as tarefas muito mais simples, pois ao se fazer chamadas a API do Windows se garante a incompatibilidade com outros sistemas operacionais.

4.2. Assemblies

Assemblies são os arquivos (EXE, DLL) "executáveis" .NET, onde estão contidas as funcionalidades, informações detalhadas em três partes: número principal, secundário e de manutenção de forma permitir o uso da versão melhor compatível.

Eles são compilados para a forma de uma linguagem intermediária (CIL), esta forma é a forma padrão do .NET ler os arquivos e executá-los e será explanada a seguir.

4.3. CIL – Common Intermediate Language

O CIL, acrônimo para *Common Intermediate Language*, é uma linguagem intermediária utilizada para a comunicação do código fonte do programa com o processador. O CIL, antigamente chamado de MSIL (Microsoft Intermediate Language), é o nível mais baixo de linguagem humanamente legível, em toda a estrutura de linguagem do .NET framework, e é a linguagem resultante da compilação de código de todas as linguagens que suportam o .NET framework, como é caso de C#, Visual Basic .NET, C++, e J#.

Durante a compilação, o código-fonte é traduzido em código CIL, que é independente de plataforma, ou seja pode ser executado em qualquer ambiente que ofereça suporte ao .NET framework.

4.4. CLR - Common Language Runtime

Os Programas escritos para o .NET Framework são executados em um ambiente de execução, que faz parte do próprio .NET Framework, que é o CLR mencionado anteriormente. O CLR fornece a aparência de uma “máquina virtual”, ou seja, o programador não precisa considerar as capacidades da CPU que irá executar o programa. O CLR também fornece alguns mecanismos de segurança, gerenciamento de memória e tratamento de exceções.

Pode-se dizer que o (CLR) é o ambiente que gerencia a execução de código. Tradicionalmente ao se criar um aplicativo, o mesmo é compilado em linguagem que o computador possa entender (Linguagem de máquina) para posteriormente executá-lo. Em diferentes tipos de computadores (Ex.: PC e Machintosh), o usuário deverá recompilar o aplicativo para o mesmo se adaptar ao computador que vai utilizá-lo. Com o .NET Framework, isso funciona de forma diferente.

Quando um programa feito em VB.NET ou C# é traduzido para a linguagem intermediária (CIL), o Common Language Runtime transforma o código, agora em linguagem nativa do computador por um compilador Just-In-Time (JIT), otimizando o melhor possível para aquele processador. A Figura 13 ilustra a forma convencional de compilação, enquanto a Figura 14 ilustra a forma de compilação com CLR.

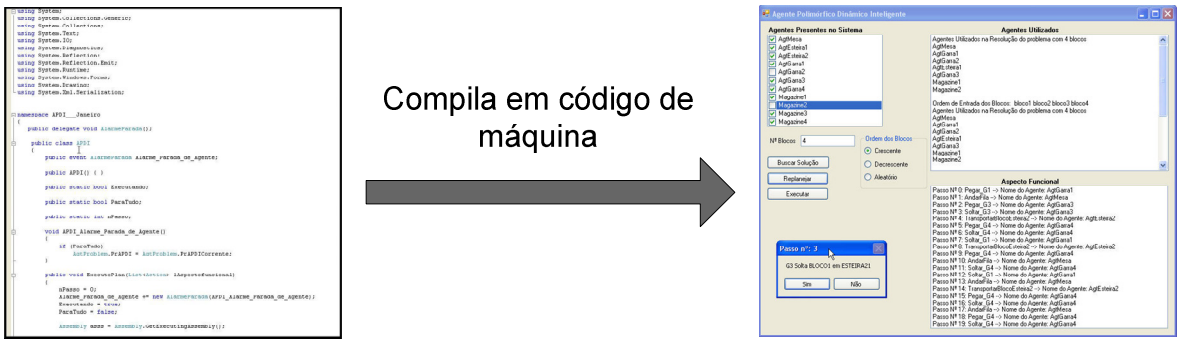

Figura 13 – Compilação Convencional

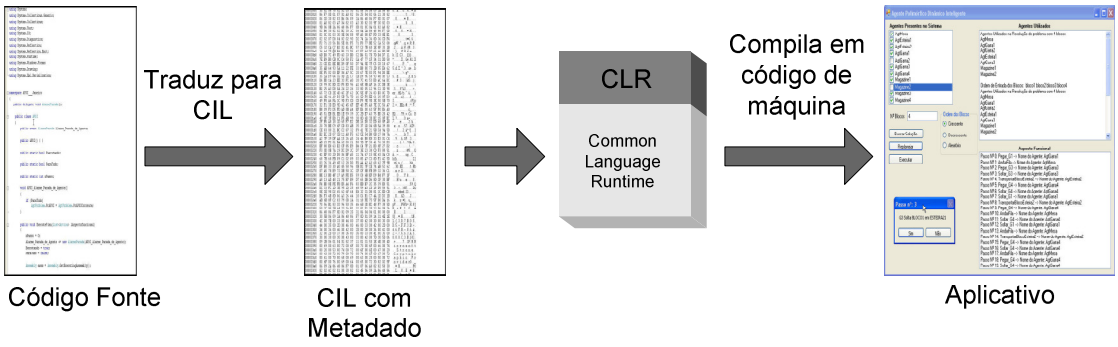

Figura 14 – Compilação com o CLR

4.5. Metadados

Quando se compila um código em C#, por exemplo, que foi a linguagem utilizada neste trabalho, eles são, portanto, compilados para a linguagem intermediária CIL, criando o que é chamado de Metadados.

Metadado é um conjunto de informações que descreve, com detalhes e através de uma linguagem neutra, o conteúdo de cada binário. Quando um módulo é executado, o seu metadado é carregado em memória e utilizado para obter informações sobre classes, membros, heranças, métodos, etc. Alguns dos dados armazenados pelo metadado;

- Descrição do *Assemblie(Descrito no item 4.2)*:
  - Identidade (nome, versão, cultura, chave pública);
  - Tipos de dados exportados;
  - Assemblies dependentes deste mesmo assemblie
  - Permissões de segurança necessárias para a execução
- Descrição dos tipos:
  - Nome, visibilidade, classe base e interfaces implementadas
  - Membros (métodos, campos, propriedades, eventos, etc...)

- Atributos
    - Elementos descritivos que modificam tipos e membros.

Como ressaltado anteriormente, não são todos os ambientes de programação que permitem a utilização destes metadados. Apenas os que apresentam uma característica de linguagem intermediária e framework de aplicações é que dão tal suporte. Sistemas desenvolvidos neste tipo de ambiente são compilados para uma linguagem intermediária (LI) gerando um código nativo. Este código nativo contém instruções de carregamento, armazenamento, inicialização, chamada a métodos presentes em objetos, acesso à memória e outras informações. A Figura 15 mostra um exemplo das informações contidas no metadado de um método qualquer.

Figura 15 – Exemplo de Metadado

Quando um compilador gera a LI, gera também seu metadado que será utilizado durante o tempo de execução. Ao se iniciar o sistema, toda esta informação é compilada em tempo-real gerando um binário que é armazenado em memória. Desta forma, quando uma chamada a uma função é realizada uma mensagem contendo seu metadado é enviada ao framework, que procura no conjunto de metadados armazenados no binário uma assinatura idêntica. Quando esta assinatura é encontrada, o método é executado. O mesmo procedimento ocorre na leitura e gravação de variáveis, propriedades, etc. A Figura 16 exemplifica o processo descrito

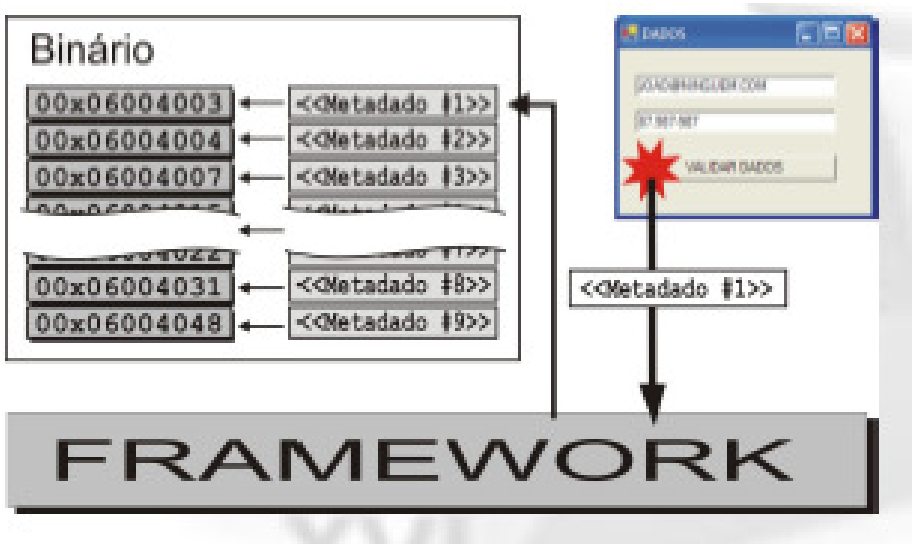

Figura 16 – Chamada a função utilizando um *framework*

Da mesma forma que uma mensagem é enviada a partir de uma chamada ordinária a uma função previamente implementada, pode-se enviar uma mensagem contendo o metadado desta função, sem que tal função tenha sido utilizada em algum ponto do código. Este é o principal conceito utilizado para implementar a programação a aspectos de forma dinâmica.

5. Programação Orientada a Aspecto

5.1. Introdução

A Programação Orientada a Objetos apareceu como a tecnologia que pôde fundamentalmente auxiliar a engenharia de software, devido à semelhança dos objetos modelados com os domínios reais. Porém, vários problemas de programação surgiram e que, para serem resolvidos, as técnicas de orientação a objetos não eram suficientes. Na verdade, alguns problemas de programação não podiam ser resolvidos nem por Orientação a Objetos nem por programação procedural. Dentro destas necessidades surgiram as técnicas de Programação Orientada a Aspectos.

A Programação Orientada a Aspectos, ou AOP (Aspect Oriented Programming) [4] surgiu da necessidade de diminuir o nível de acoplamento entre componentes ao se projetar sistemas de alta complexidade. Basicamente o AOP decompõe o domínio de um problema qualquer em dois níveis; classes e aspectos. Enquanto as classes, da Programação Orientada a Objetos representam a codificação das funcionalidades do sistema, desenvolvidas de forma nuclear e independente, os aspectos são responsáveis pela composição da solução. Esta composição está associada com a forma na qual o sistema irá operar, ou seja, diferentes formas de operação levam o aspecto a compor diferentes sistemas.

O AOP é um paradigma de programação que foi proposto com a intenção de lidar com problemas com alto índice de entrelaçamento entre componentes. AOP funciona decompondo o problema em partes para uma futura recomposição. O grande resultado deste paradigma está relacionado aos novos mecanismos de composição que diminui drasticamente o número de dependências entre os componentes. Em AOP, problemas são decompostos e modelados de acordo com o conhecimento do domínio. Algumas partes do modelo gerado se fundem com outras partes através de mecanismos de orientação a objetos (estes, são componentes

normais), mas algumas partes requerem mecanismos mais avançados (estes são chamados componentes *aspectuais* do problema ou simplesmente Aspectos).

A Figura 17 exemplifica o problema do entrelaçamento de código e a solução AOP, onde, em “A)” existe uma representação de uma estrutura de classes, onde existem algumas interdependências entre estas classes. Caso seja necessária uma atualização de código ou mudança qualquer, seria necessário realizar as mudanças em cada uma das classes. Como solucionar este problema? Em “B)” são identificados alguns componentes comuns a todas estas classes. Com isso, o problema pode ser decomposto em partes para uma futura recomposição. Estes elementos comuns estão representados pelos quadrados de cor vermelha, ou seja, foram identificados como um aspecto . Finalmente, em “C” pode-se notar que não existem mais ligações fortes entre as classes, e a única relação entre elas é o Aspecto que poderá, portanto, ter um desenvolvimento totalmente separado das estruturas de classe.

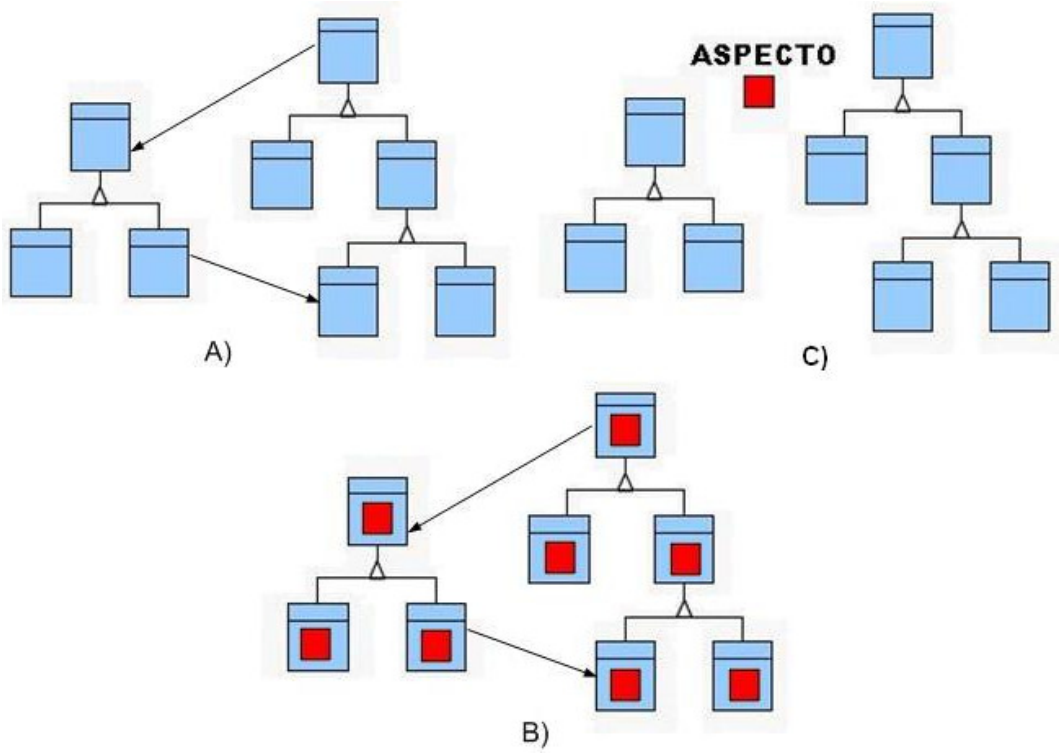

Figura 17 – Entrelaçamento de Código e a Solução AOP

Na fase de composição os agentes são reunidos por um processo chamado *weaving*, resultando no sistema final. O Processo de *weaving* pode ser implementado de duas diferentes maneiras: a) estaticamente, quando um conjunto de instruções indica ao compilador ou pré-processador como é a composição das funcionalidades dos componentes com as funcionalidades dos aspectos e b) dinamicamente, quando as funcionalidades dos componentes são juntadas, durante o tempo de execução, às funcionalidades de um dado aspecto.

O que diferencia um aspecto de um componente normal é a sua composição com o resto do sistema. Um componente regular representa um agente do sistema e tem acesso a suas funcionalidades internas. Um aspecto pega um conjunto de agentes presentes no modelo e cria conexões externas entre eles. Estas conexões entre o aspecto e o resto do sistema são definidas em um terceiro elemento, um *binder*. O *binder* usa meta-informações sobre os componentes, tais como suas variáveis e métodos, para compô-los. Desta forma as conexões por si próprias se tornam extremamente programáveis ao invés de serem definidas estaticamente dentro dos componentes.

A composição do sistema é feita através de um *weaver*, um elemento que pega as instruções de ligação entre os componentes e reúne os diferentes componentes. Neste trabalho, foi implementado um sistema com um weaver dinâmico com capacidade de *reflection*. *Reflection* é uma técnica suportada pelas linguagens de programação mais modernas, e que permite que um agente inspecione e manipule a si mesmo e/ou outros agentes. Usando a biblioteca *Reflection* o programa tem acesso aos metadados e podem fazer uso desta informação.

Uma propriedade que merece atenção nesta metodologia é que os componentes presentes no sistema são variáveis por si mesmos: eles não são conhecidos durante o tempo de programação. Usando estas (meta) variáveis ao invés de variáveis criadas durante tempo de projeto, a complicada lógica para mapear os componentes e suas funcionalidades é evitada. Outra vantagem desta

metodologia é que se algum componente novo for adicionado ao sistema, o *weaver* tem automaticamente acesso a seus metadados e ele é automaticamente adicionado ao sistema sem qualquer modificação.

Utilizando-se dos benefícios da metodologia AOP é possível desenvolver um sistema Multiagentes altamente flexível aplicado a um cenário de manufatura virtual. A idéia é desenvolver componentes representando cada agente (dispositivos de manufatura) independente do cenário no qual irão trabalhar. Cada componente tem métodos responsáveis por prover informações como suas ações, variáveis e predicados.

Para processar estas informações, uma entidade chamada Agente Monitor (AM) é responsável por procurar por todos os agentes disponíveis, capturar suas informações e repassá-las a um planejador. Após a criação de um plano de ação, o AM identifica os componentes que foram utilizados, e junta todos eles com um aspecto funcional gerando um agente final.

Juntamente com as vantagens desta metodologia, um grande problema é a padronização. Durante o tempo de execução o planejador precisa visualizar as funcionalidades do sistema para definir a estratégia de ação. Por isso, o AM deve fornecer uma lista de todas as funções disponíveis.

Utilizando-se AOP, o AM lê as meta informações contidas nos componentes e informa ao planejador. Além disso, o planejador pode gerar um plano de ações e retorná-lo ao AM para que este reúna todas as informações e possa montar o agente polimórfico. No entanto, o AM deve ser capaz de reconhecer quais variáveis, predicados e ações irão passar ao planejador, evitando informações desnecessárias. Este reconhecimento é feito através de assinaturas específicas, que podem estar presentes em vários aspectos, diferenciando partes de código que são exclusivamente utilizadas para representar o conhecimento do domínio (o que é de interesse), das outras partes que simplesmente ajudam na implementação.

Existem várias maneiras de implementar essas assinaturas. Neste trabalho, métodos e variáveis de interesse foram implementadas de acordo com interfaces específicas. Portanto, o AM lê as informações presentes nos componentes e passa ao planejador apenas aquelas provenientes destas interfaces específicas, as quais representam cada componente no formato PDDL

5.2. Aplicação de AOP no trabalho Proposto

Supondo um sistema com várias classes diferentes de Agentes, cada qual com seus respectivos métodos, propriedades e funcionalidades. Estes métodos por sua vez, possuem cada um, diferentes conjuntos de parâmetros de entrada e tipo de retorno.

Com o objetivo de se construir um Agente Polimórfico Dinâmico que incorpore vários agentes com diferentes funcionalidades, pelos métodos tradicionais de programação, seria preciso tratar cada Agente Regular separadamente. Seriam necessárias rotinas específicas para o acoplamento de cada um destes agentes. Caso novos agentes fossem incorporados ao sistema, novas rotinas seriam também necessárias. Além disto, soma-se o fato de que os métodos pertencentes a cada agente separadamente, possuem diferentes números de parâmetros, tipos de parâmetros e tipos de retorno. Fazendo-se uma simples soma do número de agentes e número de métodos que cada um possui, é possível visualizar a quantidade de código específico que teria que ser escrito, que seria uma rotina separada e dedicada para a execução de cada ação de cada agente.

O Agente Polimórfico Dinâmico não precisa conhecer todas as combinações possíveis de agentes existentes, para resolver determinado problema. Quando o AM determina os agentes presentes e os envia ao Agente Planejador, e este por sua vez retorna o plano, o APD, com o auxílio de uma lista chamada Aspecto Funcional gerada pelo AM, consegue acoplar o conjunto de agentes/ações necessárias para a resolução de qualquer problema dentro do domínio de aplicação. Isto só é possível,

se for utilizada a técnica de AOP, onde, no caso deste trabalho, o APDI é um aspecto.

Outra utilização de AOP neste trabalho, em uma abordagem diferente, é com relação aos agentes regulares. Padronizando a descrição das funcionalidades de cada agente no formato PDDL, é possível a inserção automática de um agente no sistema. Este agente deverá ser inserido como uma classe e com um determinado rótulo, e com suas funcionalidades descritas em PDDL, para então já estar pronto para ser utilizado no sistema.

Para exemplificar este processo como um todo de forma ilustrativa, a arquitetura do sistema representada na Figura 18 será utilizada como exemplo. Primeiramente serão considerados alguns agentes que são explicados no Capítulo 6, que são os Agentes Regular, Problema, Planejador, Monitor e APDI. Com a finalidade de ilustrar a utilização de AOP neste trabalho, as funcionalidades específicas de cada um não precisam ser apresentadas ainda. Nota-se que os três primeiros agentes (Regular, Problema e Planejador) são implementados segundo as interfaces *iAgent*, *iProb* e *iPlan* respectivamente. Estas interfaces específicas estão representadas pelas formas: Triângulo, pentágono e hexágono respectivamente. Isto quer dizer, por exemplo, que para se inserir um novo Agente Regular, basta que este esteja implementado segundo a interface *iAgent*. Outro exemplo, desta vez de algo já mencionado nos capítulos anteriores, seria o exemplo do Agente Planejador. Este agente utiliza um planejador para realizar algumas de suas tarefas, onde este planejador pode ser qualquer planejador existente, desde que esteja implementado segundo a interface *iPlan. C*omo exemplo, ele precisa utilizar a linguagem PDDL como linguagem padrão de entrada, precisa ter as suas especificações de parâmetros, etc. Em outras palavras, cada um destes agentes mencionados é um aspecto, podendo ser desenvolvidos separadamente um do outro e sem qualquer ligação com o restante do código do sistema muito menos entre eles mesmos.

Tendo os aspectos definidos, o próximo passo é como reuni-los e como fazer as ligações entre eles. Como dito anteriormente, isto é feito através de um *Weaver*.

No caso deste trabalho, este processo de *weaving* é realizado pelo Agente Monitor. Como é possível notar na Figura 18, o *weaver* é um sistema computacional pré-moldado para receber determinados tipos de aspectos. Isto é representado simbolicamente na figura pelas lacunas nos formatos específicos das interfaces mencionadas anteriormente. É importante notar também, que a profundidade destas lacunas determina a quantidade de aspectos possíveis de serem alocados, ou seja, podem ser encaixados: um planejador, um problema e vários agentes regulares.

O *Binder*, presente também no Agente monitor, é então responsável por fazer as ligações entre estes componentes. A partir do momento que estas ligações estão prontas, com cada componente devidamente alocado, está pronto meu sistema planejador que será responsável por gerar o Aspecto Funcional. O Aspecto Funcional, de forma simplificada é uma lista de ações a serem realizadas, juntamente com a indicação de qual agente é responsável por que ação. Em um outro nível, o APDI também é um *weaver*, onde vão ser encaixados os Agentes Regulares selecionados, juntamente com o Aspecto Funcional para que o plano possa ser executado.

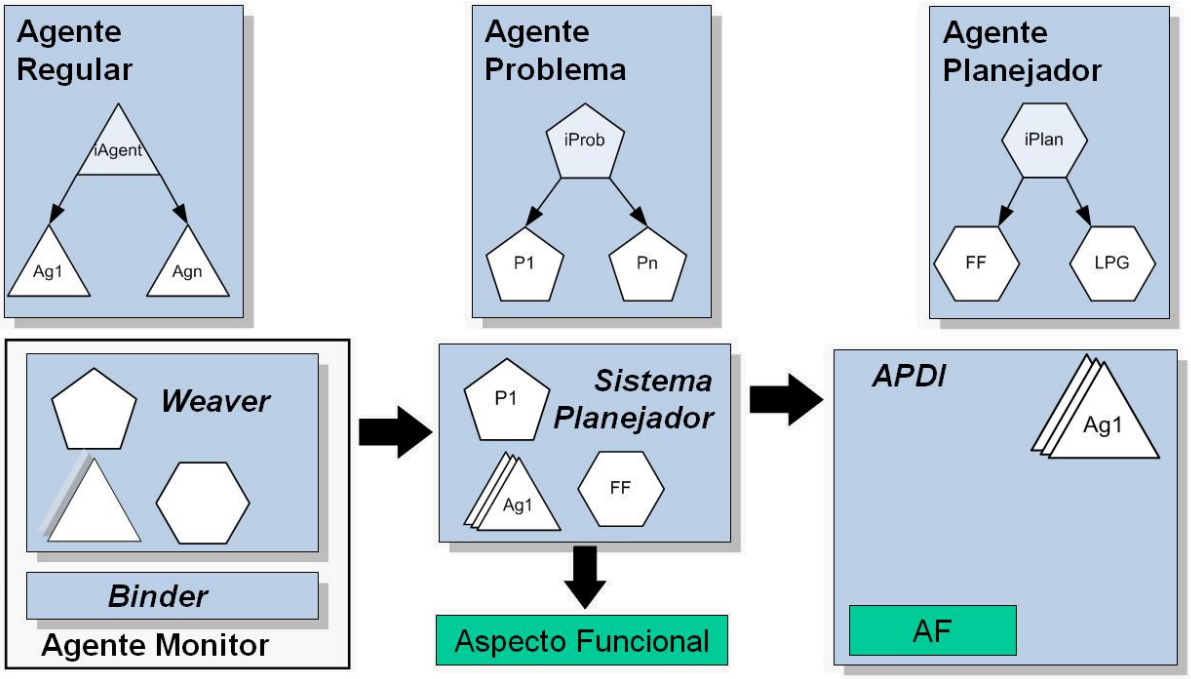

Figura 18 – Usando Modelagem Orientada a Aspecto

Portanto, não é preciso saber as características e funcionalidades dos agentes regulares, nem quais os problemas possíveis existentes, muito menos qual planejador será utilizado. Desde que modelados de acordo com as interfaces específicas, tudo está pronto para ser inserido no contexto do sistema.

5.3. Porque Utilizar AOP?

Para demonstrar as vantagens de se utilizar AOP neste trabalho será usado como exemplo os agentes regulares, ilustrados na Figura 19. Supondo que existam dois agentes distintos, um agente robótico (Rob_1) e um sistema de visão (Vis_Sys), ambos implementados segundo a interface *iAgent*. Note que cada um dos agentes internamente é um sistema orientado a objeto com várias classes interdependentes entre si, porém externamente não possuem nenhuma ligação.

No método convencional orientado a objeto, para um determinado problema "P1", seria preciso criar um solucionador dedicado, "Solver_P1" por exemplo. Este solucionador seria específico para aquele problema e teria as instruções para a solução do problema "P1". Executaria uma funcionalidade do robô, depois outra do sistema de visão, outra do robô e assim por diante, de acordo com o plano de ações. Porém para um segundo problema "P2", seria necessária a criação de um segundo solucionador "Solver_P2" e assim seria para cada problema diferente.

Usando a orientação a aspecto isto não acontece, pois este solucionador é criado somente de acordo com a necessidade. Não é preciso conhecer todos os problemas possíveis, muito menos quais os agentes existentes. Desde que modelados de acordo com as devidas interfaces, o sistema identifica estes aspectos e monta o sistema planejador de acordo com a necessidade do problema.

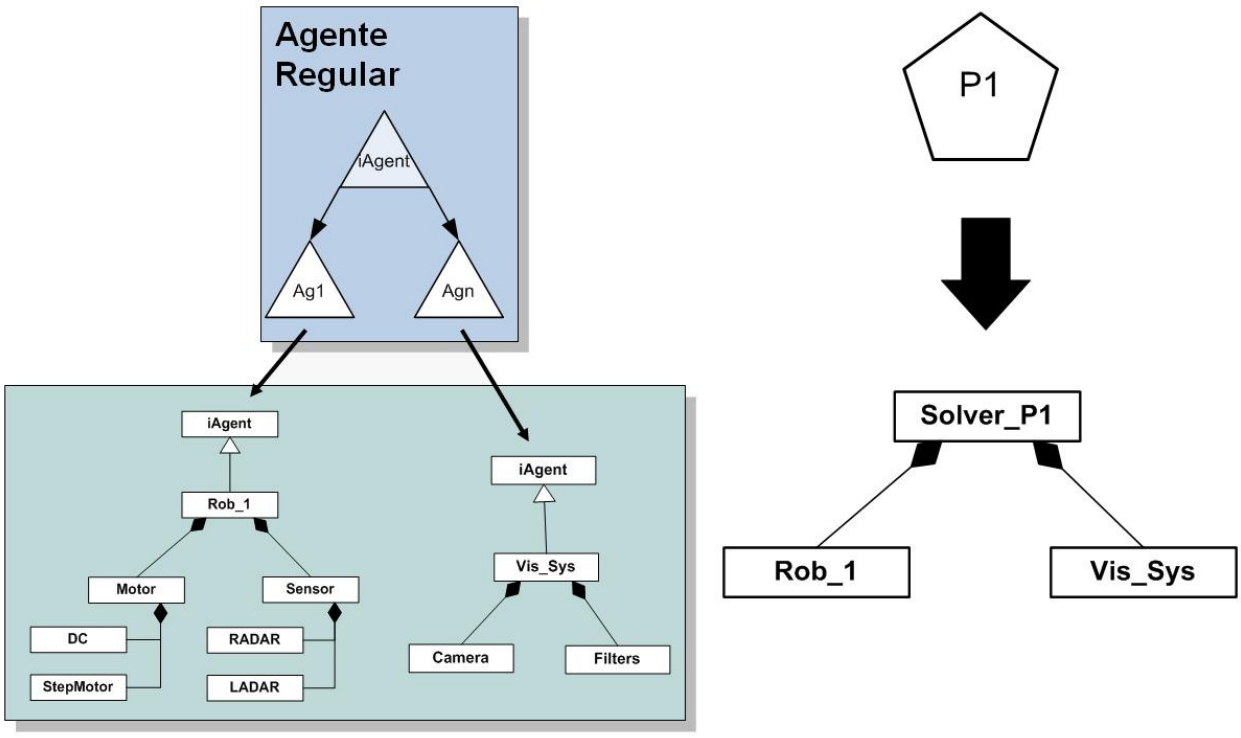

Figura 19 – Resolvedor Dedicado

Um outro exemplo de vantagem da utilização de AOP neste trabalho é mostrado na Figura 20, onde tem-se o APDI já montado, com os agentes incorporados e executando as tarefas de modo a atingir o estado objetivo. Se, durante a execução das tarefas, ocorrer algum problema, o sistema é capaz de, automaticamente e durante o tempo de execução, elaborar um novo problema com novos estados iniciais, buscar um novo conjunto de agentes para realizar a tarefa anteriormente interrompida.

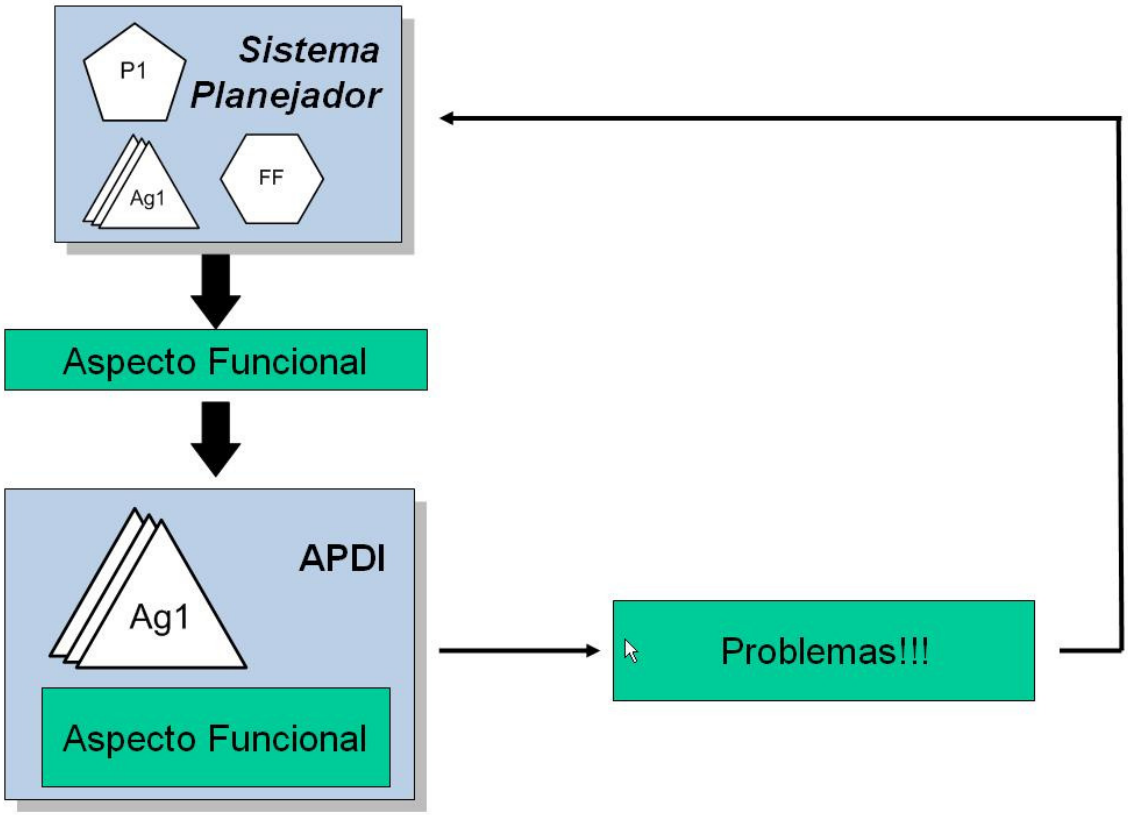

Figura 20 – Replanejamento on-line

5.4. Utilização de Reflection no trabalho proposto

Reflection é uma técnica suportada pelas linguagens de programação mais modernas, e que utilizam um *framework*. A biblioteca *Reflection* permite que o programa tenha acesso aos *metadados* e possam fazer uso desta informação.

A utilização de AOP neste trabalho foi possível graças às funcionalidades fornecidas pela biblioteca *Reflection*. Tendo que um dos objetivos do APD é utilizar-se do polimorfismo para realizar ações através dos agentes necessários, pelas técnicas tradicionais de programação, seria necessário que cada método de cada agente fosse projetado para ser instanciado pelo APDI devido à diferença de parâmetros (número e tipos) e retorno. Existiria ainda um problema de que, se um agente fosse retirado ou adicionado ao sistema, deveriam ser adicionadas ou retiradas linhas de código para adaptar essas mudanças.

Através da adição de rótulos às Classes e Métodos, durante o tempo de projeto, pode-se, através dos metadados, identificar quais classes são Agentes ou não, e ainda identificar as características de cada método dos Agentes.

Observando o exemplo de código abaixo pode-se ter uma melhor idéia da utilidade de se rotular métodos e classes.

```
public class AgtEsteira1
{
    public string Classe = "Transporte";

    public bool TransportarBlocoEsteira1(parameter bloco, parameter PosInicial, parameter PosFinal)

    public Domain OQueFaz_AgtEsteira1()...
```

Exemplo 3 – Classe do Agente Esteira

Se durante o tempo de projeto, é informado ao programa que toda classe da qual o nome começa com “Agt”, é um agente, pode ser realizada uma busca no *assemblie* do programa para filtrar classes deste tipo. Desta forma, se for acrescentado ou removido algum agente, não é preciso fazer nenhuma modificação no código do programa, pois eles vão sendo automaticamente retirados ou adicionados.

No exemplo mostrado o método “OQueFaz_AgtEsteira” é um outro exemplo de rótulo, utilizado prevendo-se a utilização de Orientação a Aspectos. Com todos os agentes contendo um método cujo nome se inicia com “OQueFaz_”, quando for preciso a inserção de um agente com novas funcionalidades, este método é o responsável por prover todas as informações necessárias, como precondições e efeitos, evitando assim, mais uma vez, a reengenharia do programa. Este conceito, pela semelhança com o metadado, porém com um nível de abstração superior, foi definido neste trabalho como “Hiperdado”, e será melhor aprofundado no capítulo 6, item 6.3.

O Exemplo 4 mostra uma outra classe que tem características diferentes das apresentados no Exemplo 3, porém não existindo diferença alguma na hora de instanciá-la.

```
public class AgtGarra1
{
    public string Classe = "Manipulacao";

    public bool Soltar_G1(parameter Bloco, parameter localizacao_bloco)...

    public bool Pegar_G1(parameter Bloco, parameter localizacao_bloco)...


    public Domain OQueFaz_AgtGarra1()...
```

Exemplo 4 – Classe do Agente Garra 1

É importante ressaltar que esta funcionalidade foi analisada de modo estático, ou seja, implementada durante o tempo de projeto. Porém, através dos metadados, é possível tirar proveito da mesma técnica, porém desta vez de modo dinâmico, ou seja, durante o tempo de execução do programa.

De forma dinâmica, esta técnica é utilizada para a execução de ações, uma vez que um agente pode ter um ou mais métodos funcionais, mas se este fato for enxergado como um aspecto, durante a leitura e interpretação de qualquer agente, este pode conter qualquer quantidades de métodos e propriedades. Da mesma forma, o método pode ter quaisquer tipos de argumentos, parâmetros ou retorno. Isto funciona da seguinte forma: Quando o APDI possui em mãos a lista de ações a serem executadas, com seus respectivos agentes (Lista com o Aspecto Funcional), seu objetivo é executar cada uma das ações. Para fazer isso, primeiramente ele procura qual o agente (Classe) é dono de tal ação. Após encontrado, é possível instanciar a classe deste agente, criando assim um objeto do mesmo. Com este objeto, através ainda dos metadados, o APDI procura um método deste objeto que possua o mesmo rótulo do método contido no aspecto funcional. Quando encontrado, este método é executado com todos os parâmetros necessários. O Exemplo 5 mostra a rotina utilizada no software APDI de forma simplificada e comentada.

```
[...]
Assembly ass = Assembly.GetExecutingAssembly();
List<Type> tipos = new List<Type>();
tipos.AddRange(ass.GetTypes());//Copia todos os tipos existentes no
assemblie
LimpaListaDeTipos(ref tipos);//Separa somente o que interessa
for (int i = 0; i < AgtMonitor.AFAcoes.Count;i++ ) //Varre todas as ações
(métodos) da Lista de Aspecto Funcional
{
      for (int j = 0; j < tipos.Count; j++)//Varre todos os métodos
      {
            Type t = tipos[j].UnderlyingSystemType;//Intificação da classe
do agente dono do método
            if (t.Name.ToLower() ==
      AgtMonitor.AFAgentes[i].Name_Domain.ToLower())
            {
                  object obj;
                  MethodInfo[] M = t.GetMethods();//Todos os métodos do
Agente selecionado são guardados no vetor M
                  for (int k = 0; k < M.Length; k++)
                  {
                        if (AgtMonitor.AFAcoes[i].Name_Action.ToLower() ==
M[k].Name.ToLower())//Se o método tá no aspecto funcional
                        {
                        ConstructorInfo cinfo =
                                          t.GetConstructor(new Type[] { });
                         obj = cinfo.Invoke(null);//Crio uma instância do
objeto do Agente em questão
                         M[k].Invoke(obj,
AgtMonitor.AFAcoes[i].parameters.ToArray()); //invoco a o método relativo à
ação necessária, e envio um vetor de parametros correspondente à ação
                   }
[...]
```

Exemplo 5 – Código simplificado da execução de ações utilizando *reflection*

Com isso, não é preciso instanciar todos os objetos que têm a possibilidade de serem utilizados. Á medida do necessário, pode-se consultar os metadados da classe, e retirar informações sobre os “construtores” da classe, e, enxergando um construtor como sendo um método, pode ser criado uma instância, ou objeto da classe em questão. Tendo um objeto instanciado desta classe, é possível invocar quaisquer que sejam os métodos pertencentes a esta mesma classe. Isto quer dizer que durante a execução do programa, dinamicamente, pode-se utilizar do polimorfismo para a execução de métodos, pois todos os métodos são executados por um objeto da classe APDI. Na realidade, é deste fato que se extraem as características “Dinâmico” e “Polimórfico” do Agente Inteligente proposto neste trabalho.

6. Agentes Polimórficos Dinâmicos Inteligentes

6.1. Introdução

Os conceitos apresentados nos capítulos anteriores tiveram o intuito de revisar toda a bagagem teórica requerida para o perfeito entendimento do conceito de Agentes Polimórficos Dinâmicos Inteligentes. Porém, é necessário ainda uma introdução à arquitetura do sistema que apresenta 7 elementos básicos :

- Agente Monitor;
- Hiperdado;
- Agente Planejador;
- Agente Problema;
- Aspecto Funcional;
- Agente Regular (componentes do sistema ou Dispositivos de Manufatura);
- Agente Polimórfico Dinâmico Inteligente

A importância e funcionalidade de cada um são explicadas e demonstradas a seguir de forma detalhada.

6.2. Agente Monitor

Da mesma forma que se pode executar uma função através do envio de uma mensagem contendo seu respectivo *metadado*, pode-se, após reconhecer os binários ativos em um sistema, importar todos os *metadados* de cada agente existente.

Com isso, e utilizando-se desta técnica, é possível ter um agente dedicado a monitorar o sistema, verificando quando outros agentes entram no mesmo. Esta tarefa define um conceito de agente monitor (AM).

Desta forma, em uma estrutura de Multiagentes não homogêneos, um AM pode centralizar o conhecimento do que cada agente tem a capacidade de realizar e, a cada nova entrada ou saída de agentes, reestruturar o sistema com o conhecimento atualizado.

Porém, conhecer os *metadados* contidos nos agentes presentes no sistema e centralizar este conhecimento em um único agente não implica que a finalidade de cada funcionalidade seja reconhecida, o que leva a um caso típico onde os dados não geram informações. Para transformar estes dados em informações um artifício semelhante ao *metadado* é utilizado. Este novo artifício criado recebe o nome de *Hiperdado*.

6.3. Hiperdados

Da mesma forma que os métodos têm *metadados* para a sua descrição, é necessário desenvolver outro conjunto de dados, denominados por este trabalho de *Hiperdados*. Logo, para cada agente desenvolvido para esta aplicação, deve-se desenvolver um h*iperdado* contendo todas as informações necessárias e pertinentes para auxiliar o AM na escolha dos Agentes necessários. Informações sobre descrição, significado de resposta, sobrecarga, parâmetros passados, significado destes parâmetros, precondições e efeitos, estão contidas no *hiperdado*, porém tudo em coerência com as necessidades do *Agente Planejador*. O formato e padronização destas informações serão discutidas quando for apresentado o *Agente Planejador.*

Com este conjunto de *hiperdados* o AM fica conhecendo todas as funcionalidades associadas aos agentes presentes no sistema com os respectivos

pré-requisitos necessários para sua execução bem como os efeitos obtidos com a execução da ação.

Porém, ainda dispondo de todo este conhecimento, o AM não tem capacidade para solucionar um problema por mais simples que seja. Tomando como objetivo ter uma entidade que seja capaz de, dado um problema, determinar como solucioná-lo e como implementar esta solução, é necessário definir duas novas entidades; Agente Problema e Agente Planejador.

6.4. Agente Problema

O agente problema é responsável por apresentar ao AM qual o estado final desejado do sistema. Isto também é feito através de hiperdados. O planejador pode ser uma entidade externa ao sistema onde, neste caso, deve-se ter um canal de comunicação com o AM ou interna como uma funcionalidade do próprio AM. Este trabalho utiliza um planejador externo.

A solução adotada por este trabalho possibilita utilizar qualquer planejador externo. Isto possibilita aproveitar toda a funcionalidade da linguagem PDDL, e a eficiência e "know-how" dos planejadores existentes. Portanto, qualquer domínio ou conjunto de domínios modelados em PDDL, já estará pronto para ser inserido no contexto dos agentes pertencentes ao sistema.

O agente problema tem ainda uma função de guardar o estado corrente do sistema. À medida que o plano de ações é executado, vão sendo criados estados correntes correspondentes a cada passo da ação. A razão disto está no fato de que, se a execução do plano, por algum motivo, tiver que ser interrompida, um novo planejamento poderá ser executado tendo como estado inicial o estado corrente no momento da paralisação.

Se durante a paralisação, ocorrer a inclusão ou exclusão de agentes ou recursos, o Agente Problema têm a função de adaptar o estado inicial incluindo e/ou

excluindo características pertencentes a estes Agentes ou recursos que foram incluídos e/ou excluídos.

6.5. Agente Planejador

O AM através dos Hiperdados dos Agentes consegue adquirir as informações de funcionalidade de todos os agentes e recursos, no formato PDDL, e através do agente problema, consegue as informações sobre o estado atual do ambiente também no formato PDDL.

O Agente Monitor, com este conhecimento necessário para realizar uma tentativa de se determinar uma forma de solucionar o problema, se comunica com o agente Planejador para que este gere um plano de ações para cumprir um objetivo. Caso seja possível encontrar uma solução, tal plano de ações contém a descrição de todas as funcionalidades necessárias para sua execução.

O Agente Planejador neste trabalho utiliza o planejador FF [20], já discutido anteriormente, para realizar o plano de ações. O FF retorna o plano gerado na forma de *Stream* (um mecanismo que permite a leitura e/ou escrita de dados de forma seqüencial em uma Coleção, String ou Arquivo), de onde as informações serão filtradas e enviadas como uma lista limpa de ações ao AM, contendo apenas o número dos passos, nome da ação e parâmetros utilizados.

A geração deste plano é um processo interativo entre o AM e o Planejador. Como explicado anteriormente no capítulo 3, apresentar muitas funcionalidades para um planejador pode gerar resultados insatisfatórios, dependendo do planejador, além do problema da escolha de ações, onde o planejador não prioriza a utilização de menos recursos. Desta forma, foi adotada a forma iterativa aonde, através de uma busca direcionada, o AM vai apresentando combinações crescentes de agentes para o Planejador até que uma solução seja encontrada. Quando isto acontece é montado o *Aspecto Funcional.*

6.6. Aspecto Funcional

O *Aspecto Funcional* é a união da solução encontrada, ou seja, o plano de ações, juntamente com a descrição de todas as funcionalidades, que por sua vez estão relacionadas com os respectivos agentes que as contém, e que são necessários para sua execução.

Com as funcionalidades e seus agentes identificados o AM, através de um processo de *weaving*, combina estes agentes com o aspecto funcional, gerando uma nova entidade denominada Agente Polimórfico Dinâmico Inteligente (APDI).

6.7. Agente Polimórfico Dinâmico Inteligente (APDI)

Um APDI pode ser considerado como uma simbiose entre agentes, onde elementos com características diferentes podem se unir e, como se fossem uma única entidade, realizar funções que nenhuma das entidades básicas seria capaz. O APDI é um objeto instanciado que através de *reflection* utiliza polimorfismo para a execução dos métodos pertencentes aos demais agentes que são, na verdade, classes diferentes.

Este novo agente, com inteligência própria e exclusivamente dedicada à resolução de uma tarefa específica, deve se desagregar do sistema de agentes até que a tarefa seja realizada.

O gráfico da Figura 21 ilustra os conceitos acima descritos onde:

- Em 1 se tem a representação do hiperdado, representado por um conjunto de pré-condições, uma funcionalidade e um conjunto de pós-condições ou efeitos;

- Em 2, o agente planejador recebe os hiperdados dos agentes 1 e 2 e, dado um problema, realiza um planejamento identificando quais as funcionalidades necessárias para solucioná-lo e gera um aspecto funcional;
- Em 3 o aspecto funcional se agrega aos agentes 1 e 2 gerando um novo agente, o APDI, uma entidade especializada em resolver os desafios apresentados pelo agente problema.

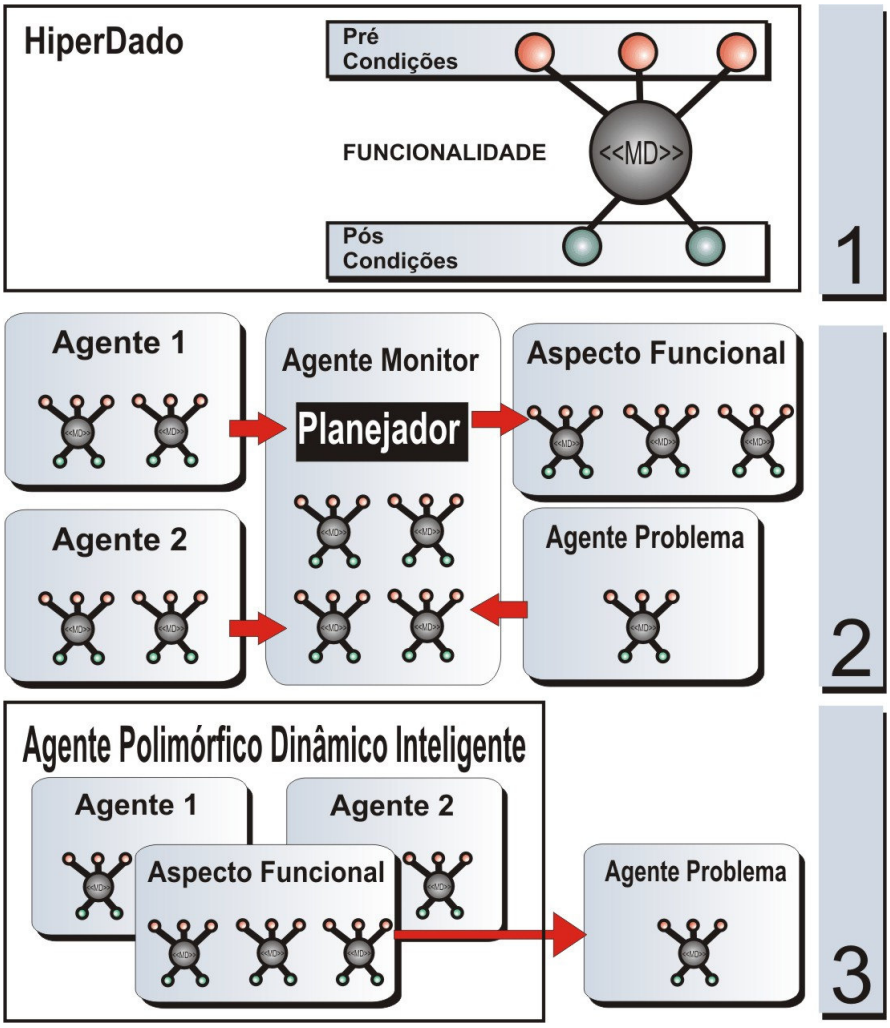

Figura 21 – Formação do APDI

Após sua formação, o APDI consegue executar as ações necessárias através dos agentes incorporados. Uma vez que nenhum dos agentes (que são na verdade classes de objetos) foi instanciado anteriormente, este polimorfismo é possível através das ferramentas dos metadados e utilizando aspecto. Com a metodologia de *reflection* é possível buscar, com a ajuda do aspecto funcional, os métodos equivalentes a cada ação do plano, conforme explicado detalhadamente no Exemplo 5, na página 59 e executá-los de forma clara e com todos os parâmetros necessários.

É importante ressaltar que os agentes podem estar fisicamente separados que a metodologia não se altera. O *Hiperdado* contém ainda o endereço do *host* do agente, permitindo que o APDI execute as ações via RPC (Remote Procedure Call)[36].

Portanto, de forma simplificada, o objetivo da arquitetura de APDI é realizar um plano de ações, em um ambiente Multiagente não homogêneo, para a resolução de um determinado problema dentro do domínio de aplicação. Os agentes reativos que compõem este sistema Multiagente, ou seja, a modelagem dos dispositivos de manufatura, são chamados neste trabalho de Agentes Regulares.

6.8. Agente Regular

Um Agente Regular pode ser qualquer agente que possui a capacidade de realizar uma tarefa. Neste trabalho os Agentes Regulares são alguns dispositivos de manufatura que tiveram suas funcionalidades modeladas em linguagem PDDL. A escolha destes agentes é totalmente dependente da escolha do domínio de aplicação a ser discutida no capítulo seguinte.

Os Agentes Regulares utilizados neste trabalho foram:

- 4 Garras Manipuladoras;
- 2 Esteiras Transportadoras

- 2 Mesas (Entrada e Saída)

É importante ressaltar que estes agentes podem ter alguns *recursos* que podem ser utilizados por eles mesmos ou por todo o domínio de trabalho, mas que também deverão ser modelados em linguagem PDDL. Estes recursos no domínio de aplicação são os *Magazines Armazenadores* que são dispositivos de armazenamento que podem ser utilizados a medida do necessário e dependendo de sua disponibilidade.

Os Agentes serão mais bem explanados no APÊNDICE B, onde se encontram suas descrições no formato PDDL e o significado de cada ação.

7. Aplicação e Resultados

7.1. Ambiente de Aplicação

Para validar a metodologia, um cenário de manufatura foi utilizado para simulação. A ilustração do ambiente proposto é mostrada na Figura 22.

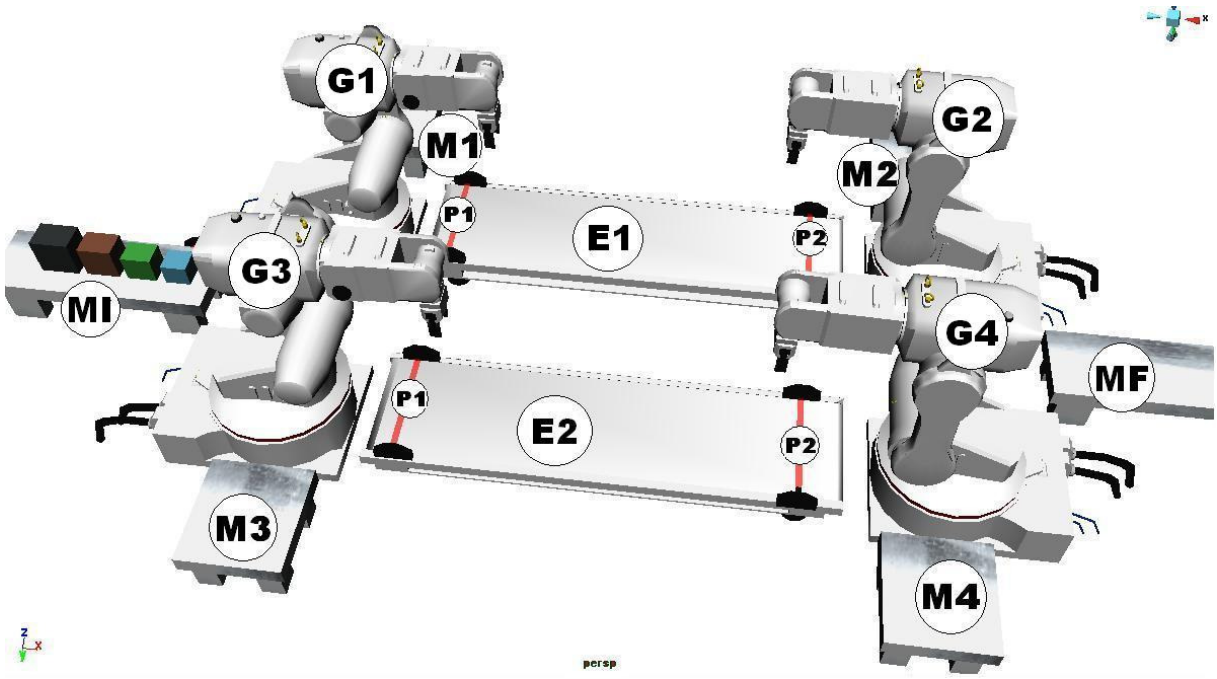

Figura 22 – Ambiente de Aplicação

Onde existem:

- 2 esteiras transportadoras (E1, E2) com posições (P1 e P2);
- 1 mesa de entrada de blocos (Mesa Inicial - MI) e 1 mesa de saída de blocos (Mesa Final - MF);

- 4 robôs manipuladores, ou garras manipuladoras(G1, G2, G3, G4);
- 4 magazines de armazenamento (M1, M2, M3, M4), cada um dedicado à um dos robôs,

O problema proposto é, dada uma ordenação específica dos blocos de entrada, encontrar uma forma de colocá-los na seqüência correta na mesa de saída, respeitando algumas restrições como:

- Da mesa inicial os blocos só podem ser retirados e nunca colocados por qualquer uma das garras. O mesmo acontece para o lado direito de cada uma das esteiras transportadoras, ou seja, nas posições P2 de cada uma delas;
- De forma análoga, da mesa final, os blocos não podem ser retirados, e sim apenas colocados, seguindo a mesma regra para o lado esquerdo das esteiras transportadoras, ou seja, nas posições P1 de cada esteira;
- Nos magazines podem ser alocados ou retirados blocos;
- As esteiras transportadoras só conseguem transportar um bloco por vez e possuem sensores em suas extremidades para indicar o momento de parada e para impedir que blocos caiam.
- Os robôs têm acesso a ambas as esteiras;

Neste cenário, como foi mostrado anteriormente, os robôs, as esteiras e as mesas são considerados os Agentes Regulares, cada qual com sua modelagem e domínio no formato PDDL, podendo ser ativados, desativados, incluídos ou excluídos do sistema. Já os Magazines Armazenadores são considerados *recursos* podendo estar disponíveis ou não.

O problema proposto é, além da ordenação dos blocos, identificar os agentes presentes no sistema e conseguir atingir o estado final desejado, com o mínimo de agentes possível e/ou diminuindo uma alocação desnecessária de recursos

A solução de um problema segue os seguintes procedimentos;

1. O Agente Monitor identifica os hiperdados dos agentes presentes no sistema e verifica quais dos agentes estão disponíveis para a realização de suas tarefas.

2. O AM monta um mínimo de agentes necessários para realizar uma tarefa simples (2 Garras, Mesas e 1 Esteira). À medida que a solução não é encontrada (após o final deste procedimento), vão sendo acrescentados um a um o restante dos agentes disponíveis;

3. Se não houve acréscimo no número de agentes mínimos após o último *loop(tentativa)* deste procedimento, significa que o sistema não pôde encontrar uma solução. Caso contrário, o Agente Problema instancia os estados iniciais e objetivos desejados, com base nos Agentes Mínimos corrente;

4. O Agente Planejador monta o domínio completo do sistema com base também nos Agentes Mínimos corrente e, juntamente com o problema montado pelo Agente Problema, faz um plano de ações e identifica os agentes necessários para sua execução.

5. Se a solução não for encontrada, repete-se o passo 1. Caso contrário, o Agente Monitor gera o Aspecto Funcional que é um algoritmo representando os metadados das funcionalidades a serem executadas pelos agentes envolvidos.

6. O aspecto funcional e os agentes envolvidos se separam do sistema de agentes para criar um novo APDI para resolver o problema proposto. Este APDI começa então a execução das tarefas.

7. Se, por algum motivo, durante a execução das tarefas, existir uma interrupção da mesma, o Agente Monitor guarda o estado corrente do ambiente, que se tornará o estado inicial do problema.

8. Se a execução for reiniciada, volta-se ao passo 1, porém com uma informação ao Agente Problema de que o estado inicial do sistema é aquele guardado pelo AM durante a interrupção da execução.

O resultado das ações acima gera uma capacidade de resolução de um problema que nenhum dos agentes apresentados possuía por si só. Este algoritmo de solução é válido para qualquer contexto de agentes e domínios que possam ser usados por esta metodologia, ou seja, não é um procedimento de solução específico do domínio de aplicação escolhido para a validação deste trabalho.

Supondo um problema em que quase todos os Agentes Mencionados estão disponíveis, com exceção do Agente Garra 4 e da Esteira nº 2. Existem quatro blocos a serem transportados. Os blocos chegam em uma ordem 1, 2, 3, 4 (sendo o bloco 4 o único que pode ser pego a princípio). O objetivo é que estes blocos estejam na mesa final na ordem 4, 3, 2, 1. Este problema proposto em formato PDDL é mostrado no APÊNDICE C, além de ser necessário para o perfeito entendimento dos passos da solução encontrada:

Após 5 tentativas (l*oops*) do procedimento da solução mostrada acima, o sistema consegue encontrar uma solução que é mostrada logo a seguir em 30 passos. Mesmo tendo todos os agentes disponíveis, o APDI só embarca os agentes necessários à resolução. Para este problema foram utilizados os seguintes Agentes e Recursos:

- Mesas: Inicial e Final;
- Garras: 1, 2 e 3;
- Esteira 1;
- Magazines 1 e 2;

Os passos da solução encontrada foram:

```
 0: PEGAR_G3 BLOCO4 MI4
 1: ANDARFILA MESAINICIAL MI3 MI4
 2: SOLTAR_G3 BLOCO4 ESTEIRA11
 3: PEGAR_G1 BLOCO3 MI4
 4: TRANSPORTARBLOCOESTEIRA1 BLOCO4 ESTEIRA11 ESTEIRA12
 5: ANDARFILA MESAINICIAL MI3 MI4
 6: PEGAR_G2 BLOCO4 ESTEIRA12
 7: PEGAR_G3 BLOCO2 MI4
 8: ANDARFILA MESAINICIAL MI3 MI4
 9: SOLTAR_G1 BLOCO3 MAGAZINE1
10: PEGAR_G1 BLOCO1 MI4
11: SOLTAR_G1 BLOCO1 ESTEIRA11
12: PEGAR_G1 BLOCO3 MAGAZINE1
13: TRANSPORTARBLOCOESTEIRA1 BLOCO1 ESTEIRA11 ESTEIRA12
14: SOLTAR_G3 BLOCO2 ESTEIRA11
15: SOLTAR_G2 BLOCO4 MAGAZINE2
16: PEGAR_G2 BLOCO1 ESTEIRA12
17: SOLTAR_G2 BLOCO1 MF1
18: TRANSPORTARBLOCOESTEIRA1 BLOCO2 ESTEIRA11 ESTEIRA12
19: PEGAR_G2 BLOCO2 ESTEIRA12
20: ANDARFILA MESAFINAL MF1 MF2
21: SOLTAR_G2 BLOCO2 MF1
22: SOLTAR_G1 BLOCO3 ESTEIRA11
23: ANDARFILA MESAFINAL MF2 MF3
24: TRANSPORTARBLOCOESTEIRA1 BLOCO3 ESTEIRA11 ESTEIRA12
25: PEGAR_G2 BLOCO3 ESTEIRA12
26: SOLTAR_G2 BLOCO3 MF1
27: ANDARFILA MESAFINAL MF3 MF4
28: PEGAR_G2 BLOCO4 MAGAZINE2
29: SOLTAR_G2 BLOCO4 MF1
```

Exemplo 6 – Passos da Solução Integral (*Stream* do Planejador)

Para uma melhor visualização do plano gerado, em uma forma gráfica, a Figura 23 mostra o mesmo plano na forma de Diagrama de seqüência.

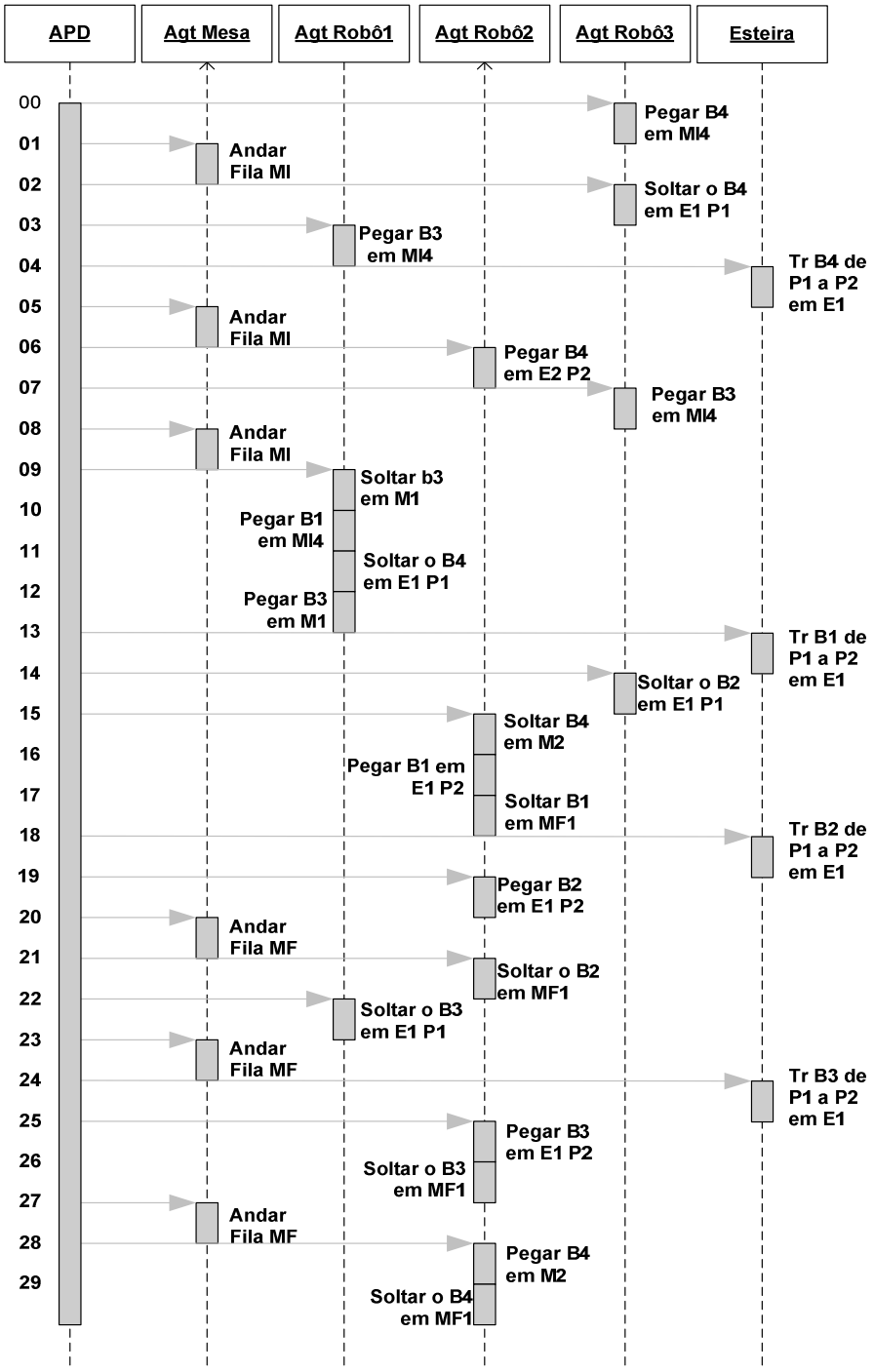

Figura 23 – Diagrama de Seqüência do plano gerado

A solução foi encontrada de forma correta, podendo o plano ser executado. E assim foi feito. Porém, para efeito de teste da dinâmica do sistema, uma simulação de parada ocorreu a execução do passo 6 deste plano gerado. Neste passo 6, a

Garra 2 deveria pegar o Bloco 4 na posição P2 da esteira 1. Durante esta parada o Agente Garra 2 foi indisponibilizado no sistema juntamente com os demais Magazines (1, 2, 3 e 4), impedindo o plano de seguir seu caminho anteriormente traçado. Porém, a Esteira 2 e o Agente Garra 4 foram inseridos no contexto do sistema, ou seja, foram disponibilizados. Com isso, tem-se a opção de replanejar a solução de acordo com o último estado do ambiente que foi guardado e se realizar uma nova tentativa de busca para um novo plano que continue a execução a partir do momento da parada.

Este replanejamento segue o mesmo procedimento de solução apresentado anteriormente, e após 4 tentativas (*loop* do procedimento) o sistema consegue encontrar uma nova solução que incorpora novos recursos e agentes que antes não haviam sido utilizados.

O novo conjunto de agentes e recursos utilizados é:

- Mesas: Inicial e Final;
- Robôs: 1, 3 e 4;
- Esteiras: 1 e 2;

Os passos da nova solução são mostrados a seguir:

```
 0: PEGAR_G1 BLOCO2 MI4
 1: ANDARFILA MESAINICIAL MI3 MI4
 2: PEGAR_G3 BLOCO1 MI4
 3: SOLTAR_G3 BLOCO1 ESTEIRA21
 4: TRANSPORTARBLOCOESTEIRA2 BLOCO1 ESTEIRA21 ESTEIRA22
 5: PEGAR_G4 BLOCO1 ESTEIRA22
 6: SOLTAR_G4 BLOCO1 MF1
 7: SOLTAR_G1 BLOCO2 ESTEIRA21
 8: TRANSPORTARBLOCOESTEIRA2 BLOCO2 ESTEIRA21 ESTEIRA22
 9: PEGAR_G4 BLOCO2 ESTEIRA22
10: ANDARFILA MESAFINAL MF1 MF2
11: SOLTAR_G4 BLOCO2 MF1
12: SOLTAR_G1 BLOCO3 ESTEIRA21
13: ANDARFILA MESAFINAL MF2 MF3
14: TRANSPORTARBLOCOESTEIRA2 BLOCO3 ESTEIRA21 ESTEIRA22
15: PEGAR_G4 BLOCO3 ESTEIRA22
16: SOLTAR_G4 BLOCO3 MF1
17: ANDARFILA MESAFINAL MF3 MF4
18: PEGAR_G4 BLOCO4 ESTEIRA12
19: SOLTAR_G4 BLOCO4 MF1
```

Exemplo 7 – Passos da solução após parada no 6º passo (*Stream* do Planejador)

Algumas conclusões podem ser retiradas deste novo plano. A Garra 4, que foi adicionada no grupo de agentes, foi utilizada na nova solução, o que é totalmente observável, uma vez que esta irá fazer a mesma função atribuída anteriormente à Garra 2. Porém há de se notar que o Magazine 4 não foi utilizado pois foi indisponibilizado no momento da parada. Se o plano inicial utilizava o Magazine 2 para armazenar blocos através da Garra 2, é de se esperar que a Garra 4 necessite analogamente do Magazine 4. A explicação para tal fato não ter impedido a geração de um novo plano está na inclusão da Esteira 2 no Grupo de Agentes. Durante a busca do novo conjunto de agentes, a esteira 2 foi utilizada e uma diferente solução pode ser encontrada. Tal fato também explica a diferença de passos na solução do problema. A solução sem parada possuía 30 passos. Como a nova solução possui 20 passos e antes da parada foram executados 5 passos, o planejamento resultante foi menor devido à utilização de mais uma esteira que possui uma capacidade de mover peças e funcionar como um *buffer* com uma capacidade maior que a de um Magazine.

A qualidade do plano não pode ser medida apenas pelo número de ações. É de se esperar que determinados recursos e/ou agentes tenham um custo maior de utilização, somando-se ao fato de que para alguns processos, não é interessante a utilização de dois agentes com características semelhantes para realizar uma tarefa onde apenas um destes agentes seria o suficiente. Além disso, com os novos recursos da linguagem PDDL como otimização [26], custo de ações [27], preferências [28], entre outras, poderia se definir custos e prioridades para cada agente, o que levaria a uma melhora qualitativa na escolha dos agentes durante o planejamento. A escolha do critério qualidade do plano, é portanto uma questão de aplicação, ou seja, dependerá sempre do domínio e objetivo.

Para ressaltar o problema da alocação desnecessária de agentes e recursos um novo problema será mostrado. Ao contrário do exemplo anterior, onde os quatro blocos de entrada estão na pior ordenação possível (ordem contrária à ordem desejada), um problema de ordenação mais simples possível será analisado, onde

os blocos estão na mesma ordem de entrada que deverão estar na saída do sistema de manufatura, ou seja, o mais simples dos casos.

Por se tratar de um problema seqüencial, uma solução para infinitos blocos (por limitações do planejador FF, o número máximo de blocos suportados neste tipo de problema foi de 19) pode ser encontrada, por exemplo, com um conjunto simples de agentes: Agente Mesa, Agente Garra 1 (ou 3) e 2 (ou 4) e Agente Esteira 1 (ou 2). E os testes para qualquer número de blocos (entre 1 e 19) mostraram este resultado considerando-se a total disponibilidade de todos os agentes. A solução para este problema com 10 blocos foi encontrada em 67 passos. Os agentes e recursos utilizados foram os Agentes: Mesa, Garra 1, Garra 2 e Esteira 1.

Para testar a eficiência da busca direcionada que auxilia a escolha de agentes, o mesmo problema foi apresentado ao mesmo planejador FF, porém, como arquivo de domínio de entrada, foi montado um domínio contendo todos os recursos e agentes existentes (Todas as esteiras, garras e magazines). O planejador conseguiu montar um plano com os mesmos 67 passos, porém além dos Agentes Mesa, Garra 1, Garra 2 e Esteira 1, também o plano utiliza os agentes: Garra 3, Garra 4 e Esteira 2. Este fato mostra na prática o problema discutido na Figura 7, onde acontece de agentes serem desnecessariamente alocados. Os passos e comparações deste problema podem ser encontrados no APÊNDICE D.

É importante ressaltar que em sistemas onde o que importa é uma solução com número de passos reduzido, independente do número de agentes alocados, este objetivo pode ser perfeitamente alcançado simplesmente desabilitando esta busca direcionada.

Outros exemplos de testes e problemas e plano gerados podem ser encontrados no APÊNDICE D.

7.2. O Software APDI

O sistema foi desenvolvido na Plataforma do Visual Studio .NET da Micosoft em linguagem C#. As chamadas às funções equivalentes às ações são feitas de forma interativa e de forma real. Isto que dizer que, por exemplo, ao ser executado o passo “PEGAR_G3 BLOCO4 MI4”, o programa irá chamar o método correspondente e irá passar os parâmetros “BLOCO4” e “MI4”.

No caso das simulações realizadas neste trabalho, o conteúdo dos métodos invocados, ou seja, o que o método realmente faz, é uma chamada a um *Messagebox* que mostra uma string com o nome da ação e os parâmetros utilizados formando uma frase. Um exemplo do método “Pegar_G1” pode ser visto a seguir.

```
public bool Pegar_G1(parameter Bloco, parameter localizacao_bloco)
        {
            if(!APDI.MessageBoxAspectoFuncional("G1 Pega " + Bloco.PName +
                                                " em " + localizacao_bloco.PName))
                return false;


            return true;

        }
```

Exemplo 8 – Método “Pegar”

Apesar da simplicidade do conteúdo dos métodos de execução das ações, a metodologia pôde ser validada, uma vez que pode se implementar qualquer rotina dentro dos métodos, e os parâmetros são passados de forma clara. Se o domínio de aplicação deste trabalho fosse um ambiente real, de forma relativamente simples, seria possível a comunicação com o robô com todas as instruções para a realização da tarefa requerida. Na Figura 24, é mostrada a Tela do Sistema APDI.

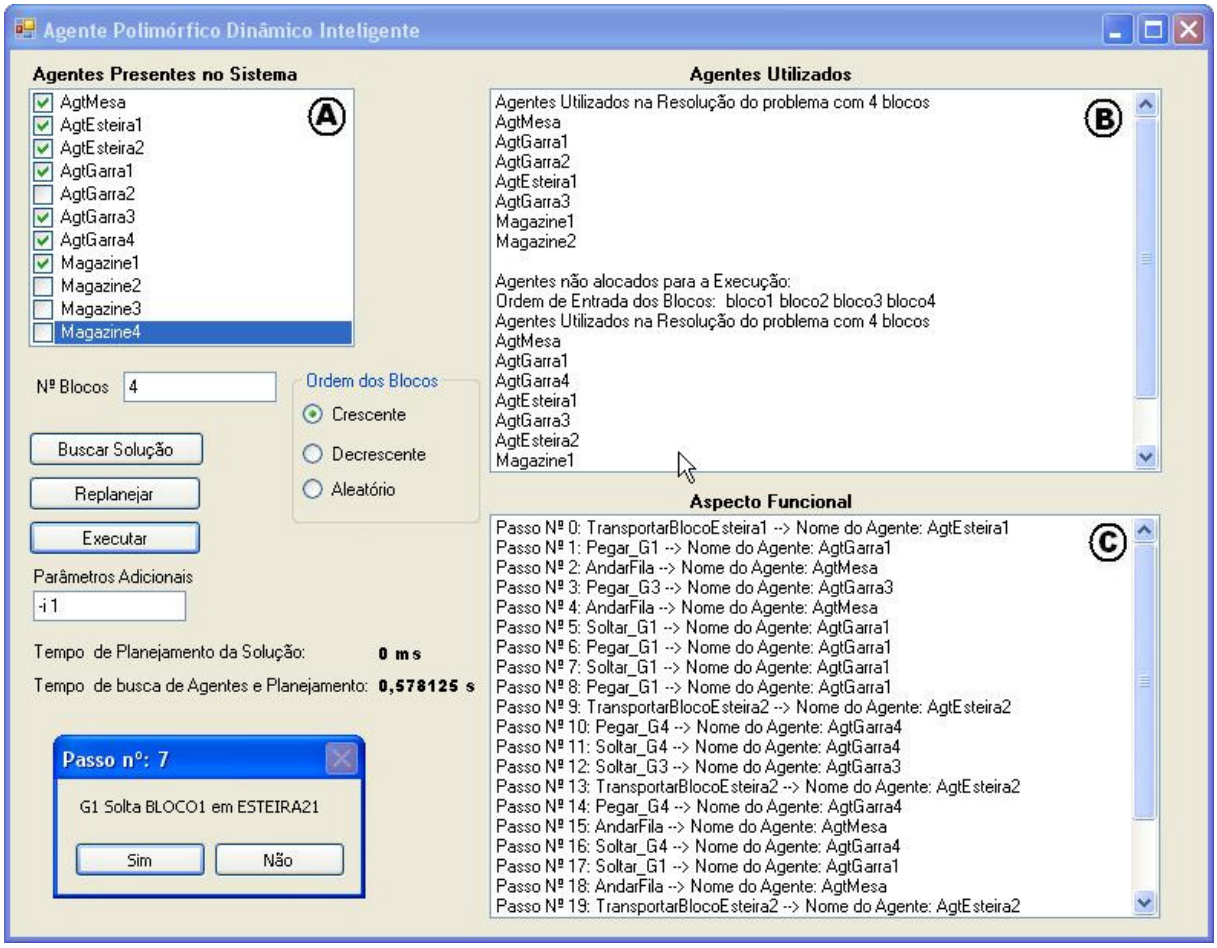

Figura 24 – Tela do Software do APDI

No quadro A da Figura 24 está a lista de agentes e recursos detectados pelo Agente Monitor. A pós a detecção, o usuário pode disponibilizar ou não cada um dos agentes e recursos. O usuário então define a ordem de entrada dos blocos e o número de blocos. Ao clicar em “Buscar Solução” o sistema irá realizar o planejamento conforme o algoritmo descrito anteriormente. Após o término da solução, se encontrada, o software pergunta se deseja executar o plano gerado. Caso não, posteriormente clicando no botão “Executar” o plano gerado poderá ser executado. O Quadro B mostra a ordem de entrada de blocos escolhida, bem como os agentes que foram utilizados para a busca do plano, não necessariamente os agentes utilizados para a busca estão contidos no plano de ação. O Quadro C mostra uma aproximação textual da representação do Aspecto Funcional.

A cada passo o usuário tem a opção de executar ou não o passo, este interatividade pode inexistir sem problemas tornando o sistema automático, porém para efeito de simulação, é uma boa ferramenta, principalmente para simular a retirada de Agentes do sistema. Caso o usuário resolva não executar o passo, o estado corrente é guardado. Clicando em "Executar" o plano gerado segue sua execução normal. Mas se o usuário clicar em "Replanejar", o sistema irá realizar uma nova busca com base nos agentes que estão disponíveis, ou seja, se não for indisponibilizado nenhum agente, o plano provavelmente será o mesmo, caso contrário, se agentes forem excluídos do sistema ou adicionados, um novo plano será gerado.

7.3. Aplicações possíveis da Metodologia

O sistema se mostrou extremamente flexível e com grande poder de aplicabilidade. As aplicações dessa metodologia se estendem em diversas áreas, tornando-a muito promissora, principalmente no caso de células de manufatura com diferentes agentes, onde o planejamento de ações é necessário.

As limitações dessa metodologia estão amarradas com as limitações da linguagem PDDL. Qualquer sistema, seja de manufatura, logística, otimização, etc, e que possam ser modelados em PDDL podem usufruir das funcionalidades deste sistema.

Outra aplicação possível para este trabalho é, olhando em um nível diferente de abstração, a utilização em sistemas multi-domínios PDDL, especificamente em sintetização de planos. Poucos são os trabalhos na área, e o que se pode encontrar hoje na literatura são propostas como a de [37], que não foram feitas totalmente para o padrão de linguagem PDDL. Considerando que cada agente é um domínio diferente, o sistema proposto nesta dissertação consegue resolver problemas de multi-domínios PDDL.

Outra utilidade interessante da metodologia apresentada neste trabalho, é que, dada a independência de cada Agente Regular, cada um deles poderia ser modelado por diferentes especialistas deixando a cargo da arquitetura de APDI, a responsabilidade de coordenar e unir as funcionalidades destes agentes de modo a se resolver determinado problema. Desta forma, ocorre uma modularização dos agentes, permitindo eventuais melhorias e modificações nestes agentes de acordo com as necessidades sem influenciar o restante do sistema.

8. Conclusão e Trabalhos Futuros

8.1. Conclusão

Este trabalho de dissertação propôs uma arquitetura inovadora de agentes inteligentes. Utilizando um *framewok* que trabalha através de *reflection*, foi possível gerar uma arquitetura apresentando uma estrutura de hiperdados capaz de descrever as funcionalidades dos agentes, um monitor responsável por identificar e informar ao planejador quais as funcionalidades disponíveis e um aspecto funcional capaz de gerar uma entidade polimórfica durante o tempo de execução com inteligência dedicada à resolução de um determinado problema. O cenário utilizado para validação do método foi um ambiente de manufatura simulado e que, apesar da simplicidade, foi suficiente para validar o conceito de APDI. O resultado apresentado corrobora a metodologia ao conseguir solucionar o problema através de uma entidade nova, não existente antes da geração do aspecto funcional, além de encontrar diferentes soluções para um mesmo problema de acordo com os recursos disponíveis. Durante uma eventual falha, indisponibilização de determinado agente, ou qualquer coisa que mude o estado corrente do ambiente durante a execução das ações, o sistema consegue realizar um replanejamento com a finalidade de encontrar uma solução alternativa, para que o estado objetivo seja alcançado. O sistema se mostrou extremamente flexível, uma vez que domínios completamente diferentes podem ser modelados e utilizados neste sistema, com inserção, remoção ou atualização de funcionalidades e agentes de forma trivial. Portanto o sistema possui um grande poder de aplicabilidade podendo ser aprimorado e melhorado futuramente. Este trabalho gerou uma publicação nos anais do XVI Congresso Brasileiro de Automática – CBA 2006 [38] e também possui um artigo aprovado e que, portanto, será apresentado no VIII Simpósio Brasileiro de Automação Inteligente – SBAI 2007.

8.2. Trabalhos Futuros

A modelagem do sistema exige um grande conhecimento da linguagem PDDL além de alguns ajustes de código. O que poderia ser feito com relação à interface com o usuário seria uma possível modelagem gráfica dos agentes e recursos de forma similar à utilizada por Dimitri Vlaska [29], mostrada no Capítulo 2 com as devidas adaptações. Desta forma o sistema teria uma interface mais amigável permitindo sua utilização por um maior número de usuários, e consequentemente, um maior número de casos.

Seria interessante também, a adequação deste sistema para os mais novos recursos da linguagem PDDL que se encontra na versão 3.0 neste ano de 2007. Desta forma, as classes que fazem *parse*, serialização e gerenciamento da linguagem PDDL, construídas neste trabalho, teriam que ser atualizadas. Da mesma forma, para atender aos novos requisitos da linguagem, seria necessária a utilização de um novo planejador que atenda aos requisitos do PDDL 3.0.

Já está sendo estudado um domínio semelhante ao domínio “Logística” ,que é bastante conhecido na comunidade de Planejamento Inteligente. Este domínio , derivado do “Logística”, compreende uma série de veículos com diferentes características como velocidade, capacidade de armazenamento e custo. O Problema é, dada uma série de produtos com diferentes origens e diferentes destinos, tem-se o objetivo de encontrar uma seqüência de ações que resolva o problema otimizando parâmetros como tempo, custo ou recursos, de acordo com a necessidade.

É de se esperar, para futuros estudos e uma provável tese de Doutorado, a continuidade deste trabalho, onde a pretensão está no fato de se encontrar domínios mais complexos e reais de Resolução Distribuída de Problemas, para a aplicação prática da metodologia no mundo real.

Referências Bibliográficas

[1] YOSHIDA, H., KAWATA, K., FUKUYAMA, Y., TAKAYAMA, S. E NAKANISHI, Y.: A Particle Swarm Optimization for Reactive Power and voltage Control Considering Voltage Security Assessment, IEEE Transactions on Power Systems, Vol. 15 nº 4, 2006.

[2] TABUADA, P., PAPPAS, G. J. E LIMA, P.: Motion Feasibility of Multi-Agent Formations, IEEE Transactions on Robotics, Vol. 21, nº 3. 2005.

[3] BAGNALL, A. J. E SMITH, G. D.: A Multiagent Model of the UK Market in Electricity Generation, IEEE Transactions on Evolutionary Computation, Vol 9. Nº 5, 2006.

[4] KICZALES, G., et al.: Aspect-Oriented Programming. In European Conference on Object-Oriented Programming, ECOOP'97, Springer-Verlag LNCS 1241, Finlândia (1997).

[5] CARDIERI, M. A. C. A.: Agentes Inteligentes : Noções Gerais. Monografia. Unicamp, 1998.

[6] RUSSEL, K. E NORVIG, P.: Artificial Intelligence, A Modern Aproach, Ed. Prentice Hall, 1995.

[7] FRANKLIN, S. AND GRAESSER, A.: Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agent. In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, 21–35. Springer-Verlag, 2006.

[8] SCHERER, A. E SCHLAGETER, G. C.: A Multi-Agent Approach for the Integration of Neural Networks and Expert Systems. Intelligent Hybrid Systems, John Wiley & Sons, 1995.

[9] DURFEE, E. H., ROSENSCHEIN, J. S.: Distributed problem solving and multi-agent systems: Comparisons and examples. In Proc. of 13th International Distributed AI Workshop, 94—104, 1994.

[10] SYCARA, K.: Multiagent Systems. AI magazine, 19(2):79-92, 1998.

[11] H. GEFFNER. Perspectives on Artificial Intelligence Planning, Proc. AAAI, 1013--1023, 2002.

[12] NEWELL A., SIMON H.A.: GPS, a program that simulates human thought, in Feigenbaum E.A., Feldman J. (eds.): Computers and Thought, New York: McGrawHill, pp. 279--296, 1963.

[13] FIKES, R. E., NILSSON, N. J.: STRIPS: a new approach to the application of theorem proving to problem solving, Artificial Intelligence 2, 189-208, 1971.

[14] IPC 2006 – 5th International Planning Competition. Hospedado na International Conference on Automated Planning and Scheduling. Disponível em: <http://www.plg.inf.uc3m.es/icaps06/competition.htm>. Acesso em: 03 abr. 2007.

[15] PEDNAULT, E., "Toward a Mathematical Theory of Plan Synthesis"; PhD Dissertation, Computer Science Department, Stanford University, 1987

[16] GHALLAB, M. et al.: PDDL---The Planning Domain Definition Language. AIPS-98 Planning Committee, 1998.

[17] GEREVINI, A., LONG, D.: BNF Description of PDDL3.0, IPC 2006, Deterministic Part. Out. Disponível em: <http://zeus.ing.unibs.it/ipc-5/pddl.html>. Acesso em: 03 Abr. 2007.

[18] NIGENDA RS, NGUYEN X, KAMBHAMPATI S.: AltAlt: Combining the advantages of Graphplan and heuristic state search. Technical Report; Arizona State University, 2000.

[19] KAUTZ, H., SELMAN, B. 1998. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. Workshop Planning as Combinatorial Search, AIPS-98, Pittsburgh, PA, 1998.

[20] HOFFMANN, J.: FF: The Fast-Forward Planning System. AI Magazine 22[3], AAAI Press, (2001) 57-62.

[21] DO, B., KAMBHAMPATI, S. Solving Planning Graph by Compiling it into CSP. Proc. AIPS-00 (to appear). 2000.

[22] GEREVINI, A., SERINA, I.: LPG: A Planner Based on Local Search for Planning Graphs. In Proc. of Sixth International Conference on Artificial Planning and Scheduling (AIPS-02), 2002.

[23] FOX, M., LONG, D.: Hybrid STAN: Identifying and managing combinatorial optimization sub-problems in planning. In Proceedings of IJCAI-2001, 2001.

[24] YOUNES, H., SIMMONS, R.: Vhpop: Versatile heuristic partial order planner. Journal of Artificial Intelligence Research 20:405—430, 2003.

[25] HOFFMANN, J. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In Proceedings of the Twelfth International Symposium on Methodologies for Intelligent Systems.

[26] YOUNES, H.L.S. et al.: "The First Probabilistic Track of the International Planning Competition", JAIR, Volume 24, pg 851-887, 2005.

[27] NYBLOM, P.: "Handling Uncertainty by Interleaving Cost-Aware Classical Planning with Execution", Proceedings of SAIS-SSLS, 2005.

[28] GEREVINI, A., LONG, D.: Plan constraints and preferences in PDDL3. Technical Report RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia, 2005.

[29] VRAKAS D., VLAHAVAS, i.: ViTAPlan: A Visual Tool for Adaptive Planning, Proceedings of the 9th Panhellenic Conference on Informatics, Thessaloniki, Greece, 2003.

[30] RICHARD M. STALLMAN, GNU EMACS Manual, iUniverse, Incorporated, 2000. Disponível em: <http://www.gnu.org/software/emacs/emacs.html>. Acesso em: 03 Abr. 2007.

[31] SINGHI, S.: PDDL Module for Emacs. V 0.100, 2005. Disponível em: <http://www.public.asu.edu/~sksinghi/pddl.htm>. Acesso em: 03 Abr. 2007.

[32] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M. (1998): "Extensible Markup Language (XML)", World Wide Web Consortium Recommendation REC-xml-19980210, February, 1998. Disponível em: <http://www.w3.org/TR/1998/REC-xml-19980210.html>. Acesso em: 03 Abr. 2007.

[33] THE C# LANGUAGE. Microsoft Corporation, 2005. Disponível em: <http://msdn.microsoft.com/vcsharp/programming/language/>. Acesso em: 03 Abr. 2007.

[34] DOT NET FRAMEWORK DEVELOPMENT CENTER. Microsoft Corporation, 2005. Disponível em: <http://msdn.microsoft.com/netframework/>. Acesso em: 03 Abr. 2007.

[35] LINDHOLM, T., YELLIN, F. The Java Virtual Machine Specification. Addison-Wesley, 1999. Disponível em: <http://java.com/pt_BR/download/index.jsp>. Acesso em: 03 Abr. 2007.

[36] HEINDEL, L. E., KASTEN, V. A.: "Highly Reliable Synchronous and Asynchronous Remote Procedure Calls", IEEE, pp. 103-107, Maio 1996.

[37] BOUGUERRA, A., KARLSSON L.: Synthesizing Plans for Multiple Domains. In Proceedings of SARA'2005. pp.30-43, 2005.

[38] HONORIO, L. M., Vidigal, M. C. A. C. F., Barbosa, D. A., Bastos, G. S. Agentes Polimórficos Dinâmicos Inteligentes: Teoria e Técnicas de Implementação In: XVI Congresso Brasileiro de Automática, Salvador - BA. Anais do Congresso Brasileiro de Automática (CBA 2006). , v.1. p.2213 – 2218, 2006.

APÊNDICE A – Grafos de Planejamento

Os grafos de planejamento são estruturas de dados (grafos) que representam planos e podem ser construídos eficientemente. São muito usados para extrair heurísticas que podem ser aplicadas a qualquer técnica de busca existente.

Além do advento de fornecer uma boa heurística, pode-se extrair uma solução diretamente de um grafo de planejamento, utilizando um algoritmo especializado como o “Planejamento em grafo”.

Um grafo de planejamento consistem uma seqüência de níveis que correspondem a períodos de tempo no plano, em que nível 0 é o estado inicial. Cada nível contém um conjunto de átomos e um conjunto de ações. Em linhas gerais, os átomos são todos os que poderiam ser verdadeiros nesse período de tempo, dependendo das ações executadas nos períodos de tempo precedentes. De forma análoga, as ações são todas elas que tem suas precondições satisfeitas pelos átomos do nível anterior. Como o grafo é montado com várias ações em paralelo, ele pode ser um pouco otimista com relação a um número mínimo de períodos de tempo ou passos para que determinado átomo se torne verdadeiro, mas de qualquer forma este número de passos dá uma boa estimativa de dificuldade que se tem de se alcançar determinado átomo ou conjunto de átomos.

Para um melhor entendimento dos grafos de planejamento, será analisado o Exemplo 9, extraído de [6], que é um problema bem simples e de fácil entendimento, mas que se fosse mais complexo, a demonstração visual do grafo se tornaria muito extensa. O grafo de planejamento relacionado ao Exemplo 9 é mostrado na Figura 25.

O Nível de estado S_0 representa o estado inicial do problema. O nível de ação A0 são apresentadas todas as ações cujas precondições são satisfeitas no nível anterior. Cada ação está conectada a suas precondições em S_0 e a seus efeitos em S_1, introduzindo neste caso em S_1 todos os átomos que não estavam presentes em S_0.

Inicio **(Ter(Bolo))**
Objetivo **(Ter(Bolo) Λ Comido(Bolo))**
Ação**(Comer(Bolo)**
PRECOND: Ter(Bolo)
EFFECT: ¬Ter(Bolo) Λ Comido(Bolo))
Ação **(Assar(Bolo)**
PRECOND: ¬Ter(Bolo)
EFFECT: Ter(Bolo))

Exemplo 9 – Problema de "ter bolo e comer bolo também"

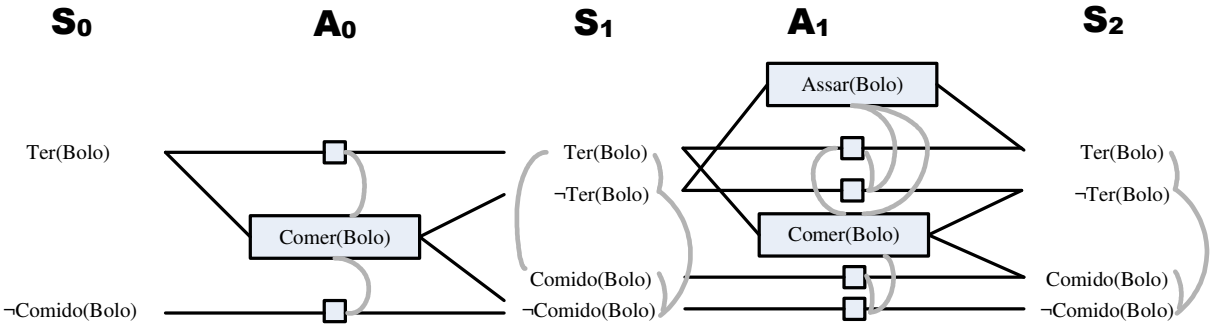

Figura 25 – Grafo de planejamento para o problema "ter bolo e comer bolo também" até o nível S_2.

Os vínculos de exclusão mútua (*Mutex*) estão representados por linhas curvas de cor cinza, e a forma na qual são computados será explanado posteriormente..

O grafo de planejamento precisa de uma forma de expressar a inação, bem como a ação, ou seja, ele precisa permitir que um átomo permaneça verdadeiro após a realização de uma ação. Para todo átomo X positivo e negativo, é adicionada uma ação de pertinência com precondição X e efeito X. Os Retângulos indicam ações (pequenos quadrados indicam ações de pertinência) e linhas retas indicam precondições e efeitos.

O nível A_0 contém todas as ações que poderiam acontecer no estado S0 e, além disso, registra conflitos entre ações. Os vínculos de exclusão mútua (*Mutex*)

estão representados por linhas curvas de cor cinza. Como exemplo, *Comer(Bolo)* é mutuamente exclusiva com a persistência de *Ter(Bolo)* ou de ¬*Comido(Bolo).*

O nível S_1, além de conter todos os átomos que poderiam resultar da escolha de qualquer subconjunto das ações em A_0, também contém vínculos *mutex*, indicando literais que não poderiam aparecer juntos. Como exemplo *Ter(Bolo)* e *Comido(bolo)* são de exclusão mútua: dependendo da escolha da ação em A_0 um ou outro poderia ser verdadeiro mas não ambos. Comparando com a busca no espaço de estados, S_1 representa estados múltiplos e os vínculos *mutex* são restrições que definem os estados possíveis.

Desta forma, a construção do grafo é continuada alternando entre o nível de estado Si e o nível de ação A_i, até chegar a um nível em que dois níveis consecutivos são idênticos. Neste momento é dito que o grafo se nivelou. Todo nível subseqüente será idêntico e, portanto desnecessário.

Acaba-se então com uma estrutura na qual todo nível A_i contém todas ações que são aplicáveis em S_i, juntamente com a informação de quais pares de ações não podem ser executados simultaneamente. De forma análoga, todo nível S_i contém todos os átomos que poderiam ser gerados a partir da escolha de qualquer conjunto de ações de A_{i-1}. É importante ressaltar que a construção do grafo de planejamento requer a escolha de ações, o que exigiria uma busca combinatória. Ao invés disto, ele registra apenas a impossibilidade de certas escolhas através da aplicação de *mutex*. A complexidade de um grafo de planejamento é um polinômio de baixa ordem em relação ao número de ações e átomos, enquanto na busca no espaço de estados a complexidade é exponencial em relação ao número de átomos.

Existem três condições para se determinar vínculos *mutex* entre ações em um determinado nível:

- *Efeitos inconsistentes*: Uma ação nega o efeito da outra. Ex.: *Comer(Bolo)* e a persistência *Ter(Bolo)*, que discordam com relação à *Ter(Bolo)*.

- *Interferência*: um dos efeitos de uma ação é a negação da precondição da outra ação. Ex.: *Comer(Bolo)* interfere com a persistência de *Ter(bolo)* negando sua precondição.

- *Necessidades Concorrentes*: Uma das precondições de uma ação é mutuamente exclusiva com uma precondição da outra. Ex.: *Assar(Bolo)* e *Comer(Bolo)* são mutuamente exclusivas, porque competem no valor da precondição.

Existem duas condições para se determinar vínculos *mutex* entre átomos de um determinado nível:

- Se um átomo é a negação do outro. Ex.: Em S_1, *Ter(Bolo)* e ¬Ter(Bolo).

- Se cada par possível que alcançariam os dois átomos são mutuamente exclusivos. Ex.: *Ter(Bolo)* e *Comido(Bolo)* são de exclusão mútua em S_1, porque a única maneira de alcançar *Ter(Bolo)* , a ação de persistência é de exclusão mútua com a única Maneira de alcançar *Comido(Bolo),* ou seja, *Comer(Bolo).* Em S_2, os dois literais não são de exclusão mútua porque existem novas maneiras de alcançá-los, como *Assar(Bolo)* e a persistência de *Comido(Bolo)* que não são de exclusão mútua.

Uma vez construído o grafo de planejamento, ele se torna uma poderosa ferramenta para se extrair informações sobre o problema. Como exemplo, se um átomo não aparece no último nível do grafo, indica que o problema não possui solução, ou seja, o custo para se alcançar o objetivo é infinito. De forma análoga, o custo de se alcançar qualquer conjunto de átomos, pode ser relacionado com o nível em que ele aparece no grafo. Desta forma, vários planejadores utilizam o grafo de planejamento para a estimação do valor da função heurística, como é o caso do planejador FF [25], que realizando uma busca no espaço de estados, utiliza-se desta estimação heurística para definir a prioridade de expansão de determinado nó durante a busca.

Por outro lado, o plano pode ser extraído diretamente do grafo de planejamento montado ao invés de simplesmente utilizá-lo para extração de heurística. O algoritmo utilizado para tal tarefa de extração de plano verifica se todos os átomos do objetivo estão presentes no nível atual do grafo, e se estão sem vínculo de exclusão mútua entre eles. Se sim, *existe a possibilidade* de existir alguma solução dentro do grafo atual, e, portanto o algoritmo tenta extrair esta solução. Caso contrário, o grafo é expandido, e as ações correspondentes ao próximo nível são adicionadas, e conseqüentemente os átomos do próximo nível. O processo segue a recorrência até que uma solução seja encontrada ou até que se determine que não exista nenhuma solução.

APÊNDICE B - Descrição dos Agentes Regulares utilizados

Agentes do tipo Garra

Os Agentes do tipo *Garra Manipuladora* possuem a capacidade de pegar blocos e colocá-los em diferentes posições. O modelo das Garras é similar, porém cada qual com suas limitações de localização e alcance. Elas conseguem realizar duas ações distintas que são “Pegar” e “Soltar” blocos. O Modelo PDDL é descrito e comentado a seguir. O Exemplo 10 é a declaração de predicados no formato PDDL, comentado, dos Agentes do tipo Garra, onde ‘X’ representa o número do agente.

```
(:predicates
(alcanca-GX  ?p - posicao ) // Define uma posição alcançável pela Garra X
(braco-livre-GX )  //Indica a disponibilidade da Garra X
(permite-pegar  ?p - posicao ) //Define as posições onde é permitida a retirada de blocos
(Localizado  ?b - bloco  ?p - posicao ) //Define a Localização do Bloco
(segurando-GX  ?b - bloco ) //Indica qual objeto a Garra X está segurando
(PosDisponivel  ?p - posicao ) //Define posições vazias, sem objetos
(permite-colocar  ?p - posicao ) //Define as posições onde é permitida a alocação de blocos
)
```

Exemplo 10 – Predicados dos Agentes do Tipo Garra

A ação “Pegar” mostrada a seguir no Exemplo 11, possui 2 parâmetros: 1 bloco e 1 posição. De forma literal pode-se entender que o significado da ação é que a Garra “pega” um determinado bloco “ob” que está localizado em uma determinada posição “pos”.

As precondições para que a garra possa pegar o bloco são:

1. A posição “pos” precisa estar ao alcance da garra;
2. A garra não pode estar segurando outro bloco;

3. A posição “pos” onde se encontra o bloco “ob” deve permitir a retira do mesmo;

4. O bloco “ob” deve estar na posição “pos”;

Os efeitos, ou adição de átomos após a realização da ação são:

1. A garra estará segurando o bloco “ob”;

2. A posição “pos” onde estava o bloco se tornará disponível;

3. O braço da garra não estará mais disponível;

4. O bloco “ob” não estará mais localizado na posição “pos”;

```
(:action Pegar-GX
      :parameters (?ob - bloco ?pos - posicao )
      :precondition  (and  (alcanca-GX ?pos )
                           (braco-livre-GX )
                           (permite-pegar ?pos )
                           (Localizado ?ob ?pos ) )
      :effect          (and (segurando-GX ?ob )
                           (PosDisponivel ?pos )
                           (not (braco-livre-GX ) )
                           (not (Localizado ?ob ?pos ) ) )
   )
```

Exemplo 11 – Ação “Pegar” no formato PDDL

A ação “Soltar” mostrada a seguir no Exemplo 12, possui também2 parâmetros: 1 bloco e 1 posição. De forma literal pode-se entender que o significado da ação é que a Garra “solta” um determinado bloco “ob” em uma determinada posição “pos”.

As precondições para que a garra possa soltar o bloco na posição são:

1. A garra precisa estar segurando o bloco “ob”;
2. A posição “pos” precisa estar disponível;
3. A posição “pos” precisa estar ao alcance da Garra;
4. Deve ser permitida a colocação de objetos nesta posição “pos”;

Os efeitos, ou adição de átomos após a realização da ação são:

1. O braço da garra estará disponível para pegar outro objeto;
2. O objeto “ob” estará localizado na posição “pos”;
3. A garra não estará mais segurando o bloco “ob”;
4. A posição “pos” não estará mais disponível;

```
(:action Soltar-GX
  :parameters (?ob - bloco ?pos - posicao )
  :precondition  (and (segurando-G1 ?ob )
                      (PosDisponivel ?pos )
                      (alcanca-G1 ?pos )
                      (permite-colocar ?pos ) )
  :effect          (and (braco-livre-G1 )
                      (Localizado ?ob ?pos )
                      (not (segurando-G1 ?ob ) )
                      (not (PosDisponivel ?pos ) ) ))
```

Exemplo 12 – Ação “Soltar” no formato PDDL

Agentes do tipo Esteira

Os Agentes do tipo Esteira Transportadora possuem a capacidade de transportar um bloco de cada vez de uma posição a outra. Estas posições são definidas por 2 sensores e podem ser melhor visualizadas na Figura 22. As esteiras

possuem uma única ação que é a de “TransportarBlocos”. O Exemplo 13 é a declaração de predicados no formato PDDL, comentado, dos Agentes do tipo Esteira, onde ‘X’ representa o número do agente.

```
(:predicates
(LimitesEsteiraX  ?pi - posicao  ?pf - posicao ) //Posições que definem os limites da esteira, e onde estão os sensores da mesma;
(Localizado  ?x bloco ?l – posicao)  //Define a Localização do Bloco
(PosDisponivel  ?p - posicao )  //Define posições vazias e disponíveis para a alocação de objetos;
```

Exemplo 13 – Predicados dos Agentes do Tipo Esteira

A ação “TransportarBlocoEsteiraX” mostrada a seguir no Exemplo 14, possui 3 parâmetros: 1 bloco e 2 posições. De forma literal pode-se entender que o significado da ação é que a esteira transporta um determinado bloco de uma posição “PosInicial” até uma posição “PosFinal” qualquer.As precondições para que a esteira possa transportar o bloco são:

1. As posições indicadas nos parâmetros devem ser as posições de localização dos sensores da esteira;
2. O bloco “b1” deve estar localizado na posição “Posinicial”;
3. Não pode existir um bloco qualquer “b2” que está localizado na posição “PosFinal”;

Os efeitos, ou adição de átomos após a realização da ação são:

1. O Bloco “b1” estará localizado na posição “PosFinal”;
2. A posição “PosInicial” estará disponível;
3. A posição “Posfinal” não estará mais disponível;
4. O bloco não estará mais localizado na posição “PosInicial”;

```
(:action TransportarBlocoEsteiraX
        :parameters (?b1 - bloco ?PosInicial - posicao ?Posfinal - posicao )
        :precondition  (and (LimitesEsteiraX ?PosInicial ?PosFinal )
                            (Localizado ?b1 ?PosInicial )
                            (not  (exists (?b2 - bloco )
                                          (Localizado ?b2 ?Posfinal ) )))
        :effect             (and (Localizado ?b1 ?Posfinal )
                                 (PosDisponivel ?PosInicial )
                                 (not (PosDisponivel ?Posfinal ) )
                                 (not (Localizado ?b1 ?PosInicial ) ) )
    )
```

Exemplo 14 – Ação Transportar Bloco no formato PDDL

Agentes do tipo Mesa

Na aplicação de trabalho desta dissertação, é utilizado um único Agente do Tipo Mesa. Este Agente controla duas mesas distintas, a de entrada e a de saída de blocos. Cada mesa possui a capacidade de armazenar uma determinada quantidade blocos com posições internas definidas. Estas mesas dão pequenos passos de forma a alocar os blocos em posições posteriores. O Agente do Tipo Mesa possui uma única ação que é a ação "AndarFila". O Exemplo 15 é a declaração de predicados do Agente Mesa no Formato PDDL.

```
(:predicates
(PosDisponivel  ?p - posicao )
 (Posterior  ?x - (either posicao bloco)  ?y - (either posicao bloco) ) //Indica o posicionamento de uma posição com relação à outra
(ultima-posicao  ?p - posicao ) //Indica qual a última posição da mesa
(Localizado  ?x - (either posicao bloco)  ?p - posicao ) )//Indica se determinado bloco ou posição está localizado em alguma posição ou local
```

Exemplo 15 – Predicados dos Agentes do Tipo Mesa

A ação “AndarFila” mostrada a seguir no Exemplo 16, possui 3 parâmetros: 1 local e 2 posições. Um local qualquer é também considerado uma posição. De forma literal pode-se entender que o significado da ação é que a mesa localizada em um determinado local “l” Dará um passo no sentido indicado pelas posições “p1” e “p2”. As precondições para que a mesa possa dar um passo no sentido requerido são:

1. Deve existir um bloco “b” qualquer tal que:

“b” está localizado em “p1”

“p2” é posterior a “p1”

“p2 deve estar disponível

“p1” e “p2 devem estar localizados no local “l”

Os efeitos, ou adição de átomos após a realização da ação são:

1. Para toda posição “pi”:

    1.1. Quando “pi”estiver localizada em l e “pi” for última posição

        1.1.1. “pi”estará disponível

2. Para todo bloco “b”, e posições “pi” e “pf”:

    2.1. Quando “pi”estiver localizado em “l” e “b”estiver localizado em “pi”e “pf” for posterior a “pi”:

        2.1.1. “b” estará localizado em “pf”, e não mais em “pi”

        2.1.2. “pf”não estará mais disponível

```
(:action AndarFila
        :parameters (?l - local ?p1 - posicao ?p2 - posicao )
        :precondition  (exists (?b - bloco )
                                (and (localizado ?b ?p1 )
                                     (Posterior ?p2 ?p1 )
                                     (PosDisponivel ?p2 )
                                     (localizado ?p2 ?l )
                                     (localizado ?p1 ?l ) ))
        :effect          (and (forall (?pi - posicao )
                                      (when   (and (localizado ?pi ?l )
                                                   (ultima-posicao ?pi ) )
                                                   (PosDisponivel ?pi ) ) ) )
                              (forall (?b - bloco ?pi - posicao ?pf - posicao )
                                      (when   (and     (localizado ?pi ?l )
                                                       (localizado ?b ?pi )
                                                       (Posterior ?pf ?pi ) )
                                              (and     (localizado ?b ?pf )
                                                       (not (localizado ?b ?pi ) )
                                                       (not (PosDisponivel ?pf ) ) ) ) ) ))
```

Exemplo 16 – Ação Andar Fila no formato PDDL

APÊNDICE C – Problema 4 blocos no formato PDDL

Este é o arquivo problema PDDL referente ao plano gerado no Exemplo 6.

```
(define (problem PrAPDI)                    ;; Nome do Problema
(:domain APDI)                              ;;Domínio de Aplicação do Problema
(:objects                                   ;;Variáveis Existentes
            MesaFinal - local
            MesaInicial - local
            mf1 - posição                   ;; mfx são posições internas  da
                                            mesa final geradas automaticamente
                                            de acordo com o número de blocos
                                            do problema. Da esquerda para a
                                            direita o número vai de 1 a n,
                                            onde n é o número de blocos. O que
                                            acontece de forma análoga com as
                                            posições mix declaradas a seguir.
            mi1 - posicao
            mf2 - posicao
            mi2 - posicao
            mf3 - posicao
            mi3 - posicao
            mf4 - posicao
            mi4 - posicao
            Esteira11 - posição                 ;EsteiraXY – X indica o
                                                número da esteira e Y indica
                                                qual posição(P1 ou P2)
            Esteira12 - posicao
            Magazine1 - posicao
            Magazine2 - posicao
            bloco1 - bloco
            bloco2 - bloco
            bloco3 - bloco
            bloco4 - bloco
  )
(:init
            (PosDisponivel MesaFinal )
            (permite-colocar MesaFinal )
            (permite-pegar MesaInicial )
            (PosDisponivel mf1 )
            (Localizado mi1 MesaInicial )
            (Localizado mf1 MesaFinal )
            (Posterior mf2 mf1 )            ;;Indica a ordem final  de
                                            um bloco com relação ao
                                            outro.
            (Posterior mi2 mi1 )
            (PosDisponivel mf2 )
            (Localizado mi2 MesaInicial )
            (Localizado mf2 MesaFinal )
```

```
              (Posterior mf3 mf2 )
              (Posterior mi3 mi2 )
              (PosDisponivel mf3 )
              (Localizado mi3 MesaInicial )
              (Localizado mf3 MesaFinal )
              (Posterior mf4 mf3 )
              (Posterior mi4 mi3 )
              (PosDisponivel mf4 )
              (Localizado mi4 MesaInicial )
              (Localizado mf4 MesaFinal )
              (ultima-posicao mf1 )
              (ultima-posicao mi1 )
              (permite-colocar mf1 )
              (permite-pegar mi4 )
              (braco-livre-G1  )
              (alcanca-G1 mi4 )
              (alcanca-G1 Esteira11 )
              (braco-livre-G2  )
              (alcanca-G2 mf1 )
              (alcanca-G2 Esteira12 )
              (LimitesEsteira1 Esteira11 Esteira12 )
              (permite-colocar Esteira11 )
              (permite-pegar Esteira12 )
              (PosDisponivel Esteira11 )
              (PosDisponivel Esteira12 )
              (braco-livre-G3  )
              (alcanca-G3 mi4 )
              (alcanca-G3 Esteira11 )
              (alcanca-G1 Magazine1 )
              (permite-colocar Magazine1 )
              (permite-pegar Magazine1 )
              (PosDisponivel Magazine1 )
              (alcanca-G2 Magazine2 )
              (permite-colocar Magazine2 )
              (permite-pegar Magazine2 )
              (PosDisponivel Magazine2 )
              (Localizado bloco1 mi1 )
              (ultimo-da-fila bloco1 )
              (Localizado bloco2 mi2 )
              (Localizado bloco3 mi3 )
              (Localizado bloco4 mi4 )
)

(:goal       (and  (Localizado bloco1 mf4 )
                   (Localizado bloco2 mf3 )
                   (Localizado bloco3 mf2 )
                   (Localizado bloco4 mf1 )
              )
))
```

APÊNDICE D – Testes Realizados

Esta seção mostrará apenas alguns dos testes realizados, pois pelas combinações de testes possíveis, seria inviável e desnecessária a apresentação de todos eles. Dos testes realizados mostrados, alguns serão apresentados em forma de tabela, contendo informação de número de blocos, tempo de solução e agentes utilizados, enquanto outros testes, considerados didáticos, serão mostrados com o respectivo plano gerado.

Uma vez que o Agente Mesa (que controla as mesas de entrada e saída de blocos) é essencial até para a mais simples das soluções, ele não constará nestas tabelas, pois este fato está implícito em todos os planos. Os demais agentes serão apresentados na forma abreviada na primeira linha de cada coluna sendo os Agentes Garra1, 2, 3 e 4, Esteira 1 e 2 e Magazines 1, 2, 3 e 4 representados respectivamente por: G1, G2, G3, G4, E1, E2, M1, M2, M3 E M4.

Todos os testes de planejamento foram realizados em um PC com processador Intel Pentium® 4com processador de 3,00 GHz e 2GB de memória RAM com sistema operacional Windows® XP Professional SP2.

Os testes serão divididos em 3 partes, separadas por nível de dificuldade com relação à ordem de entrada dos *n* blocos, que deverão apresentar uma ordem decrescente na saída (*n*, *n*-1, ..., 3, 2, 1). No item D.1 serão mostrado os testes de maior dificuldade, ou seja ordem crescente na entrada (1, 2, 3, ..., *n*-1, *n*); no item D.2 serão mostrados os testes com entrada randômica; por fim, no item D.3 serão mostrados alguns testes com menor dificuldade, ou seja, ordem decrescente, mas com uma importante relevância para corroborar a vantagem do uso econômico de agentes.

Apêndice D.1 – Ordem crescente de entrada dos blocos.

Para esta parte de testes, o valor mínimo de blocos que será usados no problema será 1 e o valor máximo será 7. A justificativa para este limite superior de blocos está no fato de que a memória do computador é esgotada antes que o problema consiga encontrar uma solução dentro do espaço de estados até então verificado *(Esta é uma limitação do planejador, uma vez que alguns planejadores consomem uma quantidade fixa de memória RAM, seja pela utilização de diferentes heurísticas ou pela utilização de memória paginada)*. Pode-se notar que matematicamente, para esta ordem de entrada de blocos, o número máximo de blocos que o sistema conseguiria ordenar seria 9.

A tabela Tabela 3 mostra os testes de planejamento até 7 blocos na entrada, contendo o número de passos da solução, tempo de planejamento da solução (T1) em milisegundos, tempo de busca de agentes e planejamento da solução, ou tempo total, (T2) em segundos e os agentes utilizados em cada planejamento.

Tabela 3 – Testes para ordem crescente de entrada de blocos

Nº de blocos	Passos	T1(ms)	T2(s)	G1	G2	G3	G4	E1	E2	M1	M2	M3	M4
1	5	31	0,145	X	X			X					
2	14	31	0,145	X	X			X		X			
3	21	46	0,266	X	X	X		X		X			
4	29	125	0,563	X	X	X		X		X	X		
5	37	1468	6,091	X	X	X		X	X	X	X		
6	44	67546	93,968	X	X	X		X	X	X	X		
7	51	132546	1255,047	X	X	X	X	X	X		X		X

Cada plano pode ser interrompido em qualquer passo para uma tentativa de replanejamento, porém é importante ressaltar que nem todos os casos permitem a continuação do plano:

- Casos onde os recursos e agentes disponíveis estão sendo utilizados em seus limites, ou seja, a retirada de qualquer um impossibilita a

realização do plano, e nenhum acréscimo pode ser feito não conseguirão encontrar uma nova solução.

- Casos onde determinada classe de agente é essencialmente necessária, e não possui nenhum representante no conjunto de agentes. Ex.: Tentativa de planejamento desde o começo sem nenhuma esteira transportadora.

- Casos onde o agente retirado anula consigo um dos objetivos. Ex.: Se uma garra está segurando determinado bloco que precisa estar na mesa final, e esta garra é removida do sistema, o replanejamento não encontra uma nova solução uma vez que a localização do bloco se perdeu com a retirada da garra.

Como mencionado anteriormente a realização de todas as possibilidades de parada, retirada de agentes e planejamento é inviável. Com isso será realizado um acompanhamento de caso seguindo o Exemplo 6 do Capítulo 7. Supondo que uma parada ocorra no mesmo passo nº 6 e os mesmos agentes (Garra 2 e Magazines) são retirados. No Exemplo 7 durante a parada eram inseridos os agentes Esteira 2 e Garra 4. Com a finalidade de encontrar uma solução com um diferente conjunto de agentes, a Garra 3 é também retirada do sistema e o Magazine 4 continua ativo ao contrário do Exemplo 7. Além disse

O novo conjunto de agentes e recursos utilizados é:

- Mesas Inicial e Final;
- Robôs: 1, e 4;
- Esteiras: 1 e 2;
-Magazine 4;

Os passos da nova solução são mostrados a seguir:

```
 0: PEGAR_G1 BLOCO2 MI4
 1: PEGAR_G4 BLOCO4 ESTEIRA12
 2: SOLTAR_G1 BLOCO3 ESTEIRA21
 3: ANDARFILA MESAINICIAL MI3 MI4
 4: TRANSPORTARBLOCOESTEIRA2 BLOCO3 ESTEIRA21 ESTEIRA22
 5: PEGAR_G1 BLOCO1 MI4
 6: SOLTAR_G1 BLOCO1 ESTEIRA11
 7: TRANSPORTARBLOCOESTEIRA1 BLOCO1 ESTEIRA11 ESTEIRA12
 8: SOLTAR_G1 BLOCO2 ESTEIRA11
 9: SOLTAR_G4 BLOCO4 MAGAZINE4
10: PEGAR_G4 BLOCO1 ESTEIRA12
11: SOLTAR_G4 BLOCO1 MF1
12: TRANSPORTARBLOCOESTEIRA1 BLOCO2 ESTEIRA11 ESTEIRA12
13: ANDARFILA MESAFINAL MF1 MF2
14: PEGAR_G4 BLOCO2 ESTEIRA12
15: SOLTAR_G4 BLOCO2 MF1
16: ANDARFILA MESAFINAL MF2 MF3
17: PEGAR_G4 BLOCO3 ESTEIRA22
18: SOLTAR_G4 BLOCO3 MF1
19: ANDARFILA MESAFINAL MF3 MF4
20: PEGAR_G4 BLOCO4 MAGAZINE4
21: SOLTAR_G4 BLOCO4 MF1
```

Exemplo 17 – Replanejamento do Exemplo 7

Com este exemplo foi montada a Tabela 4 mostrando o conjunto de agentes alocados inicialmente, e outros dois conjuntos diferentes de agentes utilizados a partir do sexto passo.

Tabela 4 – Diferentes conjuntos de Agentes para a resolução de um mesmo problema

Exemplo nº	**passo nº**	**G1**	**G2**	**G3**	**G4**	**E1**	**E2**	**M1**	**M2**	**M3**	**M4**
7	0	X	X	X		X		X	X		
8	6	X		X	X	X	X				
18	6	X			X	X	X				X

Apêndice D.1 – Ordem aleatória de entrada dos blocos.

Nesta seção serão mostrados os testes realizados para uma entrada de blocos de forma aleatória, com o número de blocos de entrada variando de 3 a 6. Os testes nesta parte serão apresentados apenas na **Tabela 5**.

Tabela 5 – Testes para ordem aleatória de entrada de blocos

nº blocos	Ordem	passos	T1(ms)	T2(s)	G1	G2	G3	G4	E1	E2	M1	M2	M3	M4
3	1 3 2	21	46	0,287	X	X	X		X		X			
3	2 1 3	23	46	0,293	X	X			X		X			
4	2 3 4 1	28	140	0,481	X	X	X		X		X	X		
4	4 2 3 1	26	62	0,546	X	X	X		X		X			
4	2 4 1 3	28	125	0,328	X	X	X		X		X			
5	2 1 4 3 5	37	796	2,671	X	X	X		X		X	X		
5	3 5 1 4 2	37	1156	1,578	X	X	X		X		X			
5	3 5 1 2 4	35	140	4,968	X	X	X		X	X	-	X		
5	5 3 2 1 4	35	109	1,953	X	X	X		X	X		X		
6	6 3 5 1 4 2	44	11390	55,420	X	X	X		X	X		X		
6	4 5 3 1 2 6	44	6203	8,046	X	X	X		X		X			
6	3 1 4 6 5 2	42	1621	53,985	X	X	X		X	X		X		
6	2 3 1 6 5 4	42	1859	36,575	X	X	X		X	X		X		
6	6 4 5 3 2 1	40	5718	10,163	X	X	X		X		X			

Apêndice D.3 – Ordem decrescente de entrada dos blocos.

Nesta seção serão mostrados os planos gerados para o problema de blocos de entrada em ordem decrescente. Como mostrado na Tabela 6, o conjunto de agentes para qualquer quantidade de blocos será sempre o mesmo para o estado inicial, variando apenas o número de passos. Os planos foram gerados para até um número máximo de 19 blocos.

Tabela 6 – Testes para ordem decrescente de entrada de blocos

nblocos	passos	T1(ms)	T2(s)	G1	G2	E1
1	5	31	0,312	X	X	X
2	12	46	0,14	X	X	X
3	19	31	0,156	X	X	X
4	26	31	0,16	X	X	X
5	33	31	0,343	X	X	X
6	40	62	0,859	X	X	X
7	47	78	0,281	X	X	X
8	54	140	0,343	X	X	X
9	61	218	0,484	X	X	X
10	67	359	0,718	X	X	X
11	75	593	1,203	X	X	X
12	82	968	1,968	X	X	X
13	89	1500	2,75	X	X	X
14	96	2359	4,203	X	X	X
15	103	3218	6,046	X	X	X
16	110	4687	8,578	X	X	X
17	117	6640	12,015	X	X	X
18	124	9109	16,796	X	X	X
19	131	12828	22,828	X	X	X

Os planos mostrados a seguir têm por principal objetivo enfatizar o problema da alocação desnecessária de agentes. O Exemplo 18 mostra um plano gerado para o problema da entrada de 10 blocos em ordem decrescente utilizando o APDI.

O conjunto de agentes e recursos utilizados é:

- Mesas: Inicial e Final;
- Robôs: 1 e 2;
- Esteiras: 1;

Os passos da solução são mostrados a seguir:

```
 0: PEGAR_G1 BLOCO1 MI10
 1: SOLTAR_G1 BLOCO1 ESTEIRA11
 2: TRANSPORTARBLOCOESTEIRA1 BLOCO1 ESTEIRA11 ESTEIRA12
 3: PEGAR_G2 BLOCO1 ESTEIRA12
 4: SOLTAR_G2 BLOCO1 MF1
 5: ANDARFILA MESAFINAL MF1 MF2
 6: ANDARFILA MESAINICIAL MI9 MI10
 7: PEGAR_G1 BLOCO2 MI10
 8: SOLTAR_G1 BLOCO2 ESTEIRA11
 9: TRANSPORTARBLOCOESTEIRA1 BLOCO2 ESTEIRA11 ESTEIRA12
10: PEGAR_G2 BLOCO2 ESTEIRA12
11: SOLTAR_G2 BLOCO2 MF1
12: ANDARFILA MESAFINAL MF2 MF3
13: ANDARFILA MESAINICIAL MI9 MI10
14: PEGAR_G1 BLOCO3 MI10
15: SOLTAR_G1 BLOCO3 ESTEIRA11
16: TRANSPORTARBLOCOESTEIRA1 BLOCO3 ESTEIRA11 ESTEIRA12
17: PEGAR_G2 BLOCO3 ESTEIRA12
18: SOLTAR_G2 BLOCO3 MF1
19: ANDARFILA MESAFINAL MF3 MF4
20: ANDARFILA MESAINICIAL MI9 MI10
21: PEGAR_G1 BLOCO4 MI10
22: SOLTAR_G1 BLOCO4 ESTEIRA11
23: TRANSPORTARBLOCOESTEIRA1 BLOCO4 ESTEIRA11 ESTEIRA12
24: PEGAR_G2 BLOCO4 ESTEIRA12
25: SOLTAR_G2 BLOCO4 MF1
26: ANDARFILA MESAFINAL MF4 MF5
27: ANDARFILA MESAINICIAL MI9 MI10
28: PEGAR_G1 BLOCO5 MI10
29: SOLTAR_G1 BLOCO5 ESTEIRA11
30: TRANSPORTARBLOCOESTEIRA1 BLOCO5 ESTEIRA11 ESTEIRA12
31: PEGAR_G2 BLOCO5 ESTEIRA12
32: SOLTAR_G2 BLOCO5 MF1
33: ANDARFILA MESAFINAL MF5 MF6
34: ANDARFILA MESAINICIAL MI9 MI10
35: PEGAR_G1 BLOCO6 MI10
36: SOLTAR_G1 BLOCO6 ESTEIRA11
37: ANDARFILA MESAINICIAL MI9 MI10
38: PEGAR_G1 BLOCO7 MI10
39: ANDARFILA MESAINICIAL MI9 MI10
40: TRANSPORTARBLOCOESTEIRA1 BLOCO6 ESTEIRA11 ESTEIRA12
41: SOLTAR_G1 BLOCO7 ESTEIRA11
42: PEGAR_G1 BLOCO8 MI10
43: ANDARFILA MESAINICIAL MI9 MI10
44: PEGAR_G2 BLOCO6 ESTEIRA12
45: SOLTAR_G2 BLOCO6 MF1
46: ANDARFILA MESAFINAL MF6 MF7
47: TRANSPORTARBLOCOESTEIRA1 BLOCO7 ESTEIRA11 ESTEIRA12
48: SOLTAR_G1 BLOCO8 ESTEIRA11
49: PEGAR_G1 BLOCO9 MI10
50: ANDARFILA MESAINICIAL MI9 MI10
51: PEGAR_G2 BLOCO7 ESTEIRA12
52: TRANSPORTARBLOCOESTEIRA1 BLOCO8 ESTEIRA11 ESTEIRA12
53: SOLTAR_G1 BLOCO9 ESTEIRA11
```

```
54: PEGAR_G1 BLOCO10 MI10
55: SOLTAR_G2 BLOCO7 MF1
56: ANDARFILA MESAFINAL MF7 MF8
57: PEGAR_G2 BLOCO8 ESTEIRA12
58: TRANSPORTARBLOCOESTEIRA1 BLOCO9 ESTEIRA11 ESTEIRA12
59: SOLTAR_G1 BLOCO10 ESTEIRA11
60: SOLTAR_G2 BLOCO8 MF1
61: PEGAR_G2 BLOCO9 ESTEIRA12
62: TRANSPORTARBLOCOESTEIRA1 BLOCO10 ESTEIRA11 ESTEIRA12
63: ANDARFILA MESAFINAL MF8 MF9
64: SOLTAR_G2 BLOCO9 MF1
65: PEGAR_G2 BLOCO10 ESTEIRA12
66: ANDARFILA MESAFINAL MF9 MF10
67: SOLTAR_G2 BLOCO10 MF1
```

Exemplo 18 – Plano gerado usando APDI

O Exemplo 19, mostrado a seguir, é um plano gerado pelo planejador FF com todos os agentes e recursos formando um único domínio.

O conjunto de agentes e recursos utilizados é:

- Mesas: Inicial e Final;
- Robôs: 1, 2, 3 e 4;
- Esteiras: 1 e 2;

Os passos da solução são mostrados a seguir:

```
 0: PEGAR_G1 BLOCO1 MI10
 1: COLOCAR_G1 BLOCO1 ESTEIRA21
 2: MOVERBLOCOESTEIRA2 BLOCO1 ESTEIRA21 ESTEIRA22
 3: PEGAR_G2 BLOCO1 ESTEIRA22
 4: COLOCAR_G2 BLOCO1 MF1
 5: ANDARFILA MESAFINAL
 6: ANDARFILA MESAINICIAL
 7: PEGAR_G1 BLOCO2 MI10
 8: COLOCAR_G1 BLOCO2 ESTEIRA21
 9: MOVERBLOCOESTEIRA2 BLOCO2 ESTEIRA21 ESTEIRA22
10: PEGAR_G2 BLOCO2 ESTEIRA22
11: COLOCAR_G2 BLOCO2 MF1
12: ANDARFILA MESAINICIAL
13: PEGAR_G1 BLOCO3 MI10
14: COLOCAR_G1 BLOCO3 ESTEIRA21
15: MOVERBLOCOESTEIRA2 BLOCO3 ESTEIRA21 ESTEIRA22
16: PEGAR_G2 BLOCO3 ESTEIRA22
17: ANDARFILA MESAINICIAL
18: PEGAR_G1 BLOCO4 MI10
19: COLOCAR_G1 BLOCO4 ESTEIRA21
20: MOVERBLOCOESTEIRA2 BLOCO4 ESTEIRA21 ESTEIRA22
21: PEGAR_G4 BLOCO4 ESTEIRA22
22: ANDARFILA MESAINICIAL
```

```
23: PEGAR_G1 BLOCO5 MI10
24: COLOCAR_G1 BLOCO5 ESTEIRA21
25: MOVERBLOCOESTEIRA2 BLOCO5 ESTEIRA21 ESTEIRA22
26: ANDARFILA MESAINICIAL
27: PEGAR_G1 BLOCO6 MI10
28: COLOCAR_G1 BLOCO6 ESTEIRA11
29: MOVERBLOCOESTEIRA1 BLOCO6 ESTEIRA11 ESTEIRA12
30: ANDARFILA MESAINICIAL
31: PEGAR_G1 BLOCO7 MI10
32: COLOCAR_G1 BLOCO7 ESTEIRA21
33: ANDARFILA MESAFINAL
34: COLOCAR_G2 BLOCO3 MF1
35: PEGAR_G2 BLOCO5 ESTEIRA22
36: MOVERBLOCOESTEIRA2 BLOCO7 ESTEIRA21 ESTEIRA22
37: ANDARFILA MESAINICIAL
38: PEGAR_G1 BLOCO8 MI10
39: COLOCAR_G1 BLOCO8 ESTEIRA21
40: ANDARFILA MESAINICIAL
41: PEGAR_G1 BLOCO9 MI10
42: ANDARFILA MESAINICIAL
43: PEGAR_G3 BLOCO10 MI10
44: COLOCAR_G1 BLOCO9 ESTEIRA11
45: ANDARFILA MESAFINAL
46: COLOCAR_G4 BLOCO4 MF1
47: ANDARFILA MESAFINAL
48: COLOCAR_G2 BLOCO5 MF1
49: PEGAR_G2 BLOCO6 ESTEIRA12
50: PEGAR_G4 BLOCO7 ESTEIRA22
51: MOVERBLOCOESTEIRA2 BLOCO8 ESTEIRA21 ESTEIRA22
52: MOVERBLOCOESTEIRA1 BLOCO9 ESTEIRA11 ESTEIRA12
53: COLOCAR_G3 BLOCO10 ESTEIRA21
54: ANDARFILA MESAFINAL
55: COLOCAR_G2 BLOCO6 MF1
56: PEGAR_G2 BLOCO8 ESTEIRA22
57: MOVERBLOCOESTEIRA2 BLOCO10 ESTEIRA21 ESTEIRA22
58: ANDARFILA MESAFINAL
59: COLOCAR_G4 BLOCO7 MF1
60: PEGAR_G4 BLOCO9 ESTEIRA12
61: ANDARFILA MESAFINAL
62: COLOCAR_G2 BLOCO8 MF1
63: PEGAR_G2 BLOCO10 ESTEIRA22
64: ANDARFILA MESAFINAL
65: COLOCAR_G4 BLOCO9 MF1
66: ANDARFILA MESAFINAL
67: COLOCAR_G2 BLOCO10 MF1
```

Exemplo 19 – Plano gerado sem APDI

Com este exemplo pôde-se constatar que a solução foi encontrada com o mesmo número de passos, uma vez que é o mais trivial dos problemas deste domínio. Porém, sem a ajuda do sistema na escolha dos agentes, o número de agentes alocados para a realização das tarefas foi o dobro, desconsiderando o agente Mesa. Os agentes sobressalentes alocados foram:

- Robôs: 3 e 4;
- Esteiras: 2;

Em um domínio onde agentes não utilizados podem ser alocados para outras tarefas, ou onde o custo de aplicação de diferentes agentes é elevado, o sistema com APDI pode ser de grande utilidade.