TIAGO STEGUN VAQUERO

ITSIMPLE: AMBIENTE INTEGRADO DE MODELAGEM E ANÁLISE DE DOMÍNIOS DE PLANEJAMENTO AUTOMÁTICO

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

São Paulo
2007

TIAGO STEGUN VAQUERO

ITSIMPLE: AMBIENTE INTEGRADO DE MODELAGEM E ANÁLISE DE DOMÍNIOS DE PLANEJAMENTO AUTOMÁTICO

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de Concentração:
Eng. Mecatrônica e Sistemas Mecânicos

Orientador:
Prof. Dr. José Reinaldo Silva

São Paulo
2007

FICHA CATALOGRÁFICA

Vaquero, Tiago Stegun
itSIMPLE : ambiente integrado de modelagem e análise de domínios de planejamento automático / T.S. Vaquero. – ed.rev. -- São Paulo, 2007.
284 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1. Engenharia do conhecimento 2. Inteligência artificial 3. Modelagem de dados I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos II. t.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela vida e por colocar pessoas tão especiais em meu caminho. Sem Ele nada disso seria possível.

Aos meus pais pelo suporte, carinho e pela minha formação.

A minha querida noiva Patrícia pelo amor, carinho, dedicação, paixão e paciência, principalmente nos momentos mais difíceis. Sem ela não existiria tanta motivação.

Ao meu orientador e professor José Reinaldo pelo apoio, pelos conselhos e direcionamentos que conduziram a um trabalho de intenso aprendizado.

Aos colegas do laboratório Design Lab, particularmente ao Eston, Daniel, Pedro e aos alunos Victor e Fernando pela imensa ajuda e companheirismo.

Aos colegas do laboratório IAAA da FEI pelo suporte, incentivo, colaboração e também pela grande ajuda.

RESUMO

O grande avanço das técnicas de Planejamento em Inteligência Artificial fez com que a Engenharia de Requisitos e a Engenharia do Conhecimento ganhassem extrema importância entre as disciplinas relacionadas a projeto de engenharia (*Engineering Design*). A especificação, modelagem e análise dos domínios de planejamento automático se tornam etapas fundamentais para melhor entender e classificar os domínios de planejamento, servindo também de guia na busca de soluções. Neste trabalho, é apresentada uma proposta de um ambiente integrado de modelagem e análise de domínios de planejamento, que leva em consideração o ciclo de vida de projeto, representado por uma ferramenta gráfica de modelagem que utiliza diferentes representações: a UML para modelar e analisar as características estáticas dos domínios; XML para armazenar, integrar, e exportar informação para outras linguagens (ex.: PDDL); as Redes de Petri para fazer a análise dinâmica; e a PDDL para testes com planejadores.

Palavras-chave: Planejamento Automático, Engenharia do Conhecimento, Engenharia de Requisitos, Modelagem e Análise de Domínio, PDDL, UML, Redes de Petri, XML.

ABSTRACT

The great development in Artificial Intelligence Planning has emphasized the role of Requirements Engineering and Knowledge Engineering among the disciplines that contributes to Engineering Design. The modeling and specification of automated planning domains turn out to be fundamental tasks in order to understand and classify planning domains and guide the application of problem solving techniques. In this work, it is presented the proposed integrated environment for modeling and analyzing automated planning domains, which considered the life cycle of a project, represented by a tool that uses several language representations: UML to model and perform static analyses of planning environments; XML to hold, integrate, share and export information to other language representations (e.g. PDDL); Petri Nets, where dynamic analyses are made; and PDDL for testing models with planners.

Key-words: Automated Planning, Knowledge Engineering, Requirements Engineering, Domain Modeling and Analysis, PDDL, UML, Petri Nets, XML.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

1. Introdução

A recente melhoria na eficiência e no aumento da demanda de sistemas de planejamento automático tem gerado uma grande expectativa com respeito à aplicação dos desenvolvimentos já alcançados em aplicações reais e complexas da engenharia. A área de pesquisa de Planejamento Automático permaneceu por muitos anos focada em resolver problemas clássicos, e, durante este período, diversas técnicas e métodos capazes de resolver problemas interessantes foram desenvolvidos. A partir do final dos anos 90 gerou-se uma grande motivação no uso das técnicas de Planejamento Automático para resolver problemas reais tais como, controle de redes de satélites (KHATIB et al., 2003), movimentação de elevadores inteligentes em grandes edifícios (KOEHLER; SCHUSTER, 2000), robôs móveis de uso doméstico, manufatura flexível, controle e otimização da produção, entre outros. Para estes problemas chamados reais, uma grande barreira se encontra na especificação dos requisitos, modelagem (principalmente na modelagem de ações) e análise de domínios, já que estes processos são considerados um grande "gargalo" no desenvolvimento de aplicações reais. O gargalo se deve a muitos fatores como: incertezas e erros que podem ocorrer em função da má comunicação ou falta de envolvimento de especialistas nas informações do domínio que está sendo analisado; membros da equipe podem apresentar vocabulário conflitante, ambíguo, e má interpretação dos conceitos e funcionalidades envolvidos, em função do uso de linguagem natural não precisa; a própria dificuldade do usuário em explicitar de forma completa, clara e estável seus requisitos.

A qualidade na realização dos processos iniciais de design, bem como o material gerado, é fundamental para o sucesso no decorrer do desenvolvimento de aplicações complexas e para a manutenção de sua viabilidade. Para ressaltar a importância das fases iniciais, Conrow e Shishido (1997) apontam resultados de uma pesquisa sobre 8.380 projetos de software comerciais, indicando que: 53% apresentavam sérios problemas de orçamento, atrasos ou tiveram poucas características e funcionalidades em relação ao requisito e 31% desses projetos foram cancelados. Em média, esses projetos obtiveram aumentos de 189% e 222% sobre os custos e o cronograma inicial, respectivamente, no momento em que foram concluídos ou cancelados. Entre os projetos concluídos apenas uma média de 61% dos requisitos inicialmente esperados (características e funcionalidades) foram satisfeitos. A maior parte dos erros dos sistemas desenvolvidos estão associados às fases de análise de requisitos e design, e só são descobertos em etapas posteriores (mais avançadas) como implementação e testes. O custo para correção de um erro ainda durante a análise equivale a 1/5 do que seria durante a fase de testes e a 1/15 depois que o sistema estivesse em uso (KOTONYA; SOMMERVILLE, 1996).

Todas as dificuldades nas fases de design estão presentes também para os projetos de Planejamento Automático. De fato, se um domínio de planejamento é bem entendido, especificado, modelado e analisado, além de caminhar o projeto ao sucesso, as técnicas de planejamento podem ser melhor utilizadas, desenvolvidas e direcionadas aos tipos de domínios onde se mostram mais eficientes, pois estas se encontram em constante evolução (MCCLUSKEY, 2002) (MCCLUSKEY et al., 2003).

Neste cenário, métodos, boas práticas, ferramentas e conceitos provenientes das Engenharia do Conhecimento e de Requisitos se tornam fundamentais para melhor caracterizar, entender e especificar domínios de planejamento e, deste modo, contribuir para o avanço dos sistemas e técnicas existentes que objetivam problemas mais complexos e

aplicações reais da indústria e de negócios. Este fato tem sido evidenciado em diversas iniciativas provenientes dos próprios desenvolvedores e pesquisadores em Planejamento Automático. Iniciativas como a primeira Competição Internacional de Engenharia do Conhecimento para Planejamento e Escalonamento (ICKEPS - *International Competition on Knowledge Engineering for Planning and Scheduling*), ocorrida em 2005, enfatizaram as necessidades de métodos e ferramentas de suporte a especificação e design de sistemas de planejamento, já que problemas reais envolvem desafios maiores, onde a contribuição de outras engenharias, tais como Engenharia de Software, de Requisitos, do Conhecimento, entre outras, faz-se necessária.

A modelagem de domínios e problemas de planejamento, realizada pela maioria dos pesquisadores da comunidade de pesquisa de Planejamento Automático, é efetuada utilizando uma linguagem padrão conhecida como PDDL - *Planning Domain Definition Language* (MCDERMOTT, 1998). Esta é a linguagem mais utilizada hoje nos sistemas de planejamento para tratar e representar os modelos dos domínios (incluindo os domínios ditos “reais”). Todavia, a PDDL tem se mostrado uma linguagem que deixa muito a desejar no processo de modelagem devido a suas restrições e limitações (BACHHUS 2003; BODDY 2003; GEFFNER 2003; MCCALLUM 2003; MCCLUSKEY 2003; MCDERMOTT 2003; SMITH 2003). Esta linguagem não foi projetada para contemplar os processos envolvidos nas Engenharias do Conhecimento e de Requisitos, e sim para ser entendida e interpretada pelos sistemas e técnicas de planejamento (algoritmos). De fato, acredita-se que uma das razões da existência de uma lacuna a ser preenchida entre os sistemas de planejamento automático e as aplicações reais seja proveniente da limitação da linguagem de modelagem e da negligência de aspectos presentes nos ciclos de vida de projetos de sistemas em geral (tais como entendimento do problema, análise de requisitos, especificação entre outros).

Devido a grande necessidade de métodos e ferramentas que auxiliem os projetistas durante o ciclo de vida de projeto de domínios de planejamento, este trabalho propõe um ambiente integrado de modelagem e análise de domínios de planejamento que é concebido através de uma ferramenta (ambiente) gráfica integrada que utiliza: a UML - *Unified Modeling Language* (OMG, 2001) - como uma linguagem de modelagem de domínios de planejamento enfatizando os processos de análise de requisitos, especificação e a análise estática; a Rede de Petri (MURATA, 1989) como um formalismo para análise dos aspectos dinâmicos do modelo do domínio; a PDDL como uma linguagem de comunicação com os planejadores para eventuais testes dos modelos; e a XML - *eXtensible Markup Language* (BRAY et al., 2004) - como uma meta-linguagem (intermediária) por ser compatível com outras linguagens (incluindo a UML, PDDL e Rede de Petri). Esta ferramenta, chamada itSIMPLE (*Integrated Tool Software Interface for Modeling PLanning Environment*), tem por objetivo auxiliar o projetista e sua equipe a realizar as fases iniciais de design de domínios de planejamento automático em um ambiente integrado utilizando linguagens bem conhecidas, tanto pela comunidade de pesquisa quanto pela industria e negócios, contribuindo assim para a evolução das técnicas de planejamento existentes de modo a serem aplicadas em sistemas reais de planejamento.

1.1. Motivações

Desde o início, na década de 60, a área de pesquisa de Planejamento Automático em Inteligência Artificial, focou grandes esforços apenas no desenvolvimento de técnicas (algoritmos) de planejamento (planejadores), reconhecidas como solucionadores gerais de problemas. Durante muito tempo, os problemas solucionados por esses planejadores eram restritos a domínios clássicos de planejamento, ou seja, os modelos de domínios eram bem conhecidos e simplificados.

A partir da década de 90, a comunidade de pesquisa desta área passou a demonstrar um grande interesse em aplicar as técnicas de planejamento a problemas reais de engenharia onde a complexidade se torna uma grande barreira. Muitas dificuldades foram encontradas, já que as técnicas utilizadas obtinham ótimas performances apenas em problemas clássicos, mas estas dificuldades deram grande força ao desenvolvimento de novas técnicas que aos poucos se aproximavam do desejado.

Um ponto importante no desenvolvimento destas técnicas foi, por muito tempo, negligenciado embora fosse um dos aspectos fundamentais para o progresso do Planejamento Automático: “Não importa o quão eficientes ou poderosas são as técnicas e mecanismos de planejamento, elas são tão boas quanto o conhecimento do domínio que é fornecido a elas. Se o modelo do domínio fornecido é falho, o resultado da aplicação das técnicas será também falho” (MCCLUSKEY, 2003b). Em anos recentes este ponto vem sendo muito discutido pela comunidade e foi assim que áreas como a Engenharia do Conhecimento e de Requisitos se tornaram áreas relevantes à pesquisa nesta área da Inteligência Artificial. Assim, os estudos sobre métodos, técnicas, ambientes e ferramentas de suporte ao processo de especificação e design dos domínios de planejamento se tornaram extremamente necessários.

Neste cenário o presente trabalho coloca foco no estudo e desenvolvimento de ambientes ferramentas na área da Engenharia do Conhecimento aplicadas ao Planejamento Automático.

1.2. Objetivo e Justificativa

O objetivo desta proposta é o estudo e o desenvolvimento um ambiente, uma ferramenta (um software, independente de qualquer técnica de planejamento), de suporte aos processos iniciais de projeto, com uma abordagem orientada a objetos, de análise de requisitos, especificação, modelagem e análise de domínios que estão presentes no ciclo de vida de projetos que focalizam na utilização do conceito de Planejamento Automático.

Esse ambiente é implementado em Java™ e é caracterizado por uma ferramenta integrada que utiliza: UML (*Unified Modeling Language*) (OMG, 2001) com uma abordagem de planejamento para as fases de análise de requisitos, especificação e modelagem (principalmente os aspectos estáticos); Redes de Petri para a análise de requisitos e análise dinâmica do modelo do domínio; a linguagem padrão em planejamento, PDDL (*Planning Domain Definition Language*) (MCDERMOTT, 1998), como uma linguagem intermediária entre a ferramenta e as técnicas de planejamento existentes; a XML (*eXtensible Markup Language*) (BRAY et al., 2004) como uma linguagem de armazenamento e representação dos modelos, assim como uma linguagem que integra e descreve todas as outras, como por exemplo, a PNML (*Petri Nets Markup Language*) (WEBER; KINDLER, 2002) que descreve o formalismo de Redes de Petri em XML e a XPDDL (*eXtensible Planning Domain Definition Language*) (GOUGH, 2006) que nada mais é que a representação da PDDL em XML.

Devido ao fato da Engenharia do Conhecimento, aplicada ao Planejamento Automático, ser uma área relativamente nova, tanto no cenário nacional quanto no internacional, existem poucas ferramentas disponíveis direcionas ao apoio das fases iniciais de projeto de um domínio de planejamento. Para que seja possível tratar domínios reais são, geralmente, necessárias ambientes e ferramentas que auxiliem o projetista a estruturar e organizar o processo de design. Este fato é verdade tanto para o desenvolvimento de softwares ou sistemas de informação quanto em problemas de planejamento e, infelizmente, pouca atenção é dada para o desenvolvimento de tais ferramentas na área de Planejamento Automático.

Como a pesquisa em Planejamento Automático focalizou por muitos anos em problemas clássicos, o desenvolvimento de ferramentas que contribuíssem para o processo de design não se mostrava necessário até então. Por muitos anos os pesquisadores utilizaram a linguagem padrão de definição de domínio de planejamento chamada PDDL. De fato, esta linguagem foi

projetada para ser utilizada como meio de comparação de performance entre as técnicas de planejamento, nas competições que ocorriam nos principais congressos da área, e não para ser uma linguagem de especificação e modelagem de domínios reais. A linguagem estava mais próxima de uma linguagem de programação acadêmica do que uma linguagem de modelagem ou especificação. Com isso, os processos de modelagem e análise de domínios se tornavam tarefas árduas quando os modelos eram representados em PDDL, de fato, apenas os especialistas em planejamento os entendiam, dificultando assim uma visão geral do modelo, e inviabilizando qualquer tentativa de reutilização.

Representações esquemáticas são mais apropriadas e mais utilizadas para as fases iniciais de projeto tais como, UML e Redes de Petri. A UML é uma linguagem muito conhecida na Engenharia de Software por contribuir nas fases de especificação e modelagem de domínios, favorecendo a uniformidade dos conceitos do modelo perante os participantes do projeto devido ao uso de diagrama. Esta linguagem vem demonstrando um grande potencial na modelagem das características estáticas e dinâmicas de domínios com abordagens orientadas a objetos. Já a Rede de Petri é um formalismo matemático muito poderoso e útil para análises dinâmica de domínios assim como para análises de requisitos.

O uso da XML em ferramentas de modelagem passou a ser, nos dias de hoje, um senso comum. A XML permite a transição de informação entre diversas plataformas e ferramentas possibilitando que um mesmo modelo seja visualizado e analisado em diversos sistemas. Sua flexibilidade de representação possibilita o intercâmbio entre linguagens sem grandes implementações. Diversos formalismos e linguagens podem ser representados em XML, como é o caso da PNML (WEBER; KINDLER, 2002) que representa modelos em Redes de Petri e da XPDDL (GOUGH, 2004) que representa modelos em PDDL.

A ferramenta proposta neste trabalho integra todas essas linguagens fazendo com que o projetista usufrua os potenciais de todas elas buscando um modelo de domínio de planejamento correto.

1.3. Contribuições

i. Revisão sobre Planejamento Automático e seu histórico. Área de pesquisa pouco difundida no cenário nacional acadêmico apesar da crescente evidência no cenário internacional.

ii. Revisão sobre a recente área de Engenharia do Conhecimento aplicada a Planejamento Automático – uma área de pesquisa mundialmente nova. Um histórico da evolução desta área é apresentado, bem como suas principais características.

iii. Estudos sobre o processo de especificação e modelagem de domínios de planejamento mais realísticos.

iv. Especificação, modelagem e análise de domínios de planejamento, utilizando linguagens bem difundido na modelagem de sistemas, tais como UML, XML, Redes de Petri, e PDDL. A discussão e a utilização dessas linguagens provêem um ponto de partida no processo de modelagem e análise de domínios reais de planejamento automático. Provê também um ambiente integrado inicial (protótipo) desenvolvido para que os pesquisadores interessados no projeto possam agregar novas funcionalidades.

v. Um protótipo de uma ferramenta integrada com os principais atributos do ambiente integrado proposto de auxílio no processo de análise de requisitos, especificação, modelagem e análise de domínios de planejamento.

vi. Ferramentas e métodos para evoluções das áreas de Planejamento Automático e a Engenharia do Conhecimento aplicada ao Planejamento Automático;

1.4. Estrutura da dissertação

Esta dissertação traz o levantamento geral sobre o Planejamento Automático e principalmente sobre as Engenharias do Conhecimento e de Requisitos aplicadas ao Planejamento Automático, áreas que são pouco difundidas no Brasil e algumas delas até no mundo. Para tal, o seu conteúdo foi organizado da seguinte maneira:

Capítulo 2: apresenta-se uma descrição dos principais conceitos envolvidos no processo de especificação, modelagem e análise de domínios de planejamento automático. Neste capítulo são apresentadas áreas como Planejamento Automático, a recente Engenharia do Conhecimento aplicada a Planejamento e Engenharia de Requisitos. As linguagens de especificação e modelagem de domínios, tanto de planejamento quanto de sistemas em geral, são descritas brevemente com o objetivo de levantar as necessidades de representação e de ferramentas que auxiliam no processo de entendimento, caracterização e modelagem de domínios que envolvam planejamento automático.

Capítulo 3: neste capítulo é apresentado o ambiente integrado proposto de modelagem e análise de domínios de planejamento proposto neste trabalho, que contempla o conceito de ciclo de vida de projetos, onde as fases são bem definidas. Os principais conceitos do ambiente são apresentados através de um domínio de planejamento clássico facilmente encontrado na literatura. Neste trabalho os esforços de pesquisa se concentram nas fases de modelagem e análise do domínio diante do cenário da pesquisa na área da Engenharia do Conhecimento para Planejamento.

Capítulo 4: apresenta-se uma ferramenta (um ambiente) de auxílio ao projetista e toda sua equipe durante as fases iniciais do ciclo de vida de projeto de planejamento. Esta ferramenta é

implementação do ambiente proposto no Capítulo 3. As características e funcionalidades da ferramenta são apresentadas mostrando os resultados já obtidos deste trabalho de pesquisa.

Capítulo 5: neste capítulo são apresentados estudos de casos que demonstram a utilização do ambiente (da ferramenta) na modelagem e análise de alguns domínios interessantes de planejamento. O capítulo apresenta três domínios modelados e analisados no itSIMPLE, sendo eles: o *Mundo de Blocos Estendido*; o *Logística Estendido*; e o domínio *Montagem Seqüencial de Carros em Linhas de Montagem*. Com os estudos de casos são apresentados alguns benefícios da utilização da ferramenta, bem como os conceitos do ambiente proposto.

Capítulo 6: apresenta-se a conclusão do trabalho e os trabalhos futuros.

Capítulo 2

2. Revisão Bibliográfica

Com o objetivo de analisar o cenário vigente na área de Planejamento Automático e contribuir com métodos e ferramentas de suporte ao processo de especificação, modelagem e análise de domínios de planejamento reais e complexos, é apresentada neste capítulo uma breve descrição das principais áreas, técnicas e linguagens associadas ao processo de design de sistemas, principalmente os de planejamento automático.

Inicialmente, como revisão da literatura, serão apresentados os principais conceitos da área de Planejamento Automático (*Automated Planning*) onde este trabalho está inserido. Em seguida será apresentada uma das mais recentes áreas dentro do Planejamento Automático, a Engenharia do Conhecimento para Planejamento. Esta área envolve diversos aspectos relacionados à especificação e modelagem de domínios de planejamento. Alguns desses aspectos são inerentes a Engenharia de Requisitos, área que também será apresentada neste capítulo. Após as descrições dessas áreas são apresentadas as linguagens utilizadas para modelar domínios. Dentre essas linguagens está presente uma das linguagens mais utilizadas em planejamento automático, a PDDL - *Planning Domain Definition Language* (MCDERMOTT, 1998). Finalmente, como muitos aspectos das fases iniciais de design não estão contemplados nesta linguagem, outras linguagens são apresentadas nesta proposta para complementar o processo de especificação, modelagem e análise de domínios de planejamento. Tais linguagens são: UML (*Unified Modeling Language*) (OMG, 2001), OCL (*Object Constraint Language*) (OMG, 2003), XML (*eXtensible Markup Language*) (BRAY et

al., 2004), Redes de Petri (MURATA, 1989) e PNML (*Petri Net Markup Language*) (WEBER; KINDLER, 2002) e XPDDL (*eXtensible Planning Domain Definition Language)* (GOUGH, 2004). Durante a exposição dos tópicos, breves históricos serão apresentados para situar o leitor no tempo e na necessidade levantada no presente trabalho.

2.1. Planejamento Automático – *Automated Planning*

2.1.1. Conceito

Planejar é o processo abstrato e deliberativo de escolha e organização de ações antecipando seus efeitos esperados. Esse processo tem como missão atingir, da melhor forma possível, os objetivos pré-estabelecidos. O Planejamento Automático é uma área da Inteligência Artificial (IA) que estuda este processo deliberativo computacionalmente.

Esta crescente área da IA está presente em cenários como: planejamento de trajetória e movimentação de sistemas automáticos móveis; planejamento de percepção envolvendo ações de sensoriamento para captação de informação do ambiente; planejamento de navegação que combina sensoriamento e definições de trajetórias; planejamento de manipulação relacionado com movimentação de objetos como, por exemplo, montagem de peças, organização de containeres, entre outros (GHALLAB; NAU; TRAVESSO, 2004).

Um modo natural de abordar os cenários acima citados seria estudá-los, representá-los e desenvolver sistemas de planejamento de forma específica, ou seja, direcionados para um caso específico com fortes dependências do domínio. Esta abordagem, porém não interessa ao Planejamento Automático. Pelo contrário, esta área da IA está interessada em estudar e desenvolver sistemas capazes de planejar sobre uma variedade de domínios distintos (sistemas independente do domínio) utilizando especificações e representações do problema, bem como

conhecimentos sobre o domínio. Evidentemente, o desenvolvimento de sistemas independente do domínio se torna mais complexo.

A pesquisa na área de Planejamento Automático coloca foco em esforços, principalmente, no desenvolvimento de sistemas computacionais de planejamento, chamados de planejadores (*planners*), que são capazes de procurar por uma seqüência de ações a serem aplicadas em um domínio para atingir um conjunto de metas ou objetivos a partir de um determinado estado inicial. Duas principais características são desejáveis destes planejadores: **eficiência** (no sentido de conduzir para uma solução satisfatória) e **desempenho** (no sentido de não perder tempo em buscas e tentativas frustradas). O problema de planejar no mundo real (o mundo das máquinas, dos elevadores, dos sistemas reais que interagem com o elemento humano, o mundo das entidades automatizados), se torna um grande desafio dado que, enquanto um sistema planeja suas ações, o ambiente (a parte do mundo que o envolve e influencia os parâmetros do planejamento) está sendo modificado por outras entidades que não estão sobre o controle do planejador. Este fato, seja ele na forma de contingências ou de concorrência entre máquinas, traz uma dificuldade ainda maior levando, por exemplo, a uma reformulação na seqüência de ações, ou plano.

Para que o planejador possa planejar é necessário que este raciocine sobre uma descrição (representação), um modelo do domínio. Este modelo possui, de um modo geral, as ações (com suas precondições e efeitos), restrições, recursos disponíveis, os objetos, os objetivos a serem atingidos (eventualmente com critérios de otimização) e o estado inicial de um problema de planejamento. Utilizando esta informação como entrada, o planejador pode fornecer uma coleção organizada das ações de forma a atingir os objetivos pré-estabelecidos, ou seja, uma solução aceitável para o problema. É possível perceber que a especificação do problema e o conhecimento sobre os domínios fornecidos ao planejador possuem um papel fundamental no processo de planejamento automático.

Para um melhor entendimento dos conceitos de Planejamento Automático e do papel do planejador no sistema como um todo, a Figura 1 apresenta um modelo conceitual trazendo uma visão dos conceitos básicos desta área da IA.

Figura 1 – Modelo conceitual de Planejamento Automático.
Fonte: (GHALLAB; NAU; TRAVESSO, 2004) pág. 9.

Na Figura 1 podemos visualizar os três principais elementos que, geralmente, estão presentes em um sistema real de planejamento: o Planejador (*Planner*) que recebe como entrada a descrição de um sistema estado-transição Σ, uma situação inicial, e os objetivos para sintetizar um plano e fornecê-lo ao controlador; o Controlador (*Controller*) recebe o estado atual do sistema real Σ através de observações e providencia a ação de acordo com o plano definido pelo Planejador; e o Sistema estado-transição Σ (*System Σ*) a ser controlado que evolui de acordo com as ações e eventos que este recebe (um sistema estado-transição pode ser representado por um grafo direcionado onde os nós são estados e os arcos são ações ou eventos). Como visto na figura acima, tanto as *ações* do controlador quanto os *eventos* exógenos (*Events*) podem contribuir para a evolução do Sistema Σ. A diferença entre *ações* e *eventos* está na capacidade do planejador de controlá-las. *Ações* são transições que são controladas pelo executor do plano. *Eventos* são transições que são contingentes, isto é, ao

invés de serem controladas pelos executores do plano, elas correspondem a uma dinâmica externa ao sistema estado-transição Σ (GHALLAB; NAU; TRAVESSO, 2004).

O papel do planejador no modelo conceitual é descobrir quais ações devem ser aplicadas para cada estado de Σ para que sejam atingidos os objetivos. Os objetivos podem ser especificados de diversas maneiras, por exemplo, uma simples descrição de um estado, ou uma descrição de uma condição a ser satisfeita durante a evolução dos estados do sistema Σ. É possível especificar objetivos através de funções de penalidade associadas a um determinado estado onde o objetivo passa a ser otimizar esta função.

As *observações* (*Obsevations*), parciais ou não, e o *status de execução* do plano (*Execution status*) são características que tornam o modelo conceitual da Figura 1 mais próximo do real. As *observações*, ou seja, conhecimento sobre o estado do sistema Σ, podem contribuir para um controle mais eficaz sobre as evoluções de Σ, possibilitando ao controlador tratar as diferenças entre o sistema estado-transição Σ e o mundo real. O *status de execução* pode trazer o modelo para um cenário ainda mais realístico onde o planejador leva em consideração um *feedback* do plano fornecido. Esse *feedback* permite ao planejador realizar supervisões e revisões dos planos sintetizados para um eventual replanejamento, o que cria um sistema fechado de planejamento (GHALLAB; NAU; TRAVESSO, 2004).

O Planejamento Automático aborda os diversos aspectos presentes no modelo conceitual apresentado, mas nem todos os conceitos são facilmente tratados. Durante muito tempo esse modelo precisou ser restringido para que fosse possível desenvolver planejadores capazes de apresentar soluções satisfatórias. Essas restrições (simplificações e suposições no modelo conceitual) representavam um conjunto (ou classe) bem definido de problemas. Este conjunto de restrições caracteriza o chamado Planejamento Clássico. Essas simplificações e suposições são descritas a seguir (GHALLAB; NAU; TRAVESSO, 2004):

- A0. **Σ Finito:** O *sistema estado-transição* Σ possui um conjunto finito de estados;
- A1. **Σ Totalmente Observável:** O sistema Σ é totalmente observável, isto é, tem-se total conhecimento do estado de Σ;
- A2. **Σ Determinístico:** O sistema Σ é determinístico;
- A3. **Σ Estático:** O sistema Σ é estático, isto é, o conjunto de *Eventos* exógenos E é vazio;
- A4. **Objetivos restritos:** Os planejadores lidam apenas com *objetivos restritos* que são especificados explicitamente no estado objetivo (*goal state*). Os *objetivos estendidos,* tais como, estados a serem evitados, restrições na trajetória da solução ou funções de otimização, não são permitidos;
- A5. **Planos Seqüenciais:** Um plano-solução de um problema de planejamento é uma seqüência finita de ações linearmente ordenada;
- A6. **Tempo Implícito:** Ações e eventos não possuem duração. Elas são transições de estado instantâneas;
- A7. **Planejamento Offline:** O planejador não percebe mudanças do sistema Σ enquanto está planejando. Ele planeja apenas com as condições iniciais e os objetivos, independentemente da dinâmica que ocorre em Σ. O planejador não recebe o *status de execução*.

As restrições e suposições descritas acima parecem tornar o planejamento apenas um processo trivial de busca por um caminho no grafo de estados, o que é um problema bem conhecido e bem resolvido na Ciência da Computação. De fato, se possuíssemos o grafo que representa Σ explicitamente não haveria muito planejamento a se fazer. Todavia, mesmo para um domínio de planejamento bem simples, o grafo de Σ pode ser tão grande que representá-lo explicitamente seria inviável. Conseqüentemente, é necessário representar o domínio de

forma implícita com uma representação compacta do sistema Σ tornando menos árdua a tarefa de busca por uma solução.

Com a evolução da pesquisa na área de Planejamento Automático, muitas das restrições e simplificações apresentadas anteriormente foram, ao longo do tempo, sendo relaxadas ou até mesmo eliminadas. Esse relaxamento nas suposições, tratadas por planejadores capazes de lidar com problemas mais complexos, criou o chamado Planejamento Neoclássico. Alguns exemplos de relaxamento nas suposições são: a suposição A0 passou a ser relaxada após a inserção de variáveis de estado numéricas causando uma explosão no número de estados; outro exemplo se encontra na suposição A4, pois no planejamento neoclássico já era possível expressar objetivos mais complexos como situações a serem evitadas, estados intermediários a serem atingidos, e outros; a suposição A6 também podia ser relaxada, pois ações com duração passaram a serem consideradas durante o processo de planejamento, entre outras suposições (GHALLAB; NAU; TRAVESSO, 2004).

Uns dos aspectos mais importantes no Planejamento Automático é a representação e modelagem dos domínios, ou seja, a representação de Σ. Todos os sistemas/domínios de planejamento Σ, e seus respectivos problemas, devem ser modelados e representados utilizando alguma linguagem para que sejam compreendidos pelo planejador. O Planejamento Clássico possui três principais formas de representar os domínios de planejamento, são elas: Teoria de Conjuntos, Variáveis de Estados e a mais utilizada representação, chamada Representação Clássica (GHALLAB; NAU; TRAVESSO, 2004). Neste trabalho será consideradas apenas a Representação Clássica e suas evoluções.

A Representação Clássica possui algumas características da representação Variável de Estados. A linguagem mais conhecida e utilizada desta representação é a chamada PDDL – *Planning Domain Definition Language* (MCDERMOTT, 1998). A PDDL é uma das

principais linguagens utilizadas na área de pesquisa de Planejamento Automático. Esta linguagem será detalhada nos próximos tópicos.

2.1.2. Histórico

A história do Planejamento Automático iniciou na década de 60 a partir de trabalhos científicos focados na criação de solucionadores (algoritmos) gerais de problemas (principalmente com o uso de lógica de primeira ordem) como, por exemplo, o GPS (*General Problem Solver*) (ERNST; NEWELL, 1969) (NEWELL; SIMON, 1972) e QA3 (GREEN, 1969). O principal objetivo dos solucionadores era encontrar, automaticamente, soluções para uma variedade de problemas nos mais variados domínios sendo o mais geral possível.

Os primeiros algoritmos de planejamento eram tidos como métodos de busca aplicados a provadores de teorema que utilizavam linguagens formais como Cálculo de Situações (*Situation Calculus*) (GREEN, 1969) ou Cálculo de Eventos (*Events Calculus*) (KOWALSKI; SERGOT, 1969) para representação dos domínios e problemas. Apesar do uso dessas linguagens apresentar um avanço considerável em relação à representação de conhecimento para o desenvolvimento de sistemas de planejamento, ainda não existia um planejador capaz de usufruir, de forma eficaz, das representações dos domínios durante a obtenção das soluções dos problemas. Foi neste cenário que, no inicio dos anos 70, um grupo de pesquisadores do Instituto de Pesquisa de Stanford criou um sistema de planejamento automático, chamado STRIPS (*Stanford Research Institute Problem Solver*) (FILKES; NILSSON, 1971), que se tornaria, além de uma referência, um pioneiro na área. De formulação simples, este planejador marcou o início da ***Era Clássica do Planejamento Automático*** que se estendeu até o começo da década de 90 (GHALLAB; NAU; TRAVESSO, 2004). O STRIPS ficou muito famoso pela sua formulação e representação de ações (ou operadores) que permaneceu quase duas décadas

como paradigma vigente. Como seu algoritmo possuía um alto grau de complexidade, muitas propostas de melhoria surgiram nesta era.

Diante de algumas limitações do STRIPS surgiram alguns planejadores como ABSTRIPS (SACERDOTI, 1974), NOAH (*Nets of Action Hierarchies*, um dos pioneiros a introduzir o conceito de planejamento hierárquico com utilização de tarefas hierárquicas) (SACERDOTI, 1977) e NONLIN (TATE, 1977) com o objetivo de atingir melhores resultados.

A década de 80 foi marcada por uma desaceleração no desenvolvimento desta área da IA. Poucos resultados foram obtidos devido a um certo desânimo da comunidade de planejamento frente às limitações da geração automática de planos com as técnicas existentes. Os planejadores mais famosos nesta década foram o SIPE (o primeiro a lidar com replanejamento) (WILKINS, 1983) (WILKINS, 1984), TWEAK (CHAPMAN, 1987), PRODIGY (MINTON, 1988) e o ABTWEAK (YANG; TENENBERG, 1990) e, apesar dos esforços dos pesquisadores, durante esta fase nenhum mecanismo robusto o bastante para resolver problemas genéricos foi construído.

A representação de domínios, principalmente a representação de ações, ganhou um grande avanço em 1989 com a introdução da linguagem ADL por Pednault (1989) (uma combinação das representações do STRIPS e Cálculo de Situações). Esta linguagem realizava uma extensão na descrição de ações, incorporando, por exemplo, efeitos condicionais (*when..do*, *if..then*) e quantificadores universais (*forall*) podendo assim descrever efeitos dependentes de contexto. Estas extensões permitiram uma maior flexibilidade e poder de descrição dos efeitos de uma ação em domínios já um pouco mais complexos.

O início da década de 90 foi marcado por uma grande busca por melhorias no processo através do qual os planejadores resolviam os problemas, e também por diversos estudos realizados sobre linguagens de representação e técnicas alternativas ao STRIPS. Destacam-se, nesta época, os planejadores O-PLAN (CURRIE; TATE, 1991) SNLP (MCALLESTER;

ROSENBLITT, 1991), POP (BARRET; WELD, 1992) e UCPOP (PENBERTHY; WELD, 1992). Todos estes eram baseados, de certa forma, nos planejadores STRIPS e o HTN (EROL; HENDLER; NAU, 1994), proveniente do planejador NOAH, que fortalecia o planejamento hierárquico. As técnicas de planejamento hierárquico têm sido muito utilizadas em aplicações reais de planejamento como, por exemplo, logística, robótica, planejamento de processo de manufatura, planejamento e escalonamento de espaçonaves, entre outras (GHALLAB; NAU; TRAVERSO, 2004).

Os planejadores desenvolvidos na *era do planejamento clássico* foram chamados de "*sistemas clássicos de planejamento*" e são, de um modo geral, sistemas restritos de transição de estados que utilizam métodos como busca para frente ou para traz, tanto no espaço de estados quanto no espaço de planos (GHALLAB; NAU; TRAVERSO, 2004), para solucionar os problemas de planejamento. Durante a era clássica, grande atenção foi dada ao desenvolvimento e melhoria das técnicas de planejamento, mas a criação e o desenvolvimento de novas linguagens de modelagem direcionadas aos domínios de planejamento foram deixados em segundo plano, assim como um melhor entendimento e especificação dos problemas e domínios deste tipo.

É importante citar que, desde o inicio da era clássica, o Planejamento Automático se tornou uma área de pesquisa muita ativa envolvendo diversos institutos de pesquisa, universidades e grandes empresas.

Mesmo após tantos avanços significativos, os planejadores não conseguiam solucionar diversos problemas, mesmo que simples, em tempo computacional satisfatório. Em meados de 1995 a história do Planejamento Automático ganhou um grande impulso quando Avrim Blum apresentou o planejador GRAPHPLAN (BLUM; FURST, 1995) que utilizava um método de extração de planos diferenciado através de grafos. Sua simplicidade aliada ao seu desempenho superior aos planejadores da época estimulou o desenvolvimento e a pesquisa de

novas técnicas de planejamento. Esse planejador marcou o início da ***Era Neoclássica do Planejamento Automático*** que fez reviver a pesquisa sobre os problemas clássicos de planejamento, pois naquele momento o planejamento clássico parecia estar perdendo forças devido ao descontentamento com expressividades dos modelos e com a complexidade dos algoritmos. O desenvolvimento de técnicas neoclássicas de planejamento trouxe novos espaços de busca e novos algoritmos que permitiram um aumento significativo no tamanho e na complexidade dos problemas de planejamento que podiam ser resolvidos (GHALLAB; NAU; TRAVESSO, 2004).

As pesquisas na *era do planejamento neoclássico* se voltaram, basicamente, para o aperfeiçoamento do GRAPHPLAN, para o uso de novas técnicas de planejamento e também para o uso de novas formas de representar e modelar domínios. No início do planejamento neoclássico destacaram-se técnicas como:

- *Planejamento com Grafos* (*Planning-graph*). Técnica que utiliza estruturas poderosas de alcançabilidade para organizar e restringir o espaço de busca. Destacam-se nesta técnica planejadores como o GRAPHPLAN e seus sucessores;
- *Satisfabilidade Proposicional* (*Propositional Satisfiability*). Técnica que traduz um problema de planejamento em um problema de satisfabilidade (SAT) para utilizar procedimentos eficientes do tipo SAT na resolução de problemas. Podemos citar o planejador SATPlan (KAUTZ; SELMAN, 1996) como um dos destaques desta técnica. Alguns planejadores combinavam técnicas de Planejamento com Grafos e SAT como, por exemplo, o BLACKBOX (KAUTZ; SELMAN, 1999);
- *Satisfação de Restrição* (*Contraint Satisfaction*). De forma similar ao anterior esta técnica traduz o problema de planejamento em um problema de satisfação de restrições trazendo para a área do Planejamento Automático métodos eficientes de planejamento. Um exemplo de planejador que utiliza técnicas avançadas de Satisfação

de Restrição é o DESCARTS (JOSLIN; POLLACK, 1996). Satisfação de Restrição em planejamento já havia aparecido algum tempo na comunidade de planejamento. Algoritmos como MOLGEN (STEFIK, 1981) (que prove uma boa ilustração do gerenciamento de restrições) e UMCP (EROL; HENDLER; NAU, 1994) (algoritmo hierárquico que usufrui bastante desta técnica) já utilizavam este tipo de técnica que mesmo tendo surgindo antes de 1995, é considerada uma técnica neoclássica (GHALLAB; NAU; TRAVESSO, 2004).

Em 1998 o Planejamento Automático recebeu um significante impulso com o surgimento de nova técnica de planejamento, apresentada por Hector Geffner, onde foi possível encontrar um novo paradigma para esta área de pesquisa: *Planejamento por Busca Heurística*. O HSP - *Heuristic Search Planner* (BONET; GEFFNER, 1999) - se tornou uma das técnicas mais rápida de planejamento e sua grande contribuição foi a definição de uma função de custo que guia o processo de busca pela solução.

Como o grande foco da comunidade de planejamento ainda era o desenvolvimento e pesquisa de planejadores, no mesmo ano de 1998 começaram a surgir esforços para definição de uma linguagem comum para modelagem dos domínios de planejamento. Com o objetivo de comparar os planejadores existentes e incentivar ainda mais a pesquisa nesta área da IA, criou-se a linguagem de definição de domínios de planejamento chamada PDDL - *Planning Domain Definition Language* (MCDERMOTT, 1998). Esta linguagem foi utilizada na primeira competição de planejadores chamada de IPC – *International Planning Competition* – que ocorreu durante um dos principais congressos na área de Planejamento Automático, o AIPS (*Arificial Intelligence Planning Systems* - 1998). Nesta competição os planejadores resolviam problemas clássicos de planejamento, bem como problemas reais simplificados.

A competição IPC tinha o objetivo de comparar a performance dos planejadores, unir os pesquisadores da área, estabelecer uma evolução nas terminologias e motivar o desenvolvimento de algoritmos, para que estes se tornassem cada vez mais eficientes. A PDDL foi uma ótima ação na tentativa de padronizar a forma de modelar domínios de planejamento, já que até então cada planejador tratava sua própria linguagem de modelagem de domínios como "*input*". De fato, a PDDL tornou-se um padrão na modelagem de domínios de planejamento, mas mesmo com as sucessivas evoluções, esta vem recebendo críticas até os dias de hoje, principalmente por não ser tão intuitiva (BACHHUS 2003; BODDY 2003; GEFFNER 2003; MCCALLUM 2003; MCCLUSKEY 2003a; MCDERMOTT 2003; SMITH 2003).

Com os avanços, planejadores cada vez mais eficientes surgiram nas competições, como, por exemplo, o FF (HOFFMAN; NEBEL, 2001) que se destacou na competição de 2000 utilizando planejamento por busca heurística, o LPG, destaque da competição de 2002, que incorporava tanto técnicas do GRAPHPLAN quanto técnicas de busca heurísticas, e o METRIC-FF (HOFFMANN, 2002) que se caracterizava como uma evolução do FF capaz de tratar restrições numéricas e funções de otimização. Em 2004 os planejadores Fast (Diagonally) Downward (HELMERT, 2004), SGPLAN (WAH; CHEN, 2004) e SATPLAN'04 (KAUTZ et. al, 2004) obtiveram ótimos resultados, surpreendendo a todos com suas performances, principalmente no tempo de resposta. A Figura 2 mostras uma linha do tempo, contemplando o planejamento clássico e o neoclássico, destacando os principais planejadores, bem como algumas linguagens.

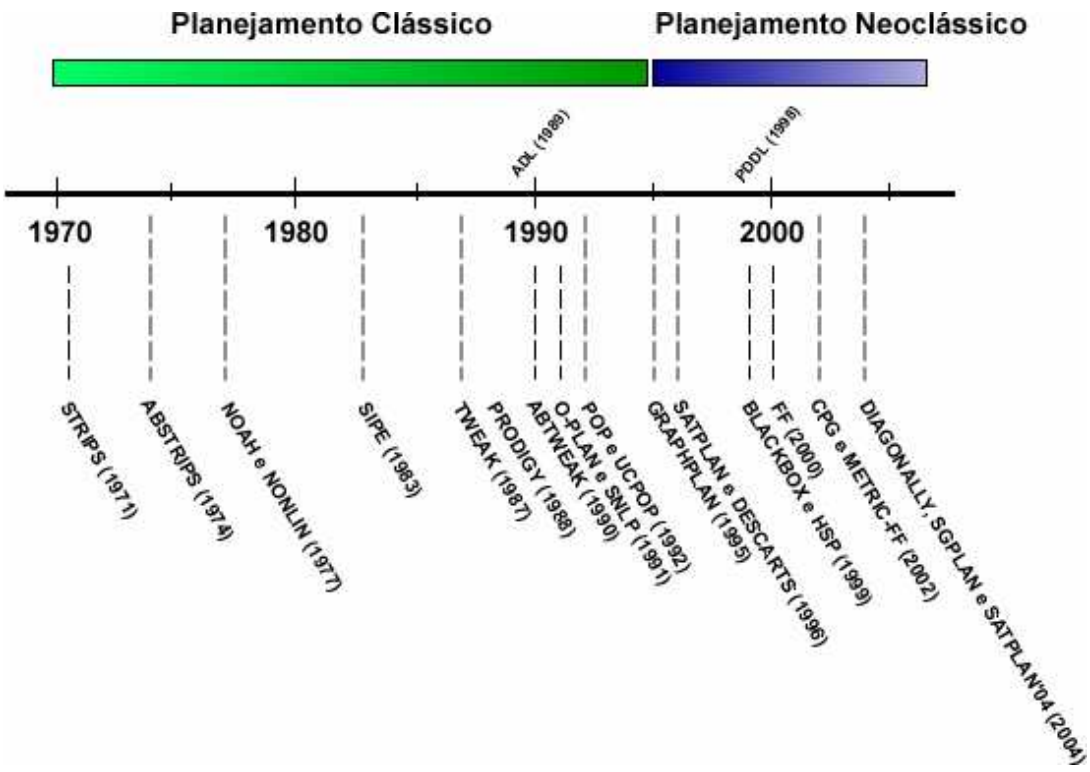


Figura 2 – Linha do tempo do Planejamento Automático (clássico e neoclássico) com os principais planejadores.

Estes avanços chamavam, cada vez mais, a atenção de novas áreas de pesquisa, grandes empresas e institutos de pesquisa fazendo com que os algoritmos evoluíssem mostrando o potencial em alcançar os problemas reais. Com isso, os congressos e competições na área de planejamento ganharam grande visibilidade e a cada ano novas idéias são agregadas aos planejadores e a área de Planejamento Automático como um todo.

As principais conferências, AIPS e ECP, (*European Conference on Planning*) se uniram em 2003 formando a conferência anual chamada ICAPS (*International Conference on Automated Planning and Scheduling*). Esta união, além de fortalecer as tendências do Planejamento Automático, fortaleceu a presença do conceito de escalonamento (*Scheduling*). O escalonamento já estava presente nos congressos anteriores, mas este conceito ganhou espaço, principalmente na competição IPC-02, quando a PDDL evolui e se tornou capaz de

expressar ações com duração e recursos. Essa evolução foi chamada de PDDL 2.1 (FOX; LONG, 2003) e esta linguagem vem evoluindo juntamente com as competições. A PDDL 2.2 (EDELKAMP; HOFFMANN, 2004) e PDDL 3.0 (GEREVINI; LONG, 2005) que serão melhor detalhadas nos próximos itens deste trabalho, são exemplos destas evoluções.

Mesmo com o grande avanço do planejamento automático e com o sucesso das competições e conferências, a comunidade de planejamento percebeu que, de fato, muita atenção estava sendo dada apenas à pesquisa e o desenvolvimento dos algoritmos e pouca atenção era dada para o processo de modelagem de problemas reais como um todo. Essa percepção estava diretamente relacionada com o objetivo de atingir problemas reais. Esse processo de modelagem de domínios de planejamento envolve, por exemplo, análise de requisitos, os métodos de modelagem, as linguagens de representação e modelagem dos domínios, ontologias, análise e verificação de domínios, entre outros. Estes conceitos, inerentes a Engenharia do Conhecimento (*Knowledge Enginnering*) e Engenharia de Requisitos (*Requirements Engineering*), eram negligenciados até então, mas percebeu-se que não importa o quão eficiente é o mecanismo de planejamento ou escalonamento, eles são tão eficientes quanto o conhecimento que estão usando. Em outras palavras, se o modelo do domínio de planejamento e/ou escalonamento for falho então a solução fornecida por um planejador será também falha. Assim, a participação dessas áreas da engenharia vem se tornando cada vez mais evidente e importante para a evolução da pesquisa em Planejamento Automático.

No próximo tópico é apresentada uma breve descrição dos conceitos da Engenharia do Conhecimento aplicada ao Planejamento Automático. Neste mesmo item é descrito como e quando a Engenharia do Conhecimento (*Knowledge Enginnering*) iniciou sua participação na área de Planejamento Automático em IA.

2.2. Engenharia do Conhecimento para Planejamento – *Knowledge Engineering for Planning*

2.2.1. Conceito

A Engenharia do Conhecimento (EC) estuda os princípios e métodos para o desenvolvimento e construção de Sistemas Baseados em Conhecimento. (SBC ou KBS - *Knowledge-Based Systems*) (STUDER; BENJAMINS; FENSEL, 1998). De um modo geral, os SBCs são programas de computador que utilizam o conhecimento representado explicitamente em uma Base de Conhecimento (BC) para resolver problemas através da utilização de métodos de resolução de problemas (em geral, máquinas de inferência provenientes da área da Inteligência Artificial). De fato, estes sistemas (SBC) manipulam conhecimento e informação de forma inteligente e são desenvolvidos para resolverem problemas que requerem grandes porções de conhecimento humano e especialização. A Figura 3 demonstra um esquema simplificado de um SBC.

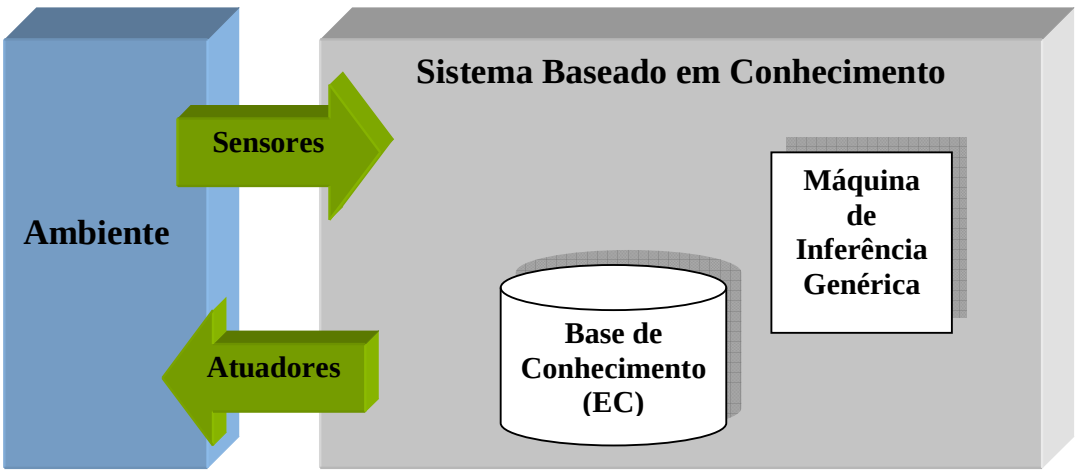


Figura 3 – Esquema básico de um Sistema Baseado em Conhecimento.

Os dois componentes principais dos SBC são: a Base de Conhecimento (EC) que contém, por exemplo, as representações de ações e acontecimentos do domínio, bem como regras; e a Máquina de Inferência Genérica, responsável pelo raciocínio automático sobre a BC e os fatos para a resolução de problemas.

Os SBC são utilizados em diversas aplicações. Algumas aplicações destes sistemas são listadas abaixo:

- Interpretação: Análise de dados para identificação dos seus significados. Por exemplo, processamento de imagem e reconhecimento de voz;
- Classificação: Determinação de falhas em sistemas a partir de um conjunto de sintomas. Por exemplo, diagnóstico de doenças e falhas de máquinas de manufatura;
- Monitorizarção: Observação contínua de um sistema para agir quando uma situação ocorre. Por exemplo: controle de tráfego aéreo e centrais de energia nuclear;
- Planejamento: Determinação de um conjunto de ações para atingir metas determinadas. Por exemplo: operações espaciais e militares, bem como controle de robôs.

Por muito tempo, pesquisas na área de IA focaram no desenvolvimento de formalismos, mecanismos de inferência e ferramentas para operacionalizar os Sistemas Baseados em Conhecimento. Tipicamente, os esforços iniciais de desenvolvimento eram restritos a realização de SBC de pequeno porte com o objetivo de estudar a viabilidade das diferentes abordagens (STUDER; BENJAMINS; FENSEL, 1998).

Embora esses estudos demonstrassem resultados promissores, a aplicação da tecnologia de SBC em uso comercial, onde seriam necessários SBC de grande porte, falhou em muitos casos, devido principalmente à abordagens de desenvolvimentos acadêmicas. Esta situação pôde ser diretamente comparada e relacionada com uma situação similar no desenvolvimento

de sistemas de software tradicionais, chamada de “crise do software”, no final da década de 60, onde os meios nos quais pequenos protótipos acadêmicos estavam sendo desenvolvidos não obtiveram sucesso para o design e a manutenção de sistemas comerciais grandes e de longa duração. Da mesma forma que a “crise do software” resultou no estabelecimento da disciplina Engenharia de Software, o cenário insatisfatório do desenvolvimento dos SBC gerou também a necessidade de abordagens mais metodológicas, bem como métodos disciplinados (STUDER; BENJAMINS; FENSEL, 1998).

Diante deste cenário, surgiu a área da Engenharia do Conhecimento com um objetivo similar a da Engenharia de Software: tornar o processo de construção de SBC uma disciplina da engenharia. Este objetivo requer uma análise do próprio processo de construção e manutenção desses sistemas e também o estudo e o desenvolvimento de métodos e processos disciplinados de eliciação, de especificação e de modelagem, bem como linguagens, métodos de solução de problemas e ferramentas especializadas para o design de SBC.

A Engenharia do Conhecimento aplicada à área de Planejamento Automático pode ser vista como um caso especial da grande área da EC. Desde sua origem, o Planejamento Automático é, de fato, baseado em conhecimento, isto é, o processo de planejamento envolve a manipulação de conhecimento de fenômenos complexos, tais como as ações.

Especificamente no Planejamento Automático é possível definir algumas terminologias para que alguns conceitos não sejam confundidos. O pesquisador Lee McCluskey (2002) deixa bem claro algumas terminologias em alguns de seus trabalhos. O conhecimento associado a uma determinada aplicação real de planejamento é denominado ***domínio***. Qualquer forma de representação simbólica de parte do domínio pode ser chamada de ***descrição do domínio***, por exemplo, um conjunto de documentos descrevendo o domínio, possivelmente em linguagem natural. O termo ***especificação do domínio*** é uma descrição abstrata do *domínio* definitiva e fechada. É esperado que uma *especificação do domínio* seja

completa e precisa. Já a representação simbólica do conhecimento que pode ser utilizada para realizar e executar operações, de modo similar ao *domínio,* é chamada de ***modelo do domínio***. O modelo do domínio é uma *especificação do domínio* na forma operacional, contendo detalhes explícitos das propriedades e da dinâmica do domínio, apropriado para ser processado por um mecanismo de planejamento. Os fenômenos presentes no *modelo do domínio* devem corresponder àqueles presentes no *domínio* real (MCCLUSKEY, 2002).

Como visto nos conceitos de planejamento, os planejadores precisam deste conhecimento, o modelo do domínio e suas respectivas representações dos problemas, para produzir seqüências de ações de modo a atingir metas e diretrizes estabelecidas, especialmente para os planejadores baseados em busca heurística. Como as aplicações de planejamento possuem uma forte dependência das bases de conhecimento, é possível afirmar que a disciplina Engenharia do Conhecimento e seus respectivos processos são de extrema importância para esta área especifica da IA.

A Engenharia do Conhecimento para Planejamento Automático, e também para Escalonamento, estuda processos que envolvem a aquisição, modelagem, validação, verificação, manutenção dos modelos de domínios, bem como a seleção e integração de técnicas de planejamento (um planejador) para raciocinar sobre o modelo especificamente para sistemas de planejamento automático. Alguns dos principais objetivos desta área estão relacionados com a exploração de técnicas e métodos provenientes das áreas da engenharia de aquisição, de requisitos, do conhecimento (grande área) e de software de modo a criar modelos de domínios satisfatórios e integrá-las às técnicas de planejamento desenvolvidas (MCCLUSKEY et al., 2003).

Capturar e representar corretamente o conhecimento em aplicações de planejamento não é uma tarefa simples. Os aspectos do conhecimento nestas aplicações são reconhecidos como sendo a maior dificuldade no projeto. Experiências com planejadores adaptados a aplicações

como, por exemplo, aeroespaciais e militares (WILKINS, 1999) (TATE; DRABBLE; DALTON 1996) (MUSCETTOLA et al., 1998), apontaram que os aspectos da EC são os que requerem mais atenção. O processo de representação do conhecimento envolvido em aplicações de planejamento está fortemente relacionado com a identificação e representação das ações que afetam os objetos do domínio. Esse conhecimento deve ser adequado para permitir que o planejador raciocine e construa planos de maneira eficiente.

Muitos conceitos inerentes às Engenharias de Software, Requisitos e outras engenharias são de grande importância para os processos da EC em domínios de planejamento. É possível citar alguns conceitos, processos e aspectos fundamentais que devem ser levados em consideração na EC para Planejamento Automático:

- **Análise de Requisitos:** analisar e entender profundamente os requisitos do domínio de planejamento para uma melhor descrição e especificação dos domínios. Ao realizar uma análise rigorosa pode-se economizar tempo e recursos nas demais fases do desenvolvimento de sistemas de planejamento automático como, por exemplo, a modelagem do domínio.
- **Os pontos de vista (*viewpoints*) no processo de EC:** o processo de aquisição de conhecimento da EC envolve papéis ou pontos de vistas diferentes para que se possa ter uma ampla visão do domínio de planejamento. Pontos de vista provenientes dos especialistas (*experts)* em planejamento, dos engenheiros do domínio, dos especialistas no domínio, especialistas em softwares, *stakeholders* e usuários são extremamente importantes para uma boa evolução do processo e um bom entendimento do domínio. Naturalmente, uma pessoa pode assumir mais de um ponto de vista. Como esse processo pode lidar com diversas visões, assim como diversos níveis de conhecimento, é necessário o uso de ferramentas de suporte para que todos tenham uma visão unificada do domínio e este se torne consistente.

- **Ferramentas de Suporte ao processo de EC:** as ferramentas de suporte são fundamentais nos processos de análise, aquisição, modelagem, verificação e validação de modelos e/ou dos sistemas de planejamento. Estas ferramentas auxiliam os designers tanto durante os processos de EC quanto na manutenção dos domínios modelados.
- **Modelagem de domínio:** o processo de modelagem de domínios de planejamento envolve a representação do domínio utilizando alguma linguagem de modelagem. Esse processo cria o *modelo do domínio*. No planejamento automático, parte-se do pressuposto que o modelo contenha uma descrição declarativa da dinâmica do domínio real e que o aspecto mais importante no modelo é a descrição das ações. Este modelo declarativo deve ser desenvolvido e elaborado independentemente do planejador ou outro software que contribua no processo. Este fato tende a facilitar a verificação e validação do modelo do domínio. Um modelo criado independentemente do planejador pode ser utilizado por uma variedade de planejadores, ou mecanismos de planejamento, e até mesmo em outras aplicações que não necessariamente são de planejamento.
- **Métodos de modelagem:** boas práticas de modelagem assim como métodos para especificação de domínios podem ser utilizados para guiar todos os processos da EC. Esses métodos contribuem para que o processo de modelagem seja organizado e estruturado e muitas vezes sistemático para que não haja possíveis perdas de informação sobre o domínio.
- **Verificação:** a verificação é o processo de correção e refinamento dos modelos dos domínios, realizado de maneira informal ou semi-formal, baseado, geralmente, na aprovação dos participantes no processo de desenvolvimento.

- **Validação:** a validação de modelo do domínio de planejamento é o processo que promove a qualidade do modelo através de métodos formais de identificação e correções de erros e inconsistências(MCCLUSKEY et al., 2003).
- **Representação do Conhecimento:** em geral, o conhecimento de domínios de planejamento a ser capturado e modelado são as ações, objetivos, atividades, recursos, tempo, restrições. Esse conhecimento deve ser representado utilizando alguma linguagem de representação. Atualmente, a linguagem padrão de representação dos modelos utilizado pela comunidade de pesquisa em Planejamento Automático é a PDDL – *Planning Domain Definition Language* (MCDERMOTT, 1998). Todavia, esta linguagem não foi desenvolvida com uma perspectiva baseada na Engenharia do Conhecimento, o que reforça a necessidade da busca de outras representações do conhecimento, ainda que transferíveis para a PDDL. Esta linguagem será melhor detalhada nos próximos tópicos.

2.2.2. Histórico

O conceito Engenharia do Conhecimento para Planejamento começou a aparecer a partir de 1999 como um dos tópicos de interesse da comunidade de planejamento. Um dos grandes precursores na introdução do conceito da Engenharia do Conhecimento na área de Planejamento Automático foi o pesquisador Lee McCluskey, além dos pesquisadores envolvidos na formação de linguagem de modelagem (por exemplo, a PDDL). O pesquisador, juntamente com seus colegas de pesquisa, já havia realizado trabalhos na área da EC para Planejamento desde o inicio da década de 90 (MCCLUSKEY; PORTEOUS, 1993) (MCCLUSKEY et al., 1995).

A entrada dos conceitos da EC na comunidade de planejamento foi devida a diversos aspectos como, por exemplo, o descontentamento com a linguagem padrão PDDL, a busca

por domínios mais complexos onde as abordagens clássicas não eram suficientes e eficazes, entre outros. Novamente, esse cenário de insatisfação pode ser relacionado e comparado tanto com a "crise do software" quanto com o próprio surgimento da EC conforme descrito anteriormente.

Alguns marcos da Engenharia do Conhecimento aplicada a Planejamento são de grande relevância. Um marco importante foi o trabalho realizado pelo pesquisador McCluskey e seus colegas sob o título "*Knowledge Engineering for Planning ROADMAP*" (MCCLUSKEY et al., 2003). Este trabalho foi iniciado em 2000 e continha os principais conceitos da EC para a comunidade de planejamento, bem como os passos que a própria comunidade deveria tomar para que as aplicações reais fossem alcançadas. Esse trabalho teve um papel fundamental no despertar da comunidade que na época, conforme visto anteriormente, focalizava esforços em domínios clássicos e, principalmente, nas técnicas de planejamento (planejadores).

Um outro importante marco ocorreu em 1997 e 1998 com o aparecimento de algumas ferramentas de planejamento que contemplavam os aspectos da EC. Antes de 1997 as ferramentas de aquisição de domínios de planejamento eram apenas consideradas verificadores de sintaxe e de erros. As duas ferramentas pioneiras na área eram o "*Common Process Editor*" (TATE; POLYAK; JARVIS, 1998), presente no sistema O-PLAN, e o "*Act Editor*" (MYERS; WILKINS, 1997), presente no SIPE. Essas ferramentas e editores nasceram devido à necessidade de desenvolvimento de aplicações reais de planejamento onde o auxílio na modelagem durante os processos de engenharia era fundamental. Essas duas ferramentas eram específicas para determinados domínios, mas estas foram construídas para ajudar a superar os problemas encontrados durante a construção de modelos de domínios complexos, fato que contribui para a relevância da EC para planejamento.

Com o passar dos anos, mesmo com um número pequeno de ferramentas, os interesses nesta área foram aumentando, direcionando alguns focos de pesquisa para sistemas e

ambientes capazes de analisar domínios. Estes sistemas e ferramentas tinham por objetivo processar um modelo de domínio e utilizar ao máximo as informações disponíveis para que estas fossem enviadas a um planejador. A análise de domínios auxiliaria o usuário a modelar e refinar o domínio de interesse. Esses sistemas seriam capazes de: verificar se uma ação (ou operador) modelada está consistente, ou seja, não produzirá um estado inválido se o estado atual for válido; raciocinar sobre as ações e as restrições para que sejam visualmente verificadas pelo usuário; verificar objetivos inconsistentes, entre outros.

As ferramentas de suporte ao processo de EC eram vistas como um ambiente onde o usuário poderia passar por todo o processo de especificação e modelagem e utilizar as técnicas de planejamento disponíveis. Uma das ferramentas pioneiras nesta visão de ambiente de planejamento foi a ferramenta GIPO (SIMPSON et al., 2001). O GIPO é um protótipo acadêmico que além de ajudar no processo de aquisição de conhecimento integra planejadores de modo que o usuário pode escolher qual técnica de planejamento deseja utilizar. Essa ferramenta motivou muitos pesquisadores a trabalhar numa área que é, até os dias de hoje, muito promissora para o Planejamento Automático, a Engenharia do Conhecimento.

Até o início do presente trabalho a principal ferramenta disponível que possuía os objetivos alinhados com a EC para Planejamento Automática era, de fato, o GIPO. Este fato demonstra a grande escassez de ferramentas e métodos de suporte ao desenvolvimento de sistemas de planejamento automático e é exatamente neste cenário que o presente trabalho visa aplicar seus objetivos.

2.3. Engenharia de Requisitos

Neste trabalho, a Engenharia de Requisitos (ER) será vista como um processo inicial sistemático e disciplinado de entendimento e levantamento de necessidades, características e funcionalidades de um domínio para que seu modelo e, conseqüentemente, a Base de

Conhecimento sejam realizados com sucesso. Portanto, a Engenhara de Requisitos transcende o domínio em questão ou seu modelo. A ER está presente nas primeiras fases no processo de desenvolvimento de sistema e que, com certeza, deve estar presente no processo de desenvolvimento de sistemas complexos e reais de Planejamento Automático.

Dentro dos conceitos da Engenharia de Requisitos é importante, inicialmente, distinguir os termos *requisito* e *especificação*. O termo *requisito* pode ser definido como "*condição necessária para a obtenção de certo objetivo, ou para o preenchimento de certo fim no domínio*". Já o termo *especificação* é "*uma descrição rigorosa e minuciosa das características de um domínio*" (FERNANDES, 2005).

De acordo com (IEEE, 1984) e (IEEE, 1991), o processo de aquisição, refinamento e verificação das necessidades tanto do usuário quanto do sistema é chamado de Engenharia de Requisitos (ER). O objetivo da ER é sistematizar o processo de definição dos requisitos, obtendo uma especificação correta e completa dos requisitos. A ER surgiu da grande área Engenharia de Software onde existem processos disciplinados visando tornar mais eficaz o software e o seu processo de desenvolvimento. Uma informação importante é que dependendo do porte e da complexidade do sistema a ser desenvolvido, o custo com as atividades de requisitos do sistema situa-se entre 10% e 15% do custo total devido a sua importância (KOTONYA; SOMMERVILLE, 1998).

Boehm (BOEHM, 1989) define a ER como uma disciplina cujo objetivo é desenvolver uma especificação completa, consistente e não ambígua, servindo de base para um *acordo* entre todas as partes envolvidas e descrevendo *o quê* o domínio é, o que vai fazer ou executar, mas não como ele será feito.

A ER estabelece o processo de definição de requisitos como um processo no qual o domínio deve ser *eliciado, analisado e documentado*. Este processo deve ser baseado em diferentes pontos de vista (conhecido como *viewpoints*) e também deve utilizar uma

combinação de métodos, ferramentas e pessoal, incluindo todas as fontes de informação e todas as pessoas relacionadas ao domínio como, por exemplo, projetistas, especialistas (*experts*), usuários, *stakeholders* e outros. O produto deste processo é uma descrição, geralmente em linguagem natural, que servirá para produzir um documento de requisitos.

No âmbito de planejamento, para se produzir um documento de requisitos completo e consistente é muito importante que o contexto em que o domínio e o problema se situam seja devidamente entendido e analisado, ou seja, os objetivos e metas do sistema a ser desenvolvido, as tarefas e atividades fundamentais para a "engenharia" deste sistema e os limites e restrições do desenvolvimento devem ser bem explorados e investigados.

Os próximos tópicos descrevem os principais conceitos da ER pertinentes ao presente trabalho.

2.3.1. Eliciação de Requisitos

Eliciação significa obter e tornar explícito o máximo de informações possíveis para o conhecimento de um objeto em questão. O termo eliciação, cuja origem em inglês é *elicitation*, tem como sentido: extração, dedução, derivação. Na fase de eliciação de requisitos o engenheiro de requisitos procura captar os requisitos do domínio, buscando obter conhecimento do domínio e do problema. Para alcançar tal objetivo, além da utilização de métodos, ferramentas e pessoal, esta fase utiliza basicamente três atividades principais: *identificação das fontes de informação*, *coleta de fatos* e *comunicação* (BERTOLIN, 1998)..

Para *identificar as fontes de informação* o engenheiro de requisitos identifica todos os recursos que disponibilizam informações sobre o domínio e do problema. Estas informações são provenientes dos pontos de vista dos participantes (autores, usuários, projetistas, *experts*, *stakeholders*, entre outros), documentações, manuais, livros sobre o tema relacionado e até mesmo outros sistemas.

Na atividade de *coleta de fatos* são feitas entrevistas com os clientes e participantes para a captura do conhecimento (considerando tanto o conhecimento explicito quanto tácito), consulta-se os materiais existentes que descrevem os objetivos dos participantes, e também efetua-se a pesquisa sobre a existência de sistemas similares para uma posterior análise. Outras técnicas importantes para a coleta de fatos sobre um sistema são: leitura de documentos, observação, questionários, análise de protocolos, participação ativa dos participantes (autor, usuário, especialistas, *stakeholders*) e reuniões (BERTOLIN, 1998).

Para que a eliciação tenha sucesso é fundamental que os engenheiros de requisitos se *comuniquem* eficazmente com todos os participantes que entendem do domínio.

2.3.2. Análise de Requisitos

A fase de análise de requisitos é fundamental para o sucesso do processo de desenvolvimento de um sistema. Nesta fase, o projetista (engenheiro de requisitos) especifica as funções e o desempenho do domínio, suas interfaces com outros sistemas e estabelece as restrições de projeto.

O objetivo da fase de análise de requisitos é avaliar e revisar o escopo do sistema (documento de requisitos). Através de um processo de descoberta, refinamento, especificação e modelagem, o projetista procura obter uma especificação de requisitos completa e consistente. Pelas razões expostas acima, é muito provável que o documento de requisitos obtido até então, possua várias inconsistências ou problemas de funcionalidade, e o projetista durante esta fase deve ser capaz de detectar e resolver inconsistências (SANTOS, 2002). As decisões de análise servem para realimentar e melhorar o documento de requisitos do sistema, pois este será a base para todas as fases de desenvolvimentos subseqüentes. Existem poucas ferramentas que auxiliam a análise de requisitos, principalmente para sistemas de Planejamento automático.

Como resultado das fases de eliciação e de análise de requisitos é desenvolvido o documento de requisitos do sistema que contém a especificação de requisitos. Este documento é utilizado como base para as fases posteriores de design. O Documento de Requisitos é descrito a seguir.

2.3.3. Documento de Requisitos

O documento de requisitos contém todos os requisitos funcionais e de qualidade do sistema, incluindo as capacidades do sistema, os recursos disponíveis e necessários, os benefícios e os critérios de aceitação. Este documento serve como um meio de comunicação entre o projetista do domínio e os participantes, a fim de estabelecer uma visão geral uniformizada (um "acordo") acerca do sistema. Deve-se evitar que durante o desenvolvimento do documento de requisitos decisões de projeto sejam tomadas (SANTOS, 2002).

Assim, devido à importância do documento de requisitos dentro do processo de desenvolvimento do sistema, é fundamental que este documento seja organizado de forma a melhorar a compreensão e a legibilidade dos requisitos, evitando que problemas e erros surjam na fase de implementação do software.

Depois de realizadas as fases de requisitos inicia-se o design onde o domínio é modelado e analisado baseado no documento de requisitos. A fase de modelagem tem por objetivo criar e desenvolver modelos que descrevem estática e dinamicamente o que o domínio deve fazer, e não como deve ser feito. Estes modelos expressam os requisitos descritos no documento de requisitos, possibilitando um maior entendimento do domínio da aplicação, servindo para ajudar a determinar se a especificação está completa, consistente e precisa. Diversos métodos e linguagens para apoiar o projetista na modelagem de sistemas existem na literatura como, por exemplo, a OMT com o uso da UML (OMG, 2001). As linguagens utilizadas no presente

trabalho para modelar e analisar domínios de planejamento serão apresentadas nas próximas seções.

2.4. Linguagens de Modelagem de Domínios

2.4.1. PDDL – *Planning Domain Definition Language*

Em 1998, foi criada a linguagem padrão de definição de domínios de planejamento chamada PDDL - *Planning Domain Definition Language* (MCDERMOTT, 1998). Esta linguagem tem por principal objetivo representar os domínios do mundo real através de uma estrutura capaz de ser entendida e interpretada por um planejador. A maioria dos planejadores, hoje desenvolvidos, são capazes de utilizar a PDDL como representação de entrada do domínio para a geração de uma solução ou plano já que esta linguagem tornou-se um padrão na área de Planejamento Automático. A representação do modelo do domínio deve ser a mais próxima possível do domínio real contendo a descrição das ações possíveis no domínio, suas pré e pós-condições, as informações sobre o estado inicial do domínio e o estado objetivo (metas) para que o planejador possa processar o modelo.

Algumas características principais da PDDL são:

- A PDDL é uma representação direcionada às ações do domínio;
- As ações representadas em PDDL são baseadas em ações do modelo STRIPS (FIKES; NILSSON, 1971) onde as pré-condições e efeitos de uma ação representam a dinâmica da execução desta no domínio;
- Possui características da linguagem ADL (PEDNAULT, 1989) que incluem a representação de efeitos condicionais nas ações, assim como qualificadores e quantificadores universais;

- Definição de restrições;
- Especificação de ações hierárquicas compostas por sub ações (MCDERMOTT, 1998);
- Devido ao fato da linguagem ser padronizada é possível que um mesmo problema representado em PDDL seja tratado por vários planejadores diferentes (portabilidade de problemas entre agentes planejadores).

A PDDL está em constante evolução desde sua criação. Uma das principais evoluções da PDDL foi a chamada PDDL 2.1 (FOX; LONG, 2003) que será utilizada como referência neste trabalho. A PDDL 2.1 foi desenvolvida com a intenção de representar domínios de planejamento determinísticos que envolvam tempo e que necessitem de recursos de manipulação algébrica, incorporando também características da linguagem ADL (PEDNAULT, 1989). A PDDL 2.1 pode ser definida em cinco níveis: o nível 1 é definido pela primeira versão da PDDL desenvolvida para a competição AIPS-98. O nível 2 é um complemento ao nível 1, permitindo a utilização de recursos numéricos como a comparação entre variáveis numéricas e atualização dos valores das mesmas. Os níveis 3 e 4 definem a representação de ações com dependência temporal tanto com efeitos não contínuos (nível 3) como com efeitos contínuos (nível 4). O nível 5 é uma extensão do nível 4 capaz de representar domínios contínuos e discretos em tempo real.

A representação de um modelo de domínio de planejamento em PDDL é dividida em duas partes. A primeira parte possui a definição do domínio onde são encontradas, principalmente, as ações possíveis no domínio assim como a declaração dos tipos de objetos existentes. Já a segunda parte possui a definição do problema de planejamento a ser resolvido onde são fornecidos os estados iniciais do problema e o objetivo a ser atingido. Cada uma dessas partes é fornecida ao planejador na forma de arquivo (.pddl). A separação da definição do domínio e

dos problemas é um fator positivo já que para uma mesma definição de domínio é possível raciocinar sobre diversos problemas.

A PDDL possui um formalismo para as definições do domínio e do problema. Este formalismo é apresentado de forma resumida nas próximas seções. Toda a definição formal da linguagem pode ser encontrada na especificação da PDDL 2.1 (FOX; LONG, 2003) ou em versões posteriores como a PDDL 2.2 (EDELKAMP; HOFFMAN, 2004) e PDDL 3.0 (GEREVINI; LONG, 2005).

2.4.1.1. Uma visão geral da sintaxe

A sintaxe da linguagem PDDL 2.1 segue o seguinte formato:

- Cada regra é escrita no formato <elemento sintático> ::= expansão.
- Os símbolos menor e maior (< e >) delimitam os nomes dos elementos sintáticos.
- Colchetes ([e]) representam informações opcionais.
- O asterisco (*) representa zero ou mais elementos, enquanto o sinal positivo (+) representa um ou mais elementos.
- Parênteses são apenas delimitadores dos elementos sintáticos, não possuindo nenhuma interpretação semântica.
- Informações adicionais e regras de expansão podem aparecer sobrescritas por um marcador. A descrição do domínio de um problema deve definir a função de cada marcador que aparecer nos elementos sintáticos.

Levando estas regras em consideração são apresentadas a seguir as sintaxes para definição de domínios e problemas

2.4.1.2. Definição de Domínios

O formalismo e a estrutura para definir domínios de planejamento na linguagem PDDL 2.1 segue a BNF apresentada na Figura 4. Nesta representação as palavras chaves devem aparecer na ordem em que são representadas na figura.

```
<domain> ::= (define (domain <name>)
                  [<required – def>]
                  [<types – def>]:typing
                  [<constants – def>]
                  [<predicates – def>]
                  [<functions – def>]:fluents
                  <structure – def>*)
<require – def>          ::= (:requirements <require – key> +)
<require – key>          ::= Podendo ser :strips :adl :typing :fluents e outros
<types – def>            ::= (:types <typed list (name)> )
<constants – def>        ::= (:constants < typed list (names) >)
<predicates – def>       ::= (:predicates < atomic formula > +)
<atomic formula skeleton>::= (<predicate> <typed list
                               (variable)>)
<predicate>              ::= <name>
<variable >              ::= ?<name>
<atomic function skeleton> ::= (<function-symbol> <typed list
                               (variable)>)
<function-symbol>        ::= <name>
<functions-def>          ::= :fluents (:functions <function typed list
                               (atomic function skeleton)>)
<structure-def>          ::= <action-def>
```

Figura 4 – Sintaxe de definição de domínios em PDDL (FOX; LONG, 2003).

A seguir são descritos alguns dos elementos da estrutura acima apresentada:

name: A categoria <*name*> consiste de uma palavra iniciada por uma letra, podendo conter caracteres alfanuméricos, além de hífens (–) e traços subscritos (_). Os nomes definidos em <*name*> devem ser únicos.

requirements: A PDDL foi estruturada em subconjuntos de características, chamados *requirements*. A cada definição de um novo modelo de domínio, os *requirements* a serem utilizados devem ser declarados, pois estes indicam características especificas que estarão

presentes no modelo. Por exemplo, uma vez que o *requirement* [*:disjunctive-preconditions*] seja declarado, é permitido que o operador lógico OU seja utilizado nos objetivos. Ou, por exemplo, se o *requirement* [*:typing*] for declarado, o modelo pode representar tipos de objetos. De fato, alguns planejadores suportam apenas algumas características declaradas no *requirements* fazendo com que o mesmo sinalize que não possui condições para processar o modelo. Os *requirements* mais comuns são:

- *:strips* - É a característica que define os domínios com ações no formato STRIPS;
- :*adl* - É um super conjunto do formato STRIPS, incluindo declaração de tipos de variáveis (:*types*), pré-condições negadas (:*negative-preconditions*), pré-condições disjuntas (:*disjunctive-preconditions*), igualdades (:*equality*), pré-condições quantificadas (:*quantified-preconditions*) e efeitos condicionais (:*conditional-effects*).

types: É uma lista usada para declarar tipos de entidades/classes que estão presentes no modelo do domínio. Os tipos são precedidos por um hífen (–) e são declarados ao final de uma lista de objetos. Nesta lista é possível definir hierarquia de tipos. Exemplo de uma declaração de tipo:

```
(:types
      car truck plane – vehicle
      airport city - place)
```

constants: Os nomes declarados neste campo, são tomados como constantes em todo o modelo domínio, bem como em todos os problemas deste domínio.

```
(:constants
      RiodeJaneiro SaoPaulo – city
      Congonhas – airport)
```

predicates: Este campo consiste de uma lista de declarações de predicados, onde para cada um deles é especificada uma lista de variáveis, além de seus argumentos.

```
(:predicates
      (at ?veh – vehicle ?pla – place))
```

functions: Este campo consiste de uma lista de declarações de predicados que serão associados a valores numéricos, onde para cada dos predicados *functions* é também especificada uma lista de variáveis, além de seus argumentos.

```
(:functions
      (fuel-used ?veh – vehicle))
```

2.4.1.3. Definição de Ações

A representação das ações é uma das principais partes dentro da definição do domínio de planejamento. A sintaxe utilizada na declaração das ações demonstrada na Figura 5 depende do conjunto de tipos declarados na cláusula :*requirements.* Como por exemplo, as ações declarados no formato ADL (:adl) (PEDNAULT, 1989) são mais abrangentes.

A condição para que uma ação seja realizada em um estado S_n depende da instanciação dos predicados da pré-condição <*GD*> e seus correspondentes valores verdade. Uma vez que os predicados da precondição sejam instanciados e seus valores habilitem a execução da ação, os predicados contidos em <*atomic – eff* > são totalmente instanciados, e estes são adicionados ao estado sucessor S_{n+1} à aplicação da ação, assim como os que são removidos do estado sucessor.

As ações declaradas com efeitos no formato (*when* <*GD*> <*atomic – eff*>) são aplicadas a um estado S_n se a expressão <*GD*> for satisfeita em S_n, fazendo com que os predicados <*atomic – eff* > sejam adicionados ao estado sucessor S_{n+1}. Ações que contenham efeitos

declarados nos formatos *(forall (<typed list (variable) >)+ <atomic – eff>)* e no formato *(forall (<typed list (variable)+ >) (when <GD > <atomic – eff>))* são aplicadas a partir de todas as possíveis instâncias de *<typed list (variable)>*. Os efeitos no formato (*and <eff – formula> <eff – formula>+)* são aplicados ao estado S_n, através da inclusão de cada um dos efeitos no estado sucessor S_{n+1}.

```
<action – def> ::= (:action <name>
                        :parameters (<typed list (variable)>)
<action –def body> ::= [:precondition <GD>]
                        [:effect <eff – formula>]
<GD>              ::= <literal term>
<GD>              ::= not <GD>
<GD>              ::= (and <GD> <GD>+)
<GD>              ::= (or <GD> <GD>+)
<GD>              ::= (imply <GD> <GD>)
<GD>              ::= (exist (<typed list (variable)>+) <GD>)
<atomic – eff>    ::= <literal term>
<atomic – eff>    ::= (and <literal term> <literal term>+)
<eff – formula*>  ::= <atomic –eff>
<eff – formula*>  ::= (when <GD> <atomic –eff>)
<eff – formula*>  ::= (forall (<typed list (variable)>+) <atomic –
                        eff>))
<eff – formula*>  ::= (forall (<typed list (variable)>+)
                        (when <GD> <atomic –eff>))
<eff – formula*>  ::= <eff – formula*>
<eff – formula*>  ::= (and <eff – formula*> <eff – formula*>+)
```

Figura 5 - Sintaxe de definição de ações em PDDL (FOX; LONG, 2003).

Para exemplificar a definição completa do domínio em PDDL a Figura 6 demonstra um domínio simples de veículos chamado “*vehicleDomain*”.

```
(define (domain vehicleDomain)
   (:requirements :strips :typing)
   (:types  vehicle
            location
            fuel-level)
   (:predicates   (at ?v – vehicle ?p – location)
                  (fuel ?v – vehicle ?f – fuel-level)
                  (accessible ?v – vehicle ?p1 ?p2 – location)
                  (next ?f1 ?f2 – fuel-level))

   (:action drive
      :parameters (?v – vehicle ?from ?to – location
                        ?fbefore ?fafter – fuel-level)
      :precondition (and      (at ?v ?from)
                              (accessible ?v ?from ?to)
                              (fuel ?v ?fbefore)
                              (next ?fbefore ?fafter))
      :effect (and (not       (at ?v ?from))
                              (at ?v ?to)
                              (not (fuel ?v ?fbefore))
                              (fuel ?v ?fafter))
   )
)
```

Figura 6 – Exemplo de definição de domínio em PDDL com uma ação de exemplo (FOX; LONG, 2003).

2.4.1.4. Definição do Problema de Planejamento

O problema de planejamento é definido basicamente por dois estados, o estado inicial do problema e o estado objetivo. O problema submetido ao agente planejador, também conhecido como arquivo de fatos, é definido em relação a um determinado domínio, descrevendo os fatos verdadeiros no estado inicial S_0, e os fatos descritos como estado objetivo S_G. A sintaxe para definição de problemas é definida, de forma simplificada, conforme Figura 7.

```
<problem>   ::= (define (problem <name>)
                  (: domain <name>)
                  [<require-def>]
                  [<object declaration>]
                  <init>
                  <goal>
                  [<metric-spec>]
<object declaration>    ::= (:objects <typed list (name)>)
<init>                  ::= (:init <init-el*>)
<init-el>               ::= <literal (name)>
<init-el>               ::=:fluents (= <f-head> <number>)
<goal>                  ::= (:goal <GD>)
<metric-spec>           ::=(:metric <optimization> <função a ser otimizada>)
<optimization>          ::= minimize
<optimization>          ::= maximize
```

Figura 7 – Sintaxe de definição de problemas (FOX; LONG, 2003).

O estado inicial declarado em *:init* de um problema é uma lista de fórmulas atômicas consideradas verdadeiras no estado inicial S_0. A descrição do estado inicial de um problema deve assumir um mundo fechado, ou seja, qualquer predicado não declarado em um estado é presumido como falso. Todos os objetos pertencentes ao domínio devem ser declarados em *:objects.* O estado objetivo é declarado em *:goal.* A solução para um problema aplicado em um domínio é a aplicação de uma seqüência de ações a partir do estado inicial S_0 levando a um estado resultante S_R e que o estado objetivo S_G seja um subconjunto de S_R.

Para exemplificar a definição de um problema de planejamento em PDDL a Figura 8 demonstra um problema clássico de locomoção de veículos onde existem dois veículos e três locais. O objetivo deste problema é fazer com que os dois carros (*car1* e *car2*) vão para o local *minas* observadas as restrições de combustível, pois um dos carros (*car1*) se encontra no local *saopaulo* e o outro se encontra no local *rio.*

```
(define (problem twovehicles)
   (:domain vehicle)
   (:objects      car1 car2 - vehicle
                  saopaulo rio minas - location
                  empty half full - fuel-level)
   (:init   (at car1 saopaulo)
            (at car2 rio)
            (fuel car1 half)
            (fuel car2 full)

            (next empty half)
            (next half full)

            (accessible car1 saopaulo rio)
            (accessible car1 saopaulo minas)
            (acessicble car2 rio saopaulo)
            (accessible car2 saopaulo minas))
   (:goal
            (at car1 minas)
            (at car2 minas))
)
```

Figura 8 – Exemplo de definição de problema em PDDL.

2.4.1.5. Histórico da Evolução da Linguagem

Até 1998, pesquisadores da comunidade de Planejamento Automático já haviam desenvolvido muitos algoritmos e técnicas de planejamento. Essas técnicas eram resultado de pesquisas em universidades e centros de pesquisas principalmente na Europa e nos Estados Unidos da América. Cada uma dessas técnicas e algoritmos tinha sua própria forma de representar os domínios e os problemas a serem resolvidos, ou seja, usavam linguagens distintas de modelagem tornando a tarefa de comparação de performance muito árdua. Por exemplo, o QA3 (GREEN, 1969) utilizava Cálculo de Situações; o planejador STRIPS utilizava sua própria linguagem baseada em Cálculo de Predicados (FIKES; NILSSON, 1971); o planejador NOAH utilizava uma linguagem chamada SOUP (*Semantics Of User's Problem*) (SACERDOTI, 1977); o NONLIN (TATE, 1976) utilizava a linguagem TF (*Task Formalism*); entre outros.

Diante desse cenário, a comunidade de planejamento automático, composta basicamente por membros da Ciência da Computação, sentiu a necessidade de um mecanismo mais homogêneo de se modelar os domínios para que fossem possíveis comparações e conseqüentemente evoluções nos algoritmos. Assim, em 1998, Drew McDermott criou a linguagem de modelagem de domínio de planejamento chamada de PDDL – *Planning Domain Definition Language* (MCDERMOTT, 1998) – com o propósito de facilitar o desenvolvimento das técnicas de planejamento e, principalmente, criar meios para se comparar as técnicas desenvolvidas.

A primeira versão da PDDL reuniu características das principais linguagens utilizadas pelos algoritmos existentes. Esta herdava os formalismos de linguagens como o STRIPS (FILKES; NILSSON, 1971), ADL (PEDNAULT, 1989), UCPOP (PENBERTHY; WELD, 1992), UMCP (EROL, HENDLER, NAU, 1994), LISP (MCCARTHY, 1962), entre outras. A linguagem tornou-se um padrão entre a comunidade de planejamento no mesmo ano de sua criação, em 1998.

No mesmo ano da padronização da linguagem ocorria a quarta edição do congresso AIPS – *International Conference on Artificial Intelligence Planning & Scheduling* –, um dos principais eventos científicos da área. Neste congresso foi inaugurada, por Drew McDermott, a primeira competição entre algoritmos chamada IPC – *International Planning Competition* – onde os pesquisadores podiam testar e comparar suas técnicas utilizando a PDDL. Apesar de algumas insatisfações da comunidade com algumas de suas características, a PDDL foi, naquele momento, uma linguagem capaz de considerar os progressos feitos na área de pesquisa de Planejamento Automático.

Com o passar dos anos, a área de Planejamento Automático passou a colocar em foco a utilização dos algoritmos (*planejadores*) em problemas reais e mais complexos. Com este

foco a PDDL passou a ter objetivos maiores, não apenas para comparações nas competições, mas também para ser aplicada e utilizada em sistemas reais.

A competição do ano de 2000 (IPC-2), que já atraiu muitos competidores, utilizou a mesma versão da PDDL não sofrendo extensões. Em 2002, Maria Fox e Derek Long estenderam a PDDL que passou a expressar tempo e expressões numéricas. A necessidade de características como essas surgiram de grandes centros de pesquisa como a NASA com aplicações de planejamento em espaçonaves onde existiam processos espaciais complexos. Essa evolução da PDDL foi chamada de PDDL 2.1 (FOX; LONG, 2003) que se tornou o padrão para a competição de 2002 (ICP-3). Essa versão era capaz de modelar, por exemplo, recursos (caminhões, ferramentas, etc.), restrições ou capacidades (restrição de peso, número de passageiros em um elevador, energia), consumo de recurso (gasolina), ações com duração, ações com expressões numéricas, métricas de plano, entre outras.

Em 2003, muitos pesquisadores questionavam o uso da PDDL nas competições e em eventuais aplicações reais, pois parecia existir uma grande lacuna tecnológica entre as competições e os domínios reais em relação a linguagem PDDL (BACHHUS 2003; BODDY 2003; GEFFNER 2003; MCCALLUM 2003; MCCLUSKEY 2003a; MCDERMOTT 2003; SMITH 2003). Muitas críticas foram lançadas sobre a PDDL, pois existia uma grande dificuldade da PDDL em expressar problemas realísticos de planejamento. Uma das críticas que resume bem o cenário da época foi exposta por Bacchus em 2003 (BACCHUS, 2003):

> A PDDL 2.1 é um padrão muito útil para a competição de planejamento, mas o seu design não considera apropriadamente a questão da modelagem do domínio. Não seria muito recomendável usá-lo para especificar ou modelar domínios de planejamento fora do contexto da competição. O campo de Planejamento em IA precisa explorar diferentes abordagens e precisa estar mais próximo efetivamente da modelagem e utilização de todos os diversos avanços que temos em domínios de planejamento.

Como visto no comentário acima existia uma grande dificuldade da PDDL de modelar problemas reais, pois a linguagem parecia não ser feita com este propósito onde muitos aspectos de modelagem deveriam ser considerados. Com o passar dos anos a linguagem foi evoluindo com o objetivo de expressar domínios cada vez mais complexos e realísticos. Foi neste contexto que a PDDL 2.2 (EDELKAMP; HOFFMANN 2004) e a PDDL 3.0 (GEREVINI; LONG, 2005) nasceram com novas características que realmente indicaram uma melhora do poder de modelagem. A PDDL 2.2 trazia novas características como os "predicados derivados" que representavam predicados que poderiam ser deduzidos de outros e o "*timed initial literal*" que permitia representar modificações de predicados em tempos específicos durante a execução do plano. Já a PDDL 3.0 trazia características como a possibilidade de expressar restrições na trajetória do plano e também restrições e invariantes de estados. As evoluções eram geralmente resultantes de discussões em competições e congressos, principalmente no ICAPS – *International Conference on Automated Planning and Scheduling* (como visto anteriormente, a união dos congressos AIPS e ECP).

Com as continuas evoluções da PDDL, em breve será possível modelar problemas de planejamento realmente complexos, mas, obviamente, essa linguagem não é recomendada durante a fase de requisitos e entendimento do problema. A PDDL acaba sendo ótima para fases posteriores como a modelagem (principalmente para problemas clássicos) ou apenas como uma linguagem intermediária entre a modelagem e o planejador. É realmente difícil modelar domínios reais utilizando a PDDL, principalmente para pessoas não familiarizadas com planejamento automático.

Neste projeto a PDDL não é utilizada como uma linguagem de modelagem e sim como uma linguagem de comunicação entre o processo de modelagem e o processo de planejamento realizado por técnicas de planejamento.

2.4.2. UML – *Unified Modeling Language*

A UML – *Unified Modeling Language* – é uma linguagem semi-formal de modelagem, de propósito geral, que se tornou um consenso e um padrão na modelagem e especificação de sistemas orientados a objetos. Foi padronizada pela OMG (*Object Mangement Group*) entre 1996 e 1997 (D'SOUZA;WILLS, 1999). Como a própria OMG descreve em (OMG, 2001):

> A UML é uma linguagem gráfica para visualização, especificar, modelagem, construção e documentação de artefatos de um sistema. Ela representa a união das melhores práticas na modelagem orientada a objeto. A UML é o produto de vários anos de trabalho onde o foco foi unificar os métodos mais utilizados no mundo adotando boas idéias de muitas aplicações industriais e, acima de tudo, um esforço concentrado em fazer as coisas claras.

De acordo com (OMG, 2001) são objetivos da UML:

- Fornecer ao usuário uma linguagem visual expressiva, pronta para o uso no desenvolvimento de modelos de negócios;
- Fornecer mecanismos para extensões e mecanismos de especialização para apoiar os conceitos essenciais;
- Ser independente de linguagem de implementação;
- Encorajar o crescimento do número de ferramentas com abordagem orientadas a objetos;
- Suportar conceitos de desenvolvimento de níveis mais elevados tais como colaborações, padrões e componentes;
- Integrar as melhores práticas de desenvolvimento de software.

A UML é baseada em diagramas. Alguns diagramas desta linguagem são: o *Diagrama de Casos de Uso*, que define de um modo abstrato o comportamento do sistema ou domínio como um todo; o *Diagrama de Classes*, que descreve as classes (entidade) que formam o domínio, bem como as associações entre elas definindo a estrutura estática do domínio; o *Diagrama de Estados* que representa os estados, condições e efeitos dos eventos e ações que ocorrem no sistema; o *Diagrama de Objetos* que representa os objetos existentes no sistema e como eles interagem em um determinado instante; o *Diagrama de Seqüência*; o *Diagrama de Atividades*, o *Colaborativo* e outros. A UML possui, além dos diagramas, uma linguagem predefinida de representação de restrições chamada OCL – *Object Constraint Language* (OMG, 2003). A OCL é uma linguagem formal utilizada para definir restrições complementares aos diagramas.

Nesta proposta, inicialmente, são utilizados os diagramas de *Casos de Uso*, *Classes*, *Estados* e de *Objetos* na modelagem. A OCL também é utilizada, principalmente para a definição das pré e pós-condições das ações pertencentes ao domínio. A seguir são descritos brevemente estes diagramas e a linguagem OCL.

2.4.2.1. Diagrama de Casos de Uso

O *Diagrama de Casos de Uso* é geralmente o primeiro diagrama desenvolvido. Com este diagrama o projetista pode: decidir e descrever os requisitos funcionais do sistema; fornecer uma descrição clara e consistente do que o sistema deve fazer; e refinar os requisitos. Neste diagrama é possível descrever o sistema de uma forma abstrata fazendo com que haja uma visão geral e simplificada do domínio.

Este diagrama mostra o relacionamento entre os *Casos de Uso* e os *Atores* (Agentes) do sistema. Um *Caso de Uso* é basicamente uma descrição (em muitos casos utilizando linguagem natural ou estruturada) de um caso ou processo do sistema. Cada *Caso de Uso*

contém geralmente os requisitos funcionais, suas restrições (pré e pós-condições e invariantes), bem como uma seqüência de ações e acontecimentos realizados por um ou mais *Atores*. *Atores* são entidades do domínio que estimulam/solicitam ações e eventos do sistema e recebem reações. Um *Ator* pode estar relacionado com mais de um *Caso de Uso*. Um exemplo de *Diagrama de Casos de Uso* é apresentado na Figura 9 onde os atores "*Cliente"* e "*Caixa"* participam do caso de uso "*Comprar item*" (caso que representa e descreve o processo de compra de um item pelo cliente).

Figura 9 – Diagrama de Casos de Uso.

Vale ressaltar que o *Diagrama de Casos de Uso* é muitas vezes utilizado tanto para a especificação do sistema quanto para a modelagem.

2.4.2.2. Diagrama de Classes

O *Diagrama de Classes* representa principalmente a estrutura estática do modelo do domínio. Considerado um dos diagramas mais importantes e conhecidos, este representa as classes (entidades ou tipos) existentes, suas propriedades, as ações que cada uma delas pode realizar e as associações entre elas, bem como as restrições presentes nestas associações (OMG, 2001).

Classes e *Objetos* são os conceitos mais importantes na modelagem orientada a objeto. As *Classes* representam um conjunto de *Objetos* que possuem estrutura, comportamento e relacionamentos similares. *Classes* do modelo possuem um mapeamento claro com o mundo real como, por exemplo, carro, mesa, aeronave, pessoa, elevador, entre outras. Possuir um

mapeamento claro entre o modelo do domínio e o domínio real é muito importante tanto para a qualidade do modelo quanto para os processo de verificação.

Para exemplificar um elemento *Classe* no *Diagrama de Classes*, a Figura 10 demonstra a classe "*Pessoa*" com alguns atributos bem como algumas ações que esta classe pode realizar.

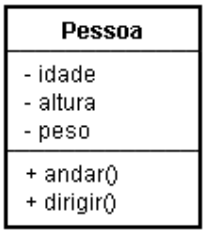


Figura 10 – Classe "Pessoa".

Qualquer relacionamento entre duas ou mais classes é uma associação. As associações criam relacionamentos semânticos entre as classes. Existem basicamente três tipos de associações nos *Diagramas de Classes*: Associação Simples, Agregação e Composição. As associações simples apenas conectam as classes identificando algum significado apropriado representado pelo nome da associação e eventualmente pela sua direção. Um exemplo de associação simples seria a associação "*esta-em*" entre as classes "*Pessoa*" e "*Lugar*". Agregação e Composição são casos especiais de associações que já trazem algum significado semântico. A associação de Agregação tem o significado "possui" ou "tem" como, por exemplo, um "*Prédio*" possui "*Elevadore*s". Já a associação Composição tem o significado "é feito de" como, por exemplo, um "*Prédio*" é feito de "*Andares*". É possível perceber, através do exemplo, que a associação de Composição representa uma dependência mais forte entre as classes, pois não existem prédios sem andares, mas existem sim prédios sem elevadores.

Um conceito muito útil no *Diagrama de Classes* é a Generalização. A Generalização contribui para uma melhor estruturação do diagrama com a utilização de heranças entre as classes. Por exemplo, as classes "*Shopping*" e "*Aeroporto*" são subclasse da classe "*Lugar*" e estas herdam todas as características da classe "*Lugar*", inclusive a associação com "*Pessoa*".

Um conceito interessante representado no *Diagrama de Classes* é a multiplicidade das associações. A multiplicidade indica e restringe como as classes, e conseqüentemente os objetos, se relacionam. É possível representar, por exemplo, que uma pessoa só pode estar em apenas um lugar (1) em um determinado momento e que em um lugar podem estar diversas pessoas ou nenhuma (0..*). Para isso usa-se a multiplicidade nas associações. A Figura 11 demonstra os conceitos de associação, generalização e multiplicidade apresentados.

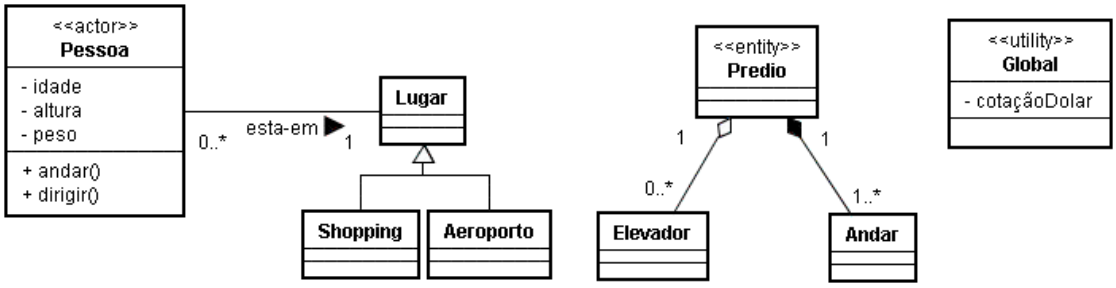


Figura 11 – Exemplo de associação entre classes, generalização e multiplicidades.

Este diagrama permite também a definição de grupos de elementos (classificação) com comportamentos similares no modelo. Este conceito é chamado de *Stereotype* e é muito útil, por exemplo, para distinguir elementos passivos e ativos no modelo. Na UML, já existem alguns *stereotypes* pré-definidos como, por exemplo: *actor* que classificada os atores do modelo, representando principalmente aqueles definidos no diagrama de casos de uso; *entity* que classifica os elementos passivos do modelo; *utility* que define um comportamento global no modelo, isto é, quando uma classe é definida com o *stereotype utility* seus atributos e métodos se tornam variáveis e procedimentos globais; entre outros. A Figura 11 também ilustra alguns exemplos do uso de *stereotypes*.

2.4.2.3. Diagrama de Estados

A UML possui alguns digramas que definem características dinâmicas das entidades existentes no modelo do sistema, tal como o *Diagrama de Estados*. Este diagrama pode ser utilizado para descrever o comportamento de instâncias das classes do domínio (os objetos) através da representação dos possíveis estados, das pré e pós-condições das ações que as afetam (D'SOUZA; WILLS, 1999). Cada *Diagrama de Estados* descreve as possíveis seqüências de estados nos quais um único objeto (de uma determinada classe) passa durante a sua vida no sistema em resposta a eventos e ações. Esta vida é baseada nos resultados das ações e das reações ocorridas. Basicamente, cada diagrama de estados representa o comportamento de uma determinada classe que, obviamente, possui comportamento (OMG, 2001).

O *Diagrama de Estados* é, de um modo geral, um grafo que representa uma máquina de estados. Os estados representam as situações nas quais um objeto pode se encontrar e as ações (arcos no grafo) representam a transição de um estado a outro (OMG, 2001). De acordo com (D'SOUZA; WILLS, 1999) é possível representar as pré e pós-condições das ações no próprio diagrama, como mostra a Figura 12. Esta figura exemplifica um diagrama de estados de uma classe onde os conceitos de estados, ações, pré e pós-condição são visualizados. É interessante notar que a pré-condição de uma ação no diagrama é formada pela descrição do estado (*Estado1*) e pela descrição na ação (*precond*), ocorrendo o mesmo para a pós-condição (*poscond* e *Estado2* - efeitos). É possível representar também os estados Inicial e Final do objeto para representar o início e o fim de sua vida no sistema. O Estado Inicial do objeto é representado por um círculo preto totalmente preenchido, já o Estado Final é representado por um círculo não totalmente preenchido como mostra a figura a seguir.

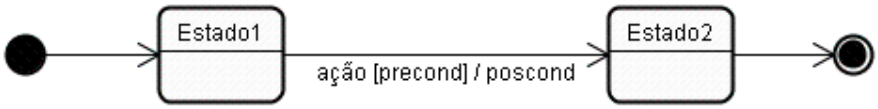


Figura 12 – Exemplo simples de Diagrama de Estados.

2.4.2.4. Diagrama de Objetos ou *Snapshot*

O *Diagrama de Objetos* representa uma situação, um estado, ou momento, do domínio. Este diagrama é também chamado de *Snapshot* e é, de fato, uma instância do diagrama de classes. O *Snaphshot* permite o usuário instanciar classes (criando assim objetos), dar valores aos atributos de cada instância das classes e associar os objetos de acordo com a situação que o projetista deseja construir. Basicamente, o *Snapshot* é uma foto de um estado do sistema em um momento específico. Uma seqüência de diagrama de objetos nos dá a evolução do sistema (no tempo se for o caso), formando um "filme" da evolução do sistema. É importante lembrar que os valores e associações realizadas no *Diagrama de Objetos* devem respeitar as restrições definidas no *Diagrama de Classes* como respeito, por exemplo, a multiplicidade. Um exemplo de *Diagrama de Objetos* é mostrado na Figura 13 seguindo o mesmo exemplo do *Diagrama de Classes* anteriormente apresentado.

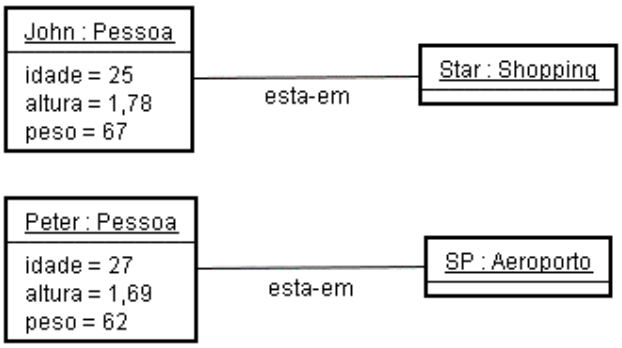


Figura 13 – Exemplo do Diagrama de Estados – *Snapshot*.

2.4.2.5. OCL – *Object Constraint Language*

Um diagrama UML, tal como o *Diagrama de Classe* ou *de Estados*, geralmente não é refinado o suficiente para prover todos os aspectos relevantes de uma modelagem, ou eventualmente de uma especificação. Existe, entre outras coisas, a necessidade de expressar restrições adicionais sobre os objetos no modelo sendo *desenvolvido*. Tais restrições são geralmente descritas em linguagem natural. As práticas na especificação e na modelagem mostram que o uso de linguagem natural sempre causa ambigüidade. Com o objetivo de evitar ambigüidades foi desenvolvida a linguagem formal OCL – *Object Constraint Language* (OMG, 2003) - para especificar restrições. A OCL é uma linguagem de especificação de restrições fácil de ler e entender favorecendo a modelagem de negócios e sistemas.

A OCL é muito similar a lógica de predicados de primeira ordem. As restrições são geralmente expressões booleanas, que podem ser utilizadas com conectores lógicos, que devem ser verdade durante todo o ciclo de vida do objeto no sistema. As expressões de restrições podem complementar a descrição das características e dos comportamentos dos objetos no sistema. É possível representar expressões sobre classes associações, estados, ações, diagramas, entre outros.

A OCL pode ser utilizada para diferentes propósitos como, por exemplo:

- para especificar invariantes sobre as classes e elementos do modelo;
- para especificar pré e pós-condições das ações/operadores realizadas pelos objetos;
- para descrever restrições de estados do domínio;
- ser utilizada como linguagem de navegação (acesso) entre classes;
- para especificar restrições sobre a ocorrência de operadores.

Para exemplificar algumas expressões em OCL sobre um *Diagrama de Classes* a Figura 14 fornece um exemplo deste tipo de diagrama para que algumas expressões sejam apresentadas.

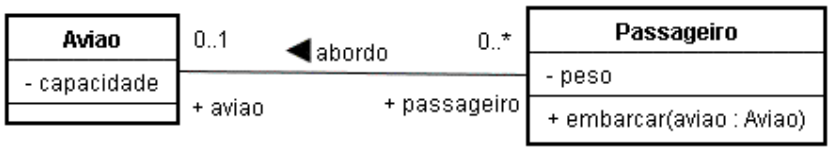


Figura 14 – Exemplo de Diagrama Classes.

Além das restrições introduzidas pelas multiplicidades entre as classes "*Aviao*" e "*Passageiro*" podemos adicionar a seguinte restrição utilizando a OCL: "em todo e qualquer estado do domínio a *capacidade* de peso do *Aviao* deve ser sempre maior ou igual que a somatória dos *pesos* dos *passageiros* que estão *abordo*". Esta invariante do domínio pode ser declarada da seguinte forma em OCL:

```
context Aviao inv:
   self.capacidade  >=  self.passageiro.peso->sum()
   -- qualquer que seja o Aviao a somatória dos pesos de todos os
      passageiros que estão a bordo do avião deve ser menor ou igual
      que a capacidade do Aviao. Observação: a palavra reservada
      "self" representa um elemento da classe Aviao.
```

Utilizando o mesmo exemplo da Figura 14 é possível declarar as pré e pós-condições da ação *embarcar*. Uma pré-condição seria afirmar que para o passageiro embarcar no avião seria preciso verificar de se o seu peso não excede a capacidade de carga do avião. Uma pós-condição para esta ação, caso a pré-condição seja estabelecida, seria afirmar que passageiro estará abordo do avião. Estas condições podem ser declaradas em OCL da seguinte maneira:

```
context Passageiro::embarcar(aviao : Aviao)
   pre:
         aviao.capacidade  >=  aviao.passageiro.peso->sum() + self.peso
   post:
         self.abordo = aviao
```

A OCL se mostra muito útil na fase de modelagem para a área de Planejamento Automático já que esta é uma linguagem formal de especificação de restrições. Restrições como invariantes são muito úteis durante o processo de planejamento para que os planejadores evitem buscas exaustivas em caminhos que não chegam a soluções plausíveis.

As restrições podem, de fato, enriquecer o modelo do domínio fazendo com que as técnicas de planejamento possam usufruir desta informação. É importante citar que muitas das capacidades de expressão das recentes evoluções da PDDL já estavam presentes na OCL.

2.4.2.6. Discussões sobre a UML

A UML está em constante evolução agregando cada vez mais experiências em modelagem, principalmente aquelas orientadas a objetos. A UML acaba sendo uma ótima linguagem para acompanhar todo o processo de especificação e design de domínios. O presente trabalho contempla o uso da UML para a análise de requisitos e a modelagem de domínios de planejamento, principalmente a modelagem da estrutura estática dos mesmos.

O uso da UML não se faz suficiente para contemplar todo o processo de design de um domínio. Os diversos diagramas construídos durante a especificação e modelagem devem ser verificados, principalmente no quesito consistência já que não existem mecanismos de verificação deste aspecto. Por um lado a UML não restringe o projetista com soluções enviesadas e nem com níveis de detalhamentos específicos, mas por outro a linguagem não possibilita uma visão unificada de todos os diagramas, principalmente nos aspecto dinâmico onde é possível desenvolver vários diagramas (diagrama de estados, de atividade, de

seqüência e o colaborativo) para descrever os mesmos comportamentos com pontos de vista diferentes.

A definição da estrutura estática de um modelo em UML não sofre tanto com múltiplos pontos de vista, mas como todos os diagramas refletem um mesmo domínio esta estaria presente também em um processo de verificação de consistência.

A utilização da UML como linguagem de especificação e modelagem de domínio de planejamento é proposta neste trabalho devido, principalmente, a necessidade de se utilizar ferramentas que possibilitem os processos de especificação e modelagem de domínios reais e complexos. Estes processos envolvem vários níveis de abstração do modelo e uma visão unificada do mesmo proveniente de todos os participantes no processo de design como, por exemplo, o projetista, o especialista em planejamento, o especialista no domínio, os usuários, os *stakeholders*, entre outros.

Na abordagem proposta neste trabalho a UML se mostra uma linguagem excelente para especificação e modelagem de domínios de planejamento com uma abordagem orientada a objetos. Os conceitos de classes, propriedades, estados, ações, efeitos e pré-condições são conceitos intrínsecos à área de Planejamento Automático e na verdade intrínsecos a maioria dos grandes sistemas e domínios, não só os de planejamento. Mesmo para os domínios de planejamento o uso da UML deve ser complementado com alguns mecanismos de suporte ao projetista para que este possa, por exemplo, verificar a consistência dos diagrama gerados. Para isso a utilização de outras ferramentas que contemplam sistemas dinâmicos, tais como Redes de Petri, fazem-se necessárias para contribuir na realização de um processo correto de verificação e validação de modelos de domínio.

2.4.3. Redes de Petri – *Petri Nets*

2.4.3.1. Conceitos Básicos

O conceito de Redes de Petri (RdP) foi introduzido por Carl Adam Petri em sua tese de doutorado (1962) como um formalismo matemático para descrever relações entre condições e eventos no estudo de protocolos de comunicação entre componentes assíncronos em computadores. Embora ocorresse uma ampla divulgação acadêmica deste conceito ao longo de três décadas, o potencial só foi reconhecido na metade da década de oitenta, onde esta teoria foi usada para implementações práticas nas áreas de informática e manutenção devido à disponibilidade de recursos mais poderosos de hardware e software.

A Rede de Petri é um formalismo matemático que possui uma representação tanto algébrica quanto gráfica oferecendo uma abordagem uniforme para modelagem, análise e projeto de sistemas a eventos discretos. A RdP é, de fato, uma representação fundamental e extremamente efetiva para a modelagem de sistemas. A sua base teórica permite o desenvolvimento de poderosas ferramentas de análise e síntese de estratégia de controle.

Em geral, utilizam-se as Redes de Petri para modelagem conceitual da estrutura dinâmica do sistema. Sua aplicação tem se estendido a uma grande quantidade e variedade de sistemas como, por exemplo, sistemas de comunicação, sistemas de software, simulação, processos seqüenciais em sistemas flexíveis de manufatura, entre outros.

De um modo geral, as Redes de Petri são grafos compostos de: nós *lugares* (*P*) que graficamente são representados por círculos; nós *transições* (*T*) que graficamente são representadas por barras; *arcos* que conectam *lugares* a *transições* e *transições* a *lugares* (os arcos nunca conectam *lugares* a *lugares* ou *transições* a *transições*) e estes podem receber pesos (1,2,3,....); e marcas que são utilizadas para definir o estado da rede. A RdP é considerada um *multigrafo* (já que é permitido múltiplos arcos direcionados saírem de um nó

e irem até outros seguindo as restrições de ligação) *direcionado* (pois os arcos possuem orientações definidas) *bipartido* (já que os nós são particionados em dois conjuntos, *lugares* e *transições*, e os arcos conectam necessariamente elementos de conjuntos distintos) (MURATA, 1989). Um exemplo simples da representação gráfica de RdP é ilustrado na Figura 15 onde todos os arcos possuem peso 1.

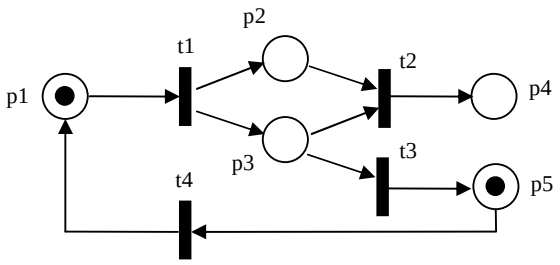


Figura 15 - Exemplo de uma Rede de Petri.

A execução da Rede de Petri é representada pelos disparos das transições presentes na rede. Estes disparos são definidos pela distribuição das marcas na rede. As *marcas* localizam-se nos *lugares* e condicionam o disparo das *transições.* A condição necessária para que haja um disparo de uma *transição* é a existência de *marcas* em todos os *lugares* de entrada (*pré-condições*). Ocorrendo o disparo, as *marcas* são removidas dos *lugares* de entrada e são criadas novas *marcas* nos *lugares* de saída (*pós-condições*) daquela *transição* (MURATA, 1989). A Figura 16 ilustra o disparo de uma transição onde (a) representa o momento antes do disparo e (b) representa o momento após o disparo da *transição*.

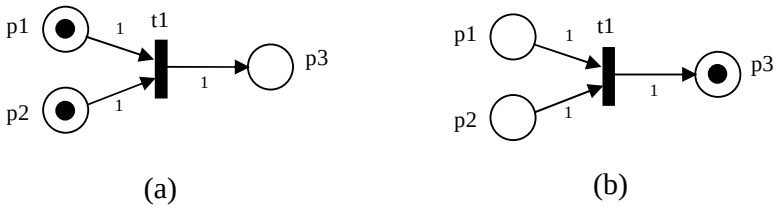


Figura 16 - Exemplo de um disparo de uma transição na Rede de Petri.

2.4.3.2. Definição Formal de uma Rede de Petri

De acordo com Murata (1989) é uma 5-tupla, $PN = (P, T, F, W, M_0)$ onde:

- $P = \{p_1, p_2, \ldots, p_m\}$ é um conjunto finito de lugares,
- $T = \{t_1, t_2, \ldots, t_n\}$ é um conjunto finito de transições,
- $F \subseteq (P \times T) \cup (T \times P)$ é o conjunto de arcos (relação de fluxo),
- $W: F \rightarrow \{1, 2, 3, \ldots\}$ é a função de peso,
- $M_0: P \rightarrow \{0, 1, 2, 3, \ldots\}$ é a marcação inicial da rede.

Uma estrutura de uma Rede de Petri $N = (P, T, F, W)$ sem nenhuma marcação inicial especifica é denominada *N*.

Uma Rede de Petri com uma marcação inicial é denominada por (N, M_0).

A marcação da uma Rede de Petri (ou seja, o estado da rede) muda de acordo com as seguintes regras de disparo das transições:

1) Uma transição *t* é dada como *habilitada* se cada lugar de entrada *p* de *t* está marcado com pelo menos $w(p,t)$ marcas, onde $w(p,t)$ é o peso do arco de *p* para *t*.
2) Uma transição *habilitada* pode ou não ser disparada (dependendo se o evento realmente ocorre ou não).
3) Um disparo de uma transição t remove $w(p,t)$ marcas de cada lugar de entrada *p* de *t*, e adiciona $w(t,p)$ marcas para cada lugar de saída *p* de *t*, onde $w(t,p)$ é o peso do arco de *t* para *p*.

2.4.3.3. Principais Propriedades das Redes de Petri

A RdP pode apresentar diversas propriedades (MURATA, 1989). As principais propriedades de uma Rede de Petri, representada por um conjunto *P* de *lugares*, um conjunto *T* de *transições*, um conjunto de arcos *F* (relação de fluxo) que associam os elementos de *P* e *T*, e M_0 a marcação inicial da rede, são apresentadas a seguir:

- ***,Dinâmica:*** depende da marcação inicial M_0. É possível o uso de grafos para analisar as redes;
- ***Estrutural:*** a rede depende de sua estrutura topológica e não de sua marcação inicial M_0.
- ***Alcançabilidade (Reachability):*** sendo a marcação inicial M_0 de uma rede *N*, uma marcação M_n é *alcançável* a partir de M_0 se existe uma seqüência de disparos que transformam M_0 em M_n. Esta propriedade é bastante utilizada para verificação das propriedades dinâmicas das redes onde são analisados todos os conjuntos de marcações M_n para alcançar um determinado estado de marcação desejado da rede. Existem algoritmos que respondem à seguinte pergunta: "M_n é alcançável a partir de M_0?" Contudo, estes algoritmos são, no mínimo, NP-hard.
- ***Limite de Marcas (Boundness):*** uma rede $PN = (P, T, F, W, M_0)$ é *k-limitada* se o número de marcas em cada lugar *p* jamais excede *k*, considerando todos os estados da rede alcançáveis a partir de M_0. Análises podem ser efetuadas em relação ao número de marcas em uma determinada marcações da rede.
- ***Vivacidade:*** uma rede $PN = (P, T, F, W, M_0)$ é viva se, em qualquer marcação M_n alcançável a partir de M_0, qualquer transição poderá ser disparada em M_n ou em marcações alcançáveis a partir de M_n.
- ***Reversibilidade:*** seja R(M_0) o conjunto de estados de uma rede *PN* alcançáveis a partir da marcação M_0. Esta rede será reversível se M_0 for alcançável por qualquer M_n pertencente a R(M_0), ou seja, é sempre possível retornar ao estado inicial da rede.
- ***Cobertura:*** uma marcação *M* na rede *PN* é dita coberta se existe *M'* em R(M_0) tal que $M'(p) \geq M(p)$ para cada *p* pertencente ao conjunto *P*, onde $M(p)$ é o número de marcas no lugar *p* na marcação *M*.

- ***Paralelismo estrutural:*** é quando há na rede *transições* paralelas ou concorrentes sem possuírem um *lugar* de entrada em comum;
- ***Conflito estrutural:*** é quando as *transições* com *lugares* de entrada em comum estão em disputa não determinística de recursos (*marcas*);
- ***Assincronismo:*** é quando um *lugar* possui mais de uma *marca*, e estas estão disponíveis, mas somente uma pode ser disparada pela *transição*. É necessária uma ordem parcial para a passagem das *marcas*.
- ***Hierarquia:*** é a substituição simbólica de uma região da rede por apenas um *lugar*, ou transição;

Com propriedades como estas apresentadas acima é possível efetuar análises, verificações e validações de redes através de ferramentas e algoritmos de uma forma sistemática.

2.4.3.4. Extensões das Redes de Petri

Algumas extensões das Redes de Petri são encontradas na literatura. Algumas dessas extensões são:

- **Redes Ordinárias**: onde os *lugares* possuem capacidade ilimitada de *marcas* e os arcos possuem peso 1 (MURATA, 1989);
- **Rede Lugar/Transição**: que ao contrário das ordinárias possui capacidade finita de marcas nos *lugares* (MURATA, 1989);
- **Redes Condição/Evento**: neste tipo de rede cada uma das marcas representa uma condição no modelo. A presença de uma marca na rede representa o vigorar da correspondente condição, enquanto que a ausência representa o não vigorar da correspondente condição. Nesta rede existem regras que condicionam a habilitação de uma transição e regras que definem o efeito de uma transição. A regra principal para a

ocorrência de um disparo de uma *transição* é que os *lugares* de entrada desta *transição* estejam condicionados (com marcas) e os *lugares* de saída (pós-condições) não (PETERSON, 1981);

- **Redes Predicado/Transição**: estas redes combinam o comportamento dinâmico da rede com a forte expressividade de predicados lógicos (primeira ordem) onde os lugares são associados a valores verdade (GENRICH; LAUTENBACH, 1981);
- **Redes de Marcas Individuais (Redes Coloridas)**: onde as *marcas* são individualizadas fazendo com que as *transições* se comportem de acordo com o tipo das *marcas* que se encontram, por exemplo, nos *lugares* de entrada (pré-condições) (JENSEN; 1981);
- **Redes Orientadas a Objetos**: nestas redes as *marcas* são objetos que possuem atributos com valores individualizados. Os valores destes atributos podem ser modificados durante os disparos das *transições* (VALK, 2004);
- **Redes Temporizadas**: onde as *transições* possuem um parâmetro que representa o tempo do disparo (RAMCHANDANI, 1974) (MERLIN, 1974) (MURATA, 1989).

2.4.3.5. Visão Geral das Principais Características das Redes de Petri

A RdP é uma técnica extremamente efetiva para a modelagem de sistemas. Algumas características das RdP são extremamente importantes na modelagem de domínios dinâmicos. Algumas das principais características das Redes de Petri são:

- Expressividade na representação de diferentes tipos de sistemas e domínios;
- Representa características estáticas e dinâmicas;
- Possui formalização matemática;

- Possui representação gráfica que facilita o entendimento do funcionamento do modelo do sistema;
- Permite representar, de modo natural, conceitos como sincronização de processos, paralelismo, eventos concorrentes, causalidade, compartilhamento de recursos e conflitos;
- Interpretabilidade, pois os elementos das redes podem ser entendidos de diversas formas. Por exemplo, os *lugares* podem ser entendidos como *estados, pré-condições*, *pós-condições, sinais de entrada e saída*, entre outros. Já as *transições* podem ser entendidas também por *ações, eventos, passos de um processamento*, etc;
- Permite identificar clara e explicitamente estados e ações;
- Possibilita representar diferentes níveis de abstração na modelagem dos domínios;
- Possui métodos de modelagem e ferramentas de análise e visualização das redes disponíveis na literatura;
- Permite simulações;
- É um formalismo que permite extensões;
- É possível considerar regras temporais nas *transições*;
- É possível associar regras de decisões nos *lugares* para casos onde haja conflito de *transições*;
- É possível modelar redes orientadas a objeto;
- É possível considerar outros tipos de *arco* como, por exemplo, arco inibidor ou habilitador de *transição*.

2.4.3.6. Redes de Petri e Planejamento Automático

Algumas características das Redes de Petri são muito atrativas para área do Planejamento Automático, tanto para a modelagem e verificação de domínio quanto para o desenvolvimento

de técnicas de planejamento. Neste projeto, o formalismo das RdP é aplicado na análise dinâmica dos domínios de planejamento. Alguns fatores das RdP que fundamentam esta utilização em planejamento são:

- Representa a dinâmica e a estrutura de um sistema segundo o nível de detalhamento desejado (sendo uma rede hierárquica);
- Identifica estados e ações de modo claro e explícito;
- Tem a capacidade de representar de forma natural as características dos Sistemas a Eventos Discretos SED´s (sincronização, assincronismo, concorrência, causalidade, conflito, compartilhamento de recursos, etc.) muito importante na formulação de um plano;
- Os conceitos envolvidos na representação de um plano são perfeitamente expressos em RdP. Por exemplo, uma característica fundamental existente em um plano é a seqüência de ações (ou eventualmente uma rede de planos alternativos), e esta característica é claramente representada pelas RdP. Podemos citar outras características de um plano como o paralelismo, ou a sincronização de ações;
- Oferece um formalismo gráfico que permite a documentação, visualização, verificação e monitoração do sistema, facilitando assim o diálogo entre o modelo e projetista que analisa o comportamento dinâmico do mesmo;
- Pode prover uma visão unificada e geral do comportamento dinâmicos do modelo do domínio de planejamento;
- Possui uma semântica formal e precisa que permita que o mesmo modelo possa ser utilizado, por exemplo, para análise de propriedades, avaliação do desempenho, bem como para a construção de simuladores a eventos discretos e controladores (para implementar ou gerar códigos para controle de sistemas). A rede também pode servir para verificar comportamentos indesejáveis como bloqueio, limitação, entre outros;

- Incorpora conceitos de modelagem por refinamento ("*top down*") e do tipo composição modular ("*bottom up*") através de técnicas como: modularização, reutilização, refinamento;
- Redes de Petri possuem uma ótima capacidade de lidar com recursos. Muitos domínios reais de planejamento possuem os conceitos de recursos e, de fato, a habilidade de lidar com recursos é muito procurada pela comunidade de Inteligência Artificial que está muito embasada na nas representações lógicas;
- Suas propriedades podem ser utilizadas para realizar análises e verificações sobre o domínio de planejamento;
- Muitos domínios de planejamento são modelados utilizando redes hierárquicas de ações (ou tarefas) e as RdP possibilitam estas representações;
- Algumas propriedades e informações dos modelos em RdP podem contribuir para o processo de planejamento (ex.: alcançabilidade), ou seja, podem ajudar o planejador a evitar buscas indesejáveis;

2.4.4. XML – *eXtensible Markup Language*

A XML – *eXtensible Markup Language* (BRAY et al., 2004) - foi criada pela WWW Consortium em 1996 e sua primeira versão disponibilizada em 1998. Essa linguagem nasceu a partir da SGML (*Standard Generalized Markup Language*) com os objetivos de se criar uma meta-linguagem capaz de adicionar semânticas para uma linguagem muito utilizada, o HTML, e para melhorar a flexibilidade de intercambio de documentos e informação na Internet.

A XML é uma linguagem que descreve dados estruturados bastante útil nas declarações de conteúdo e na obtenção de resultados mais significativos de busca através de ambientes de

múltiplas plataformas. Isto faz com que a XML seja uma linguagem apropriada para troca de dados e informação entre sistemas de informação (BRAY et al., 2004).

Esta linguagem é baseada no conceito de *tags* (marcações) que auxiliam na estruturação dos dados e da própria representação. Utilizando a XML é possível declarar dados com grande flexibilidade como, por exemplo, preços de livros, tipos de transporte, taxas comerciais, e qualquer outro tipo de dado. Com o uso das *tags* é possível criar estruturas específicas para cada tipo de aplicação desde que todos os sistemas que irão utilizar a estrutura saibam interpretá-la. Um pequeno exemplo de dados estruturados em XML para uma simples aplicação de armazenamento de dados de livros é apresentado a seguir:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<eBooks>
      <eBook id=”1”>
            <name>Aprendendo XML</name>
            <author>John</author>
            <year>2006</year>
      </eBook>
      <eBook id=”2”>
            <name>Planejamento Automático</name>
            <author>Peter</author>
            <year>2005</year>
      </eBook>
</eBooks>
```

Esta linguagem vem sendo muito utilizada por diversas ferramentas para armazenamento e transferência de dados. Muitos softwares utilizam a XML para representar os seus arquivos de armazenamento de informação como, por exemplo, modelos de banco de dados, modelos de Rede de Petri (através da chamada PNML – *Petri Nets Markup Language* (WEBER; KINDLER, 2002)), UML, PDDL (através da XPDDL (GOUGH, 2004)) entre outros. Isto se deve ao fato desta linguagem ser facilmente manipulada em nível de programação e também

pela grande evolução de tecnologias associadas a esta linguagem (por exemplo, técnicas de busca de informação em arquivos XML).

2.4.5. XPDDL – *eXtensible Planning Domain Definition Language*

A XPDDL – *eXtensible Planning Domain Definition Language* (GOUGH, 2004) - consiste num formato XML para representação de modelos de planejamento baseada na PDDL, ou seja, a representação da PDDL utilizando o conceito de *tags*. A XPDDL foi criada pelo pesquisador Gough em 2004 com o objetivo de substituir a linguagem PDDL por uma linguagem de base XML devido a suas vantagens. Mesmo estando ainda nos estágios iniciais de desenvolvimento (versão 0.1), os benefícios da utilização de uma linguagem de base XML favorecem o uso desta linguagem em ferramentas e ambientes onde há troca de informação bem como processos de tradução. A definição desta linguagem pode ser encontrada em (GOUGH, 2004).

A XPPDL0.1 (GOUGH, 2004) suporta apenas as características da PDDL2.1 e PDDL2.2, mas tende a evoluir, e suas características são muito similares a PDDL. A tabela a seguir mostra uma pequena seção de código em PDDL e o seu correlato em XPDDL.

Tabela 1 - Exemplo comparativo de representação entre a PDDL e a XPDDL.

PDDL	XPDDL
precondition: (not (exists (?a1 - airplane) (and (not (= ?a1 ?a)) (blocked ?s2 ?a1))))	<precondition> <not> <exists> <parameter name="a1" type="airplane"/> <suchthat> <not> <equals> <parameter id="a1"/> <parameter id="a"/> </equals> </not> <predicate id="blocked"> <parameter id="s2"/> <parameter id="a1"/> </predicate> </suchthat> </exists> </not> </precondition>

Neste trabalho a XPDDL é vista como uma linguagem estruturada auxiliar que contribui na passagem de informação e dos modelos ao planejador.

2.4.6. PNML – *Petri Nets Markup Language*

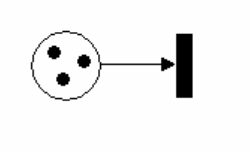
A PNML - *Petri Net Markup Language* (WEBER; KINDLER, 2002) - consiste num formato XML para representação de Redes de Petri. Este formato nasceu da necessidade de realizar transferências de modelos das redes entre as ferramentas disponíveis em diferentes países do mundo. Com uma representação única seria possível exportar e importar os modelos em outras ferramentas e conseqüentemente usufruir o potencial e as funcionalidades de cada uma delas como, por exemplo, análise, verificação, validação, simulação e implementação de redes.

A PNML foi projetada para ser o formato de transferência de modelos independente de ferramenta ou plataforma específica. O desenvolvimento desta representação foi guiado por objetivos como: ser legível e facilmente modificada tanto pelo ser humano quanto pela

máquina (característica garantida pela XML); ter a possibilidade de representar qualquer versão e tipo de RdP com qualquer tipo de extensão, ou seja, ser flexível; facilitar a realização de grandes projetos com muitas equipes envolvidas favorecendo a troca de informação; e estimular a criação de bibliotecas de modelos comuns e bem conhecidos entre outros.

Este formato tem sido adotado por muitas ferramentas para especificação, verificação e visualização de RdP, sendo muito útil na migração de redes entre ferramentas e para a reusabilidade de modelos. Hoje a PNML é suficientemente geral para representar todas as versões e tipos de Redes de Petri, pois a mesma pode distinguir as características gerais e específicas de um determinado tipo de redes. A estrutura da linguagem PNML e sua definição podem ser encontradas em (WEBER; KINDLER, 2002).

Para exemplificar a estrutura da linguagem, a Figura 17 a seguir demonstra a representação gráfica de um *lugar* com três *marcas* atrelado a um *arco,* que o relaciona a uma *transição,* e sua respectiva representação correspondente em PNML.



```
<pnml xmlns="http://www.example.org/pnml">
      <net id="n1" type="http://www.example.org/pnml/PTNet">
            <name>
                  <text>An example P/T-net</text>
            </name>
            <place id="p1">
                  <graphics>
                        <position x="20" y="20"/>
                  </graphics>
                  <initialMarking>
                        <text>3</text>
                  </initialMarking>
            </place>
            <transition id="t1">
                  <graphics>
                        <position x="60" y="20"/>
                  </graphics>
                  <toolspecific tool=" itSIMPLE" version="0.1">
                        <hidden/>
                  </toolspecific>
            </transition>
            <arc id="a1" source="p1" target="t1">
                  <graphics>
                        <position x="30" y="5"/>
                        <position x="60" y="5"/>
                  </graphics>
            </arc>
      </net>
</pnml>
```

Figura 17 - Exemplo da Representação PNML de uma Rede de Petri simples.

Uma questão importante levada em consideração pelos autores da linguagem PNML (WEBER; KINDLER, 2002) foi quanto a estruturação de redes muito grandes. Algumas abordagens da PNML fornecem alternativas para facilitar a visualização destas redes, bem como o desenvolvimento e reutilização de redes de grande porte. Um exemplo é a definição e instanciação de *módulos*. Está abordagem é chamada de *PNML modular* (KINDLER; WEBER, 2001), uma extensão da PNML que permite a construção de RdP de uma maneira modular. Utilizando o conceito de módulos, uma rede (um módulo) pode ser definida como um objeto de alto-nível, que pode ser instanciado uma ou várias vezes dentro de outra rede.

A *PNML modular* é também independente de qualquer tipo de Rede de Petri. Embora simples e sem uma teoria profunda, esse conceito provê meios poderosos de construir RdP. A *PNML modular* possui uma semântica clara, que é também independente de um tipo particular de rede, e pode ser encontrada em (KINDLER; WEBER, 2001). Esta abordagem foi criada, principalmente, pela falta de um conceito apropriado de modularidade nas Redes de Petri.

Para ilustrar a idéia básica da *PNML modular*, bem como o conceito de modularidade nas redes, será utilizado um exemplo simples, onde o tipo da RdP é o lugar/transição. A Figura 18 define o módulo M1. Este módulo possui dois lugares *x* e *y*, duas transições *t1* e *t2* e alguns arcos.

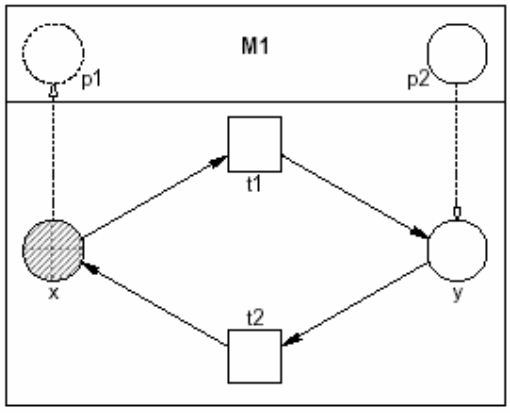


Figura 18 – Um módulo M1. Fonte: (KINDLER; WEBER, 2001).

Os lugares *x* e *y*, as transições *t1* e *t2* e os arcos representam uma implementação interna do módulo. Este fato faz com que os elementos não sejam acessíveis fora do módulo (no ambiente). Para dar ao ambiente acesso a algum elemento interno do módulo M1, é definido uma *interface* no módulo. No exemplo do módulo M1 na Figura 18 a *interface* é constituída de um lugar *p1*, que é importado do ambiente do módulo (externo ao módulo), e o lugar *p2*, que é *exportado* para o ambiente do módulo. O lugar importado *p1* é um parâmetro do

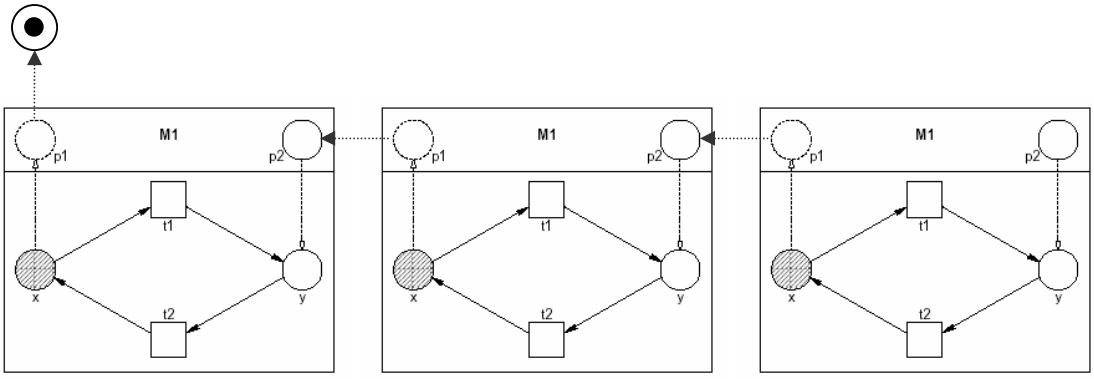
módulo, que deve ser fornecido quando o módulo é instanciado (mostrado a seguir), e é representado por um circulo pontilhado. O lugar exportado *p2* pode ser usado no ambiente (por exemplo, por outro do modulo instanciado), e este é representado por um circulo cheio. A interface e a implementação do módulo são relacionados por referências. No exemplo, o lugar *x* referência o lugar importado *p1*, que é representado por uma seta pontilhada de *x* para *p1*. Desta forma, o lugar *x* é um representativo do lugar que será fornecido como parâmetro quando o módulo M1 for instanciado. Já o lugar exportado *p2*, referencia o lugar *y*. Desta forma, uma referência para o lugar *p2* de uma instância do módulo, na verdade, refere-se ao lugar *y* (KINDLER; WEBER, 2001). Para visualizar estes conceitos a Figura 19 mostra o código PNML do módulo de exemplo M1 e a Figura 20 fornece um exemplo de instanciação dos módulos para a criação de uma Rede de Petri que será elaborado a seguir.

```
<module name="M1">
  <interface>
    <importPlace id="p1"/>
    <exportPlace id="p2" ref="y"/>
  </interface>
  <referencePlace id="x" ref="p1"/>
  <transition id="t1"/>
  <transition id="t2"/>
  <place id="y"/>
  <arc source="x" target="t1"/>
  <arc source="t1" target="y"/>
  <arc source="y" target="t2"/>
  <arc source="t2" target="x"/>
</module>
```

Figura 19 – O código PNML do módulo M1. Fonte: (KINDLER; WEBER, 2001).

É possível criar uma rede a partir de várias instâncias do módulo M1. A Figura 20 mostra uma representação gráfica e uma representação textual (não representando nenhuma

linguagem de programação) de uma rede n1 com três instâncias do módulo exemplo M1, que são nomeados de `m1`, `m2` e `m3`, respectivamente.




```
def net n1:
{    place p: 1;
     m1 = instace M1(p1 = p);
     m2 = instace M1(p1 = m1.p2);
     m1 = instace M1(p1 = m2.p2);
}
```

Figura 20 – Uma rede n1 construída a partir de três instâncias do módulo M1 (KINDLER; WEBER, 2001).

Na definição da rede n1, é definido primeiramente o lugar `p` com marcação inicial 1. Logo depois, são definidas três instâncias de M1. A primeira `m1` utiliza o lugar `p` anteriormente definido como sendo o parâmetro `p1`. A segunda instância utiliza o lugar exportado `p2` de `m1` (denominado `m1.p2`) como sendo o parâmetro para `p1`. Da mesma forma, a terceira instância utiliza `m2.p2` como parâmetro que define `p1` (KINDLER; WEBER, 2001). Desta forma, a rede n1 fornece uma Rede de Petri lugar transição mostrada na Figura 21. A semântica desta definição de redes a partir dos módulos pode ser encontrada em (KINDLER; WEBER, 2001) onde importação e exportação de transições também é possível.

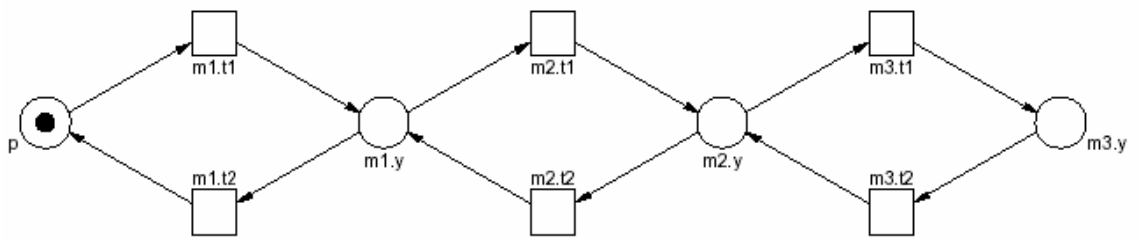


Figura 21 – A Rede de Petri n1. Fonte: (KINDLER; WEBER, 2001).

Na Figura 21, os nomes de cada lugar e transição, nas diferentes instâncias, são definidos pelos seus respectivos módulos para evitar nomes duplicados. É interessante notar que os lugares internos importados foram fundidos (KINDLER; WEBER, 2001) formando uma rede normal lugar/transição. A representação da rede n1 da Figura 21 em *PNML modular* é apresentada na Figura 22 a seguir.

```
<net id="n1">
  <place id="p">
    <initialMarking>
      <value>1</value>
    </initialMarking>
  </place>
  <instance id="m1" ref=URI#M1>
    <importPlace parameter="p1" ref="p"/>
  </instance>
  <instance id="m2" ref=URI#M1>
    <importPlace parameter="p1" instance="m1" ref="p2"/>
  </instance>
  <instance id="m3" ref=URI#M1>
    <importPlace parameter="p1" instance="m2" ref="p2"/>
  </instance>
</net>
```

Figura 22 – Código PNML da Figura 21. Fonte: (KINDLER; WEBER, 2001).

Neste trabalho a PDDL e a *PNML modular* são vistas como linguagens estruturadas auxiliares que contribuem na análise das características dinâmicas do modelo do domínio de planejamento realizada em Redes de Petri.

Capítulo 3

3. Proposta de Ambiente Integrado de Modelagem e Análise de Domínios de Planejamento Automático

Neste trabalho de pesquisa, os conceitos apresentados no Capitulo 2 foram estudados de forma a levantar aspectos e características de um ambiente de modelagem e análise inseridos em um ciclo de vida de projeto de um domínio de planejamento automático. O objetivo foi estudar a classe de problemas de engenharia que são caracterizados pela necessidade do uso do planejamento automático, e as técnicas existentes para modelá-los e analisá-los. Para tal foram estudados métodos, linguagens e ferramentas que contribuíssem para os processos de design de um sistema deste tipo.

Durante muitos anos os domínios e os problemas de planejamento modelados e analisados pelos pesquisadores da área eram considerados clássicos, ou seja, os domínios possuíam diversas simplificações de modo a dispensar processos rigorosos de especificação, modelagem e análise. Estes domínios e problemas clássicos possuíam um “enunciado” bem definido (fechado) possibilitando assim representá-los diretamente em linguagens, como a PDDL, que muitas vezes não propiciam uma visão geral clara do sistema, e/ou não disponibilizam mecanismos eficientes de análise de requisitos, verificação e validação de modelos, e/ou são mais próximas de linguagem de máquina a ser interpretada por sistema de planejamento. Como os estudos e os desenvolvimentos de técnicas de planejamento se concentravam basicamente nos problemas clássicos, não existia aparentemente a necessidade

de métodos disciplinados de especificações, modelagem e análise, muito menos de um ambiente que integrasse esses métodos.

A partir do momento que a comunidade de pesquisa em Planejamento Automático passou a ter como objetivo comum abordar problemas reais de engenharia, a utilização de fundamentos e conceitos práticos aplicados no projeto de sistemas reais passou a ser, além de necessário, fundamental para a evolução das ferramentas, bem como um melhor entendimento destes problemas reais. Alguns trabalhos relacionados à modelagem e análise de domínios de planejamento são encontrados na literatura, mas pode-se dizer que poucos contemplavam um ambiente integrado para realização de diversos processos iniciais de design inclusos em um ciclo de vida de projeto. Baseado nestes fatos, a proposta de um ambiente integrado (chamado aqui simplesmente de *Ambiente*) de modelagem e análise de domínio de planejamento automático, apresentado neste capítulo, inserido no contexto de ciclo de vida de projeto, além de favorecer as boas práticas já presentes no desenvolvimento de software, sistemas de informação e sistemas de engenharia em geral, focaliza principalmente as fases iniciais do ciclo de vida de projeto, tais como a definição do problema e o design (modelagem, análise, design preliminar e testes de protótipos). O *Ambiente* proposto contempla não somente a modelagem e análise de domínios, mas também processos que antecedem essas atividades, como a análise de requisitos e geração de especificações que serão tratadas de forma similar ao desenvolvimento de sistemas de engenharia em geral. Devido ao fato do ambiente integrado proposto colocar foco inicialmente nas fases iniciais (requisitos, modelagem e análise) de projeto, algumas das demais fases (testes com planejadores, escolha de técnicas de planejamento, implementações, testes, entre outras) serão discutidas brevemente neste capítulo.

O *Ambiente* visa integrar as ferramentas e linguagens mais adequadas e mais utilizadas nas fases iniciais de projeto para que os progressos já atingidos sejam utilizados. Neste

trabalho foram escolhidas, inicialmente, ferramentas e linguagens vastamente utilizadas em aplicações reais de engenharia, tais como a UML, XML, Redes de Petri e PDDL, conforme visto no Capítulo 2, para demonstrar e consolidar os conceitos presentes no ambiente proposto.

Nesse *Ambiente*, os processos de *Análise de Requisitos* e *Especificação* são realizados utilizando o *Diagrama de Casos de Uso* da UML juntamente com a Rede de Petri para verificação dos requisitos seguindo o trabalho realizado por Santos e Silva (2004). O processo de *Modelagem de Domínios de Planejamento* segue uma abordagem orientada a objetos utilizando a UML, através de seus diagramas. Devido ao fato dos domínios de planejamento possuírem características específicas, algumas regras são sugeridas durante a modelagem utilizando a UML de modo a gerar um processo disciplinado de modelagem para este tipo de problema. Esta utilização da UML para modelar domínios de planejamento é nomeada neste trabalho de "*UML para Planejamento*" (ou "*UML: Uma abordagem para Planejamento"*) e convenientemente apelidada de UML.P (VAQUERO; TONIDANDEL; SILVA, 2005). O processo de *Análise do Modelo do Domínio* será realizado utilizando as Redes de Petri para a verificação do comportamento dinâmico do domínio.

Depois de modelado e analisado o domínio, problemas de planejamento podem ser testados e novamente verificados com a utilização de técnicas de planejamento gerais (observadas as restrições), os planejadores, através da linguagem PDDL. De fato, é possível analisar qual técnica de planejamento disponível é mais recomendada para o modelo gerado para que nas fases de desenvolvimento estes sejam eventualmente reutilizados para a construção de planejadores dedicados ao domínio. Esta análise não será tratada neste trabalho, pois não está no escopo da proposta, mas alguns pontos importantes são discutidos.

Este capítulo inicia com uma discussão sobre alguns aspectos relevantes dos domínios de planejamento que devem ser levados em consideração no ambiente durante os processos de

design. Em seguida, é apresentada uma breve descrição do conceito de ciclo de vida de projeto no qual o *Ambiente* está inserido. A proposta do ambiente integrado é então apresentada baseada no cenário das áreas do Planejamento Automático e da Engenharia do Conhecimento aplicada ao Planejamento Automático apresentado no Capítulo 2, bem como os trabalhos disponíveis na literatura. Neste tópico é apresentado o conceito do *Ambiente*, bem como a integração dos processos iniciais de projetos. Os processos dentro do *Ambiente* são apresentados da seguinte maneira: a identificação dos requisitos, análise de requisitos e especificação são descritas sucintamente; a aquisição do conhecimento e modelagem de domínios e problemas utilizando a *UML para Planejamento* são apresentadas; a análise dinâmica do modelo utilizando Redes de Petri é também apresentada; uma breve descrição sobre testes com planejadores é elaborada em seguida; e por fim, é discutido o papel dos planejadores no ambiente. Este capítulo é finalizado com uma breve discussão sobre alguns trabalhos já desenvolvidos que abordam os aspectos de análise e modelagem de domínios de planejamento e uma discussão da necessidade de uma ferramenta que contemple todos os aspectos do *Ambiente* proposto.

3.1. Alguns Aspectos dos Domínios de Planejamento Automático

Conforme visto na revisão da literatura no Capítulo 2, o Planejamento Automático em IA estuda o raciocínio automático inteligente de planejamento envolvendo basicamente objetos, ações, eventos, objetivos, atividades, tempo e recursos. O Planejamento Automático é baseado principalmente em conhecimento. Como visto anteriormente, o planejamento envolve a manipulação do conhecimento de fenômenos complexos tais como as ações. Com este fato, a representação do conhecimento através de um modelo de domínio acaba sendo um dos

principais aspectos no desenvolvimento de sistemas de planejamento (MCCLUSKEY et al., 2003).

Além dos domínios de planejamento serem complicados de se modelar, seus modelos são difíceis de se projetar, escrever, corrigir, verificar, validar e manter, mesmo para os especialistas na área (WILKINS, 1999) (TATE; DRABBLE; DALTON 1996). Quanto maior e mais complexa a aplicação de planejamento, maior são os desafios encontrados na formação do modelo de domínio, fato que ocorre também no desenvolvimento de sistemas de engenharia em geral. Especificamente em planejamento, é geralmente necessário que o modelo contenha uma descrição declarativa da dinâmica do domínio, para a geração de uma base de conhecimento. De fato, um dos componentes mais importantes do modelo é a descrição das ações já que os principais resultados disponibilizados por um sistema de planejamento são as seqüências de ações necessárias para atingir um determinado objetivo, ou meta (MCCLUSKEY et al., 2003).

Certas propriedades dos domínios de planejamento podem dificultar sua modelagem e seu entendimento, principalmente os domínios reais de planejamento. Os domínios podem possuir: número infinito de ações e estados, efeitos de ações desconhecidos, estados desconhecidos, incertezas, restrições e relacionamentos desconhecidos, ambiente não observável, eventos exógenos, entre outros. Como visto anteriormente, as pesquisas nesta área utilizam simplificações e relaxamentos para tornar esses domínios tratáveis. Mas é importante ressaltar que ao tratar problemas reais, essas características deverão ser bem entendidas e representadas de alguma maneira no modelo. De fato muito trabalho precisa ser feito para que as representações e técnicas de planejamento possam tratar características como as citadas acima.

3.2. Visão Geral de Ciclo de Vida de Projeto

Um ambiente de modelagem e análise de domínios de planejamento automático reais deve estar inserido no contexto de "Ciclo de Vida de Projeto" de sistemas reais. O conceito de "Ciclo de Vida de Projeto" é definido pelo conjunto de fases de um projeto. Este conceito é muito praticado, por exemplo, na Engenharia de Software.

Todo projeto pode ser subdividido em fases ou etapas. É possível fazer um paralelo com a divisão da própria evolução da vida onde existem fases como: o "nascimento" representando o surgimento da necessidade e aceitação dos desafios e responsabilidades envolvidas; o "crescimento" representando o planejamento e a execução dos esforços necessários; o "declínio" que representa o início do alcance das metas; e por fim a "morte" representando o fim do ciclo de vida onde "todos" os objetivos foram atingidos. A utilização dessas divisões em projetos nos possibilita algumas vantagens (PMI, 2000):

- Melhor acompanhamento e controle dos recursos utilizados, do tempo investido, e do desenrolar dos acontecimentos favorecendo possíveis ajustes e as correções necessárias;
- A correta análise do que foi feito e o que não foi feito no projeto;
- Identificação do ponto exato em que o projeto se encontra;
- Pode contribuir ainda para a reutilização de conhecimento e experiências acumuladas no projeto.

Cada fase é marcada pela entrega de algum material ou apresentação de algum resultado, por exemplo, a fase inicial, onde ocorrem análises de requisitos e especificações, é geralmente finalizada com a entrega do documento de requisitos ou da especificação. A definição das fases do ciclo de vida é baseada, basicamente, na identificação dos materiais e resultados a

serem entregues, que devem ser suficientemente adequados para suportar uma contribuição adequada para o problema que o projeto se propõe a resolver. Conseqüentemente, a seqüência lógica das fases contribui para a correta definição do resultado final do projeto. Logo, o conceito de ciclo de vida de projeto é fundamental para propiciar apoio na gestão do projeto (PMI, 2000).

Na literatura existem variações quanto ao número de fases no ciclo de vida do projeto, mas basicamente podemos considerar um modelo onde existem apenas três fases: ***Definição*** (definição dos requisitos, especificação do artefato, estimativa de metas e definição da equipe); ***Design*** (conversão dos requisitos em modelo, análise de modelos, design preliminar e testes de protótipos); ***Implementação*** (produção do artefato, testes e verificação de performance). Este modelo propicia uma visão geral das operações e atividades presentes em cada fase.

Alguns pontos importantes devem ser levados em consideração acerca da tomada de decisão ao longo do projeto, como por exemplo:

- Os custos de projeto são geralmente crescentes à medida em que a fase avança;
- Os riscos são geralmente decrescentes à medida que a fase avança;
- Ignorar fases pode causar uma série de problemas em cadeia bem como o não atendimento dos objetivos;
- A participação e comunicação de todas as partes envolvidas (usuários, especialistas, *stakeholders*, etc) é fundamental, principalmente na fase de definição;
- Caso seja necessário alterar o escopo do projeto, o momento ideal de fazê-lo é na fase inicial. Com o passar do tempo o custo de mudança torna-se cada vez maior;
- A capacidade de adaptação de um projeto é maior no início, ficando menor com o passar das fases.

Analisando os pontos citados acima é possível perceber que grande atenção deve ser dada às fase inicial do ciclo de vida do projeto, pois problemas na definição do problema e no levantamento dos requisitos podem causar impactos negativos nas fases de design e implementação onde os custos e uso de recursos envolvidos são maiores. Este fato não se mostra diferente para projetos de sistemas de planejamento automático, principalmente os complexos e reais. Baseado neste fato, o presente trabalho coloca foco exclusivamente nas fases iniciais de definição do problema, aquisição do conhecimento e modelagem dos domínios, pois estas são cruciais para o sucesso da aplicação de planejamento automático.

Na próxima seção é apresentado o *Ambiente* proposto de modelagem e análise de domínios de planejamento automático que visa contemplar os aspectos relevantes tanto dos domínios de planejamentos quanto do ciclo de vida de projeto.

3.3. O *Ambiente* Integrado Proposto

A proposta de um ambiente integrado para a modelagem e análise de domínios de planejamento apresentada neste tópico foi desenvolvida motivada não só pelos desafios encontrados nos domínios de planejamento e pela falta da prática do conceito de ciclo de vida de projeto nestes domínios, mas também por alguns trabalhos e cenários que ressaltavam a necessidade de pesquisas nesta área da IA. Alguns pontos que levaram à elaboração desta proposta são:

- O objetivo comum da comunidade de pesquisa de alcançar problemas reais de planejamento;
- A necessidade de unir a teoria à prática de planejamento (MCDERMOTT; HENDLER, 1995) (GIL et al., 1995) (GHALLAB; NAU; TRAVESSO, 2004);

- Alguns trabalhos mostram que simples variações na representação dos domínios de planejamento podem mudar a performance de um planejador. Este fato fortalece a importância de um processo de modelagem disciplinado e bem elaborado.
- Um dos principais trabalhos nesta área, realizado por McCluskey et al. (2003), com o título de "*Knowledge Engineering for Planning ROADMAP*", traz diversas discussões construtivas sobre modelagem de domínios, linguagem de representação, ferramentas e ambientes de suporte à engenharia do conhecimento, ontologias para planejamento, entre outros aspectos. Neste trabalho são apresentadas diversas ações que a comunidade de Planejamento Automático deve abordar durante o desenvolvimento de ferramentas e métodos de modelagem. Esta proposta contempla diversos aspectos provenientes deste trabalho.
- Falta de métodos de especificação e modelagem de domínios de planejamento automático (MCCLUSKEY et al., 2003);
- Falta de ferramenta de suporte ao processo de especificação, modelagem e análise de domínios de planejamento;
- A linguagem padrão PDDL não é recomendada para ser usada como uma linguagem de modelagem inicial e sim como uma linguagem a ser interpretada pelo planejador. Para problemas clássicos o uso desta linguagem se faz viável;
- Dificuldade na visualização e entendimento das linguagens utilizadas em Planejamento Automático;
- Problemas reais possuem grandes desafios de modelagem;
- Demanda de um ambiente que possa integrar todos esses conceitos da Engenharia de Requisitos e dos *viewpoints* dos participantes (SANTOS; SILVA, 2004) (SANTOS, 2002).

Levando em consideração os pontos listados acima, trabalhos que agreguem conhecimento e experiência nesta área da IA são bem aceitos tanto pela comunidade de pesquisa quanto pelas empresas. Neste panorama é que este trabalho descreve, focalizando as fases de modelagem e análise do modelo do domínio, um ambiente de design que integra diversas ferramentas e linguagens (usuais e vastamente utilizadas) de modo a unir o potencial de cada uma delas.

O principal objetivo do *Ambiente* é integrar os pontos de vista (*viewpoints*) dos participantes e os diversos contextos do design (análise de requisitos, especificação, modelagem, verificação, validação, testes, entre outros) para o desenvolvimento de um sistema de planejamento automático onde cada contexto esteja interligado pela estrutura do *Ambiente*. A interligação dos contextos faz com que o projetista possa efetuar diversos processos sobre um mesmo domínio onde a passagem de um contexto a outro é realizada de forma transparente. Por exemplo, durante a construção de um modelo o projetista pode realizar análises dinâmicas ou estáticas do modelo do domínio utilizando uma outra linguagem ou ferramenta, fazendo com que a análise traga um refinamento do próprio modelo. Um visão geral do ambiente integrado é representada na Figura 23.

Figura 23 – Visão geral do ambiente integrado.

O *Ambiente* proposto provê mecanismos de mudança de contexto através de uma representação central do modelo sendo realizado, chamada neste trabalho de *Representação Core*. Com uma *Representação Core* é possível representar o modelo, ou parte dele, em outras linguagens para que outras ferramentas sejam utilizadas, cada qual com seu objetivo específico: análise, modelagem, validação, teste, entre outros. Esta representação será melhor apresentada e contextualizada nos próximos capítulos.

No presente trabalho o *Ambiente* integra principalmente os processos de *Análise de Requisitos*, *Especificação*, *Modelagem*, *Análisee do Modelo do Domínio* e eventuais testes com planejadores. Este trabalho propõe processos de modelagem e análise do modelo do domínio direcionados ao design de domínios de planejamento automático, utilizando principalmente a linguagem UML (UML.P) e as Redes de Petri, respectivamente. Os tópicos

a seguir apresentam os principais processos, dentro do ciclo de vida de projeto, presentes no *Ambiente*.

Durante a explicação dos processos será utilizado como exemplo o domínio clássico de planejamento chamado *Logística*. Neste domínio simples (e fechado) existem, basicamente, *Caminhões* e *Aviões* que transportam *Pacotes* entre *Lugares*. *Caminhões* se locomovem entre *Locais* da mesma *Cidade* e *Aviões* somente entre *Aeroportos*. A Figura 24 ilustra este domínio clássico.

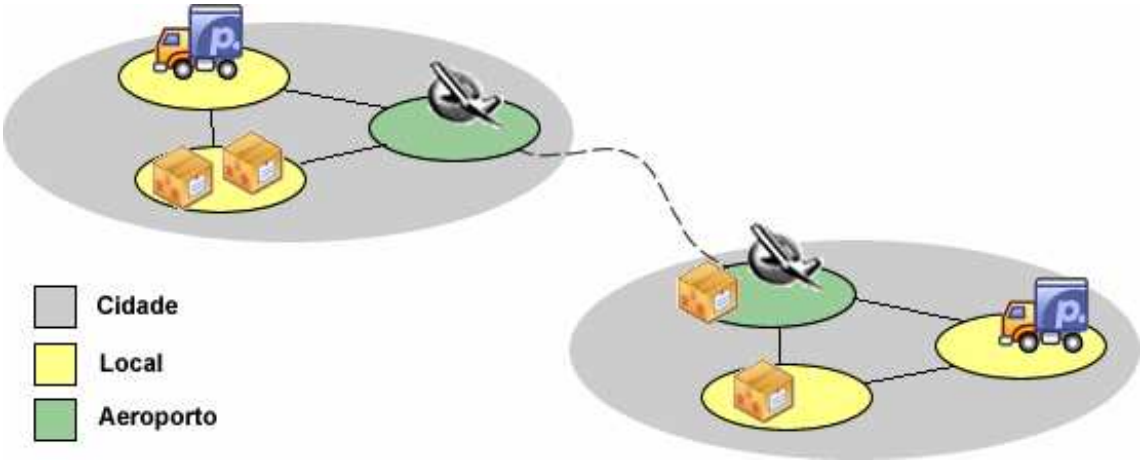


Figura 24 - O domínio clássico de planejamento *Logística*.

3.3.1. Requisitos e Especificação

Neste trabalho é sugerida a realização, no contexto do *Ambiente*, das fases de requisitos, análise de requisitos e especificação (presentes em um ciclo de vida de projeto) da mesma forma dos procedimentos já descritos anteriormente, bem como aqueles utilizados no desenvolvimento de sistemas em geral. Mesmo assim alguns pontos são reforçados no ponto de vista de planejamento.

Nas fases iniciais, além dos documentos, manuais e modelos já existentes, o papel dos pontos de vista (*viewpoints*) sobre o domínio a ser analisado é de fundamental importância. Pontos de vista provenientes dos *stakeholders*, projetista, especialista em sistemas baseados em conhecimento, usuários, especialistas do domínio, especialistas em planejamento automático, desenvolvedores, agregam informações aos requisitos e ao conhecimento do domínio de modo a definir as principais características existentes (SANTOS; SILVA, 2004) (MCCLUSKEY et al., 2003).

A definição dos requisitos, a sua análise e a especificação são processos cruciais para definir o papel do sistema de planejamento automático e o tipo de planejamento adequado.

A análise dos requisitos pode ser vista como um processo de verificação das informações adquiridas perante os diversos pontos de vista. Alguns trabalhos apresentam processos sistemáticos de análise de requisitos. Um dos principais trabalhos nesta área é o trabalho de Santos e Silva (2004) que utiliza uma abordagem direcionada a *Casos de Uso* (*Use Case Driven*) onde o *Diagrama de Casos de Uso* da UML é utilizado para definir o escopo do domínio, onde os *Casos de Uso* são definidos de forma estruturada com o objetivo de se gerar uma Rede de Petri para que esta possa ser analisada.

O *Diagrama de Caso de Uso* representa o domínio em um nível alto de abstração onde o escopo do domínio pode ser definido inicialmente. Este diagrama facilita a unificação de todos os pontos de vista dos participantes que por sua vez podem iniciar discussões sobre os requisitos. No contexto de planejamento, este diagrama pode evidenciar as principais entidades presentes no domínio, suas funcionalidades, os *Agentes* (*Atores*, entidades que agem sobre o domínio) e algumas restrições. O diagrama ajuda o projetista e os participantes a decidir e especificar o que pertence e o que não pertence ao domínio, os requisitos de cada *Caso de Uso* do domínio, bem como os principais *Agentes* do domínio de planejamento.

A construção do *Diagrama de Casos de Uso* acaba sendo uma tarefa relativamente simples mesmo para problemas reais de planejamento. Devido ao fato do diagrama ser definido de forma informal, é possível obter uma boa visão do domínio como um todo, de forma abstrata. Um exemplo simples é mostrado na Figura 25. Esta figura representa uma visão abstrata do domínio clássico de planejamento da Figura 24, o *Logística*. Mesmo em um nível de abstração alto, o domínio da *Logística* é claramente visto e entendido usando uma descrição geral do domínio através do *Diagrama de Casos de Uso*.

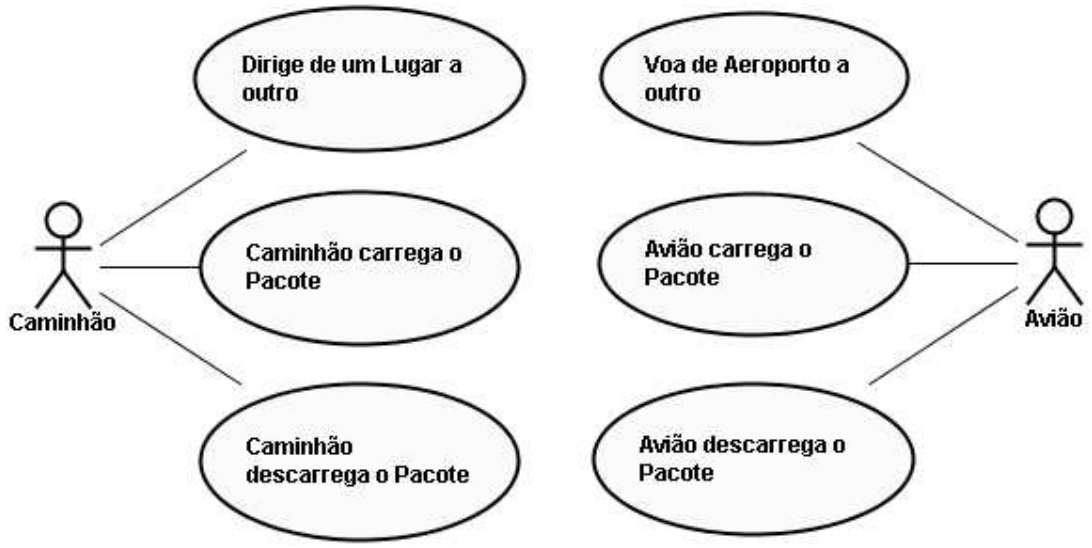


Figura 25 - O domínio clássico de planejamento *Logística* representado no *Diagrama de Casos de Uso.*

Na UML, os *Casos de Uso* são geralmente descritos em linguagem natural em níveis de abstração desejados. Conforme visto no Capítulo 2, cada *Caso de Uso* contém uma breve descrição, os requisitos funcionais, suas restrições (pré e pós-condições e invariantes), bem como uma seqüência de ações e acontecimentos, ou fluxo de eventos, realizados por um ou mais *Agentes*. Conforme descrito anteriormente, a representação e a descrição e do fluxo de eventos (eventos exógenos ou não) de um *Caso de Uso*, para a análise de requisitos proposta em (SANTOS; SILVA, 2004), pode ser realizada de forma estruturada. Devido ao fato da

possibilidade das especificações em linguagem natural gerar ambigüidade e redundância a proposta de uma definição estruturada de casos de uso pode minimizar estes problemas. Tal estruturação é definida em (SANTOS; SILVA, 2004) com objetivo geral de verificação de requisitos. A estruturação permite o projetista simular o *Caso de Uso* através de uma Rede de Petri onde o mesmo pode visualmente verificar o que foi especificado e refinar os requisitos se necessário. Esta abordagem estruturada de definição do fluxo de eventos é utilizada neste trabalho como parte da definição de um *Caso de Uso*, pois pode contribuir muito para o levantamento inicial das possíveis ações existentes no domínio, bem como as pré e pós-condições das mesmas. Outra abordagem na definição do fluxo de eventos de um *Caso de Uso* seria através da utilização do Diagrama de Seqüência, mas esta abordagem não será apresentada neste trabalho.

Seguindo o exemplo *Diagrama de Caso de Uso* da Figura 25, o *Caso de Uso* "*Dirige de um Lugar a outro*" poderia ser descrito como mostra a seguir.

Tabela 2 - Descrição do *Caso de Uso* "*Dirige de um Lugar a outro*" da Figura 25.

Caso de Uso: ***Dirige de um Lugar a outro***
Agente: Caminhão **Descrição:** Este Caso de Uso descreve a atividade do caminhão de se locomover de um lugar a outro respeitando as restrições. ***Restrições:*** **Pré-condição** - O caminhão deve estar em um lugar de partida pertencente a uma cidade; - O lugar de destino deve pertencer à mesma cidade do lugar de partida. **Pós-condições** - O caminhão passa a estar no lugar de destino. **Invariantes** Durante a locomoção do caminhão nenhum pacote que se encontra dentro do mesmo é retirado.

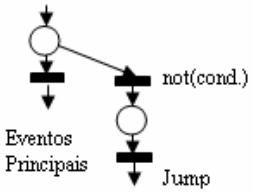
A descrição do fluxo de eventos do *Caso de Uso* acima descrito utilizando a proposta de Santos e Silva (2004) seria basicamente a seguinte: “Definir o lugar de destino; Verifica se o lugar de destino e o lugar de partida pertencem à mesma cidade; se pertencem a mesma cidade, o Caminhão se move do lugar de partida até o lugar de destino; caso não pertençam a mesma cidade (fluxo alternativo), procura por um destino válido”. A estruturação desse fluxo de eventos é apresentada na Tabela 3.

Tabela 3 - Descrição do fluxo de eventos do *Caso de Uso* “*Dirige o Caminhão*” da Figura 25.

●	Início
⬭	Definir o lugar de destino;
◇	Verifica se o lugar de destino e o lugar de partida pertencem a mesma cidade;
↳	Procura por um destino válido
⬭	Mova do lugar de partida até o lugar de destino;
◉	Fim

Um esquema simples da tradução do fluxo de eventos estruturado para uma Rede de Petri é apresentado na Tabela 4. Todo o processo de tradução do caso de uso estruturado para uma Rede de Petri está definido em (SANTOS; SILVA, 2004). A Figura 26 demonstra a RdP resultante.

Tabela 4 - Esquema de tradução de fluxo de eventos estruturado de um *Caso de Uso* para Redes de Petri (SANTOS; SILVA, 2004).

Elementos da Estruturação dos Casos de Uso	**Representação em Redes de Petri**
● Início	
⬭ Básico	
◇ Condicional ↳ Jump (Pulo)	
◉ Fim	

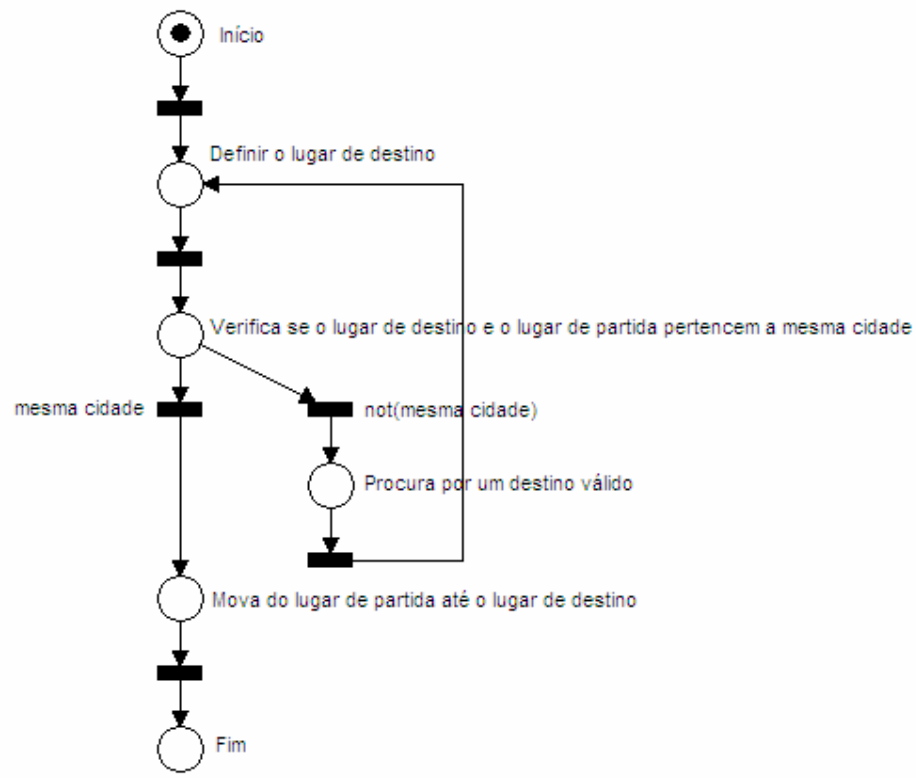


Figura 26 - Rede de Petri resultante do fluxo de eventos do *Caso de Uso* "*Dirige de um Lugar a outro*".

Com o entendimento do domínio e análise de requisitos é possível realizar a *especificação do domínio* representada por um documento que descreve as característica e funcionalidades do domínio. Geralmente o *Diagrama de Casos de Uso* da UML, juntamente com as descrições dos *Agentes* e dos *Casos de Uso*, fazem parte do corpo da especificação do domínio.

3.3.2. Modelagem

Um dos principais aspectos que dificultam a modelagem de domínios de planejamento é a falta de métodos e princípios de modelagem voltados a esses tipos de domínios. Esses métodos disciplinam todo o processo de modelagem. A definição de métodos de modelagem

é, geralmente, proveniente de experiências vividas por profissionais que acumularam vivência no desenvolvimento de sistemas. No caso do Planejamento Automático, os métodos e procedimentos de modelagem estão ainda surgindo, como é o caso da proposta apresentada neste tópico.

A abordagem proposta aqui leva em consideração os métodos e boas práticas de modelagem provenientes do desenvolvimento de software, onde a UML é vastamente utilizada, e dos métodos de modelagem da abordagem conhecida por "*Naked Object*" (PAWSON; MATTHEWS, 1999), onde o projetista deve procurar modelar apenas as características principais dos domínios (como elas realmente são no mundo real e também como elas são de fato afetados pelo ambiente que envolve o domínio) realizando processos exaustivos de refinamento. Como a UML é muito utilizada no desenvolvimento de software, os projetistas geralmente desenvolvem seus modelos com o objetivo de torná-los o mais próximos da programação que será realizada na fase de implementação, ou seja, muitos diagramas da UML utilizam termos e estruturas de forma a facilitar o trabalho do programador durante a fase de implementação. Na abordagem de modelagem apresentada neste trabalho, que utiliza a UML, o processo de modelagem visa tornar o modelo domínio o mais próximo do domínio real (com um mapeamento claro entre o modelo e o domínio real) e eventualmente utilizar artifícios de programação para representar algumas características. Durante a explicação do processo de modelagem serão apresentados alguns pontos que evidenciam esta diferença de abordagem na modelagem.

Nos tópicos a seguir é apresentada, primeiramente, uma definição de domínios e problemas de planejamento do ponto de vista do autor e em seguida a proposta de modelagem de domínios e problemas de planejamento utilizando a *UML para Planejamento.*

3.3.2.1. Definição de Domínio e Problema de Planejamento

Neste trabalho um domínio $\mathcal{D}$ é visto como uma 6–tupla {A,, *Ag, Rs, S, Ru, Ac*} onde:

- A = (*T*, *At*; τ) é um alfabeto composto por um conjunto de tipos abstratos *T* e um conjunto de atributos *At*, estruturados pelo mapeamento $\tau : At \rightarrow T$;
- *Ag* é um conjunto finito de instâncias de agentes (atores) do domínio, cuja classe $\mathcal{C}_{Ag} \subseteq T$. Agentes da classe $\mathcal{C}_{Ag}$ são capazes de modificar os seus atributos ou os atributos de outras classes;
- *Rs* é o conjunto finito de instâncias de recursos do domínio, cuja classe $\mathcal{C}_{Rs} \subseteq T$. Recursos da classe $\mathcal{C}_{Rs}$ não são capazes de modificar os seus atributos e nem os atributos de outras classes;

Cada objeto, seja do tipo agente ou recurso possuem, além dos atributos (contidos em *At*) instanciados $\tau(t)$ que definem a sua identidade ($t \in T$), outros (também contidos em *At*) que são chamados de propriedades deste objeto *t*. Uma mudança de propriedade não afeta a identidade do objeto.

- *S* é um conjunto de *snapshots* do domínio, composto por n-uplas, cada uma delas contendo todas as instâncias de agentes e recursos do domínio, $\imath \subseteq (Ag \times Rs)$, que tem suas propriedades igualmente instanciadas. A este conjunto denomina-se conjunto de estados do sistema;
- *Ru* é um modelo lógico do domínio que descreve, em lógica de primeira ordem, as relações entre os elementos do domínio, restrições e invariantes;

- *Ac* é um conjunto finito de ações que envolvem pelo menos um agente ou classe de agente e recursos, e resultam na mudança de estado.

Um problema de planejamento $\mathscr{P}$ em $\mathcal{D}$ é uma 3-tupla $\{S_0, S_G, \overline{Ac}\}$ onde:

- $S_0 \in$ S é um estado específico chamado de estado inicial;
- $S_G \in$ S é um estado específico chamado estado objetivo ou estado final;
- $\overline{Ac} \subseteq Ac$ é um conjunto de ações pertinentes ao problema (que podem ser utilizados na execução do plano).

Eventualmente podem ser definidas *métricas*, *restrições* e *invariantes* do problema. Estas *métricas* representam funções que indicam a proximidade do estado corrente para o estado objetivo. Eventualmente podem ser usadas para otimizar o plano. *Restrições* são propriedades extemporâneas do domínio. *Invariantes* são propriedades que identificam o domínio e devem ser válidas em qualquer estado.

Algumas observações em relação à definição acima são:

- Da definição acima alguns elementos de $\mathcal{D}$ identificam certos grupos de características. Os A, e *Ru* elementos compõe as características da *estrutura estática* de um domínio. Já os elementos *S* e *Ac* caracterizam os *aspectos dinâmicos* do mesmo.
- Em muitos domínios, principalmente aqueles de planejamento automático, enumerar os elementos de *S* se torna uma tarefa difícil devido ao enorme número de estados.
- Conforme visto na definição acima, no presente trabalho o tempo não é considerado.

3.3.2.2. Modelagem utilizando a UML para Planejamento

Como visto a UML – *Unified Modeling Language* – é uma linguagem semi-formal de modelagem muito utilizada para modelar sistemas em geral, principalmente aqueles cuja abordagem é orientada a objetos (OMG, 2001). Devido ao fato da UML ser de propósito geral, algumas características intrínsecas aos domínios de planejamento (e conseqüentemente ao Planejamento Automático) devem ser levadas em consideração no uso desta linguagem. Para tal, a UML.P (*UML para Planejamento*) é proposta neste trabalho como uma forma de utilização da linguagem UML, que leva em consideração abordagens disponíveis na literatura (PAWSON; MATTHEWS, 1999) (D'SOUZA, WILLS, 1999), para a modelagem de domínios de planejamento automático. Assim, os conceitos e características destes domínios são mapeados e caracterizados nos diagramas UML.

Utilizando a UML, alguns diagramas são muito apropriados no contexto de domínios de planejamento, tais como o *Diagrama de Casos de Uso* (como visto na análise de requisitos), *Diagrama de Classes, Diagrama de Estados* e o *Diagrama de Objetos*. É evidente que os outros diagramas também são apropriados, mas nesta proposta apenas estes diagramas são abordados.

As descrições do uso de cada um dos diagramas para a modelagem de domínio de planejamento a seguir mostram como os projetistas e *stakeholders* podem melhor modelar e entender os domínios. Primeiramente é mostrado como o projetista pode modelar o domínio e posteriormente o problema de planejamento. O processo de modelagem aqui proposto é dividido em duas etapas. A primeira considera a modelagem do domínio de planejamento $\mathcal{D}$, e a segunda considera a modelagem do problema $\mathcal{P}$.

3.3.2.2.1. Modelagem do Domínio

Para modelar o domínio de planejamento é necessária, em linhas gerais, a modelagem da estrutura estática do modelo, as características dinâmicas, os agentes e os recursos conforme visto na definição do domínio no tópico anterior. Para modelagem da estrutura estática do modelo (os conjuntos *T*, *At*, τ e *Ru*) é utilizado o *Diagrama de Classes*. Já a dinâmica é modelada utilizando o *Diagrama de Estados* onde é possível representar as interações e o comportamento das ações (o conjunto *Ac*) sobre os objetos. Por fim, é utilizado o *Diagrama de Objetos* para a representação dos agentes (*Ag*) e recursos (*Rs*) do modelo do domínio.

3.3.2.2.1.1. Estrutura Estática

Entre todos os diagramas da UML, o *Diagrama de Classes* é o mais utilizado e o mais conhecido para representar e modelar a estrutura estática de domínios com uma abordagem orientada a objetos. Este diagrama é utilizado para representar o modelo estático conceitual (abstrato) do domínio de planejamento mostrando as entidades (chamadas aqui de "classes" ou "tipos abstratos") existentes, os seus relacionamentos, atributos, características, os nomes das ações (que em programação são chamadas de métodos) e restrições.

De fato, classes e objetos são os conceitos mais importantes na modelagem orientada a objetos. As classes representam as entidades que fazem sentido no contexto do domínio real. Estas são extraídas facilmente da análise de requisitos e da especificação do domínio (PAWSON; MATTHEWS, 1999). Conforme Pawson e Matthews (1999), através de um levantamento das principais classes existentes no domínio e posterior refinamento, é possível obter as classes essenciais para o modelo do domínio, aquelas que realmente fazem sentido no domínio real.

Nesta proposta, as classes que mudam o arranjo dos objetos em um domínio através de suas ações são chamadas aqui de *Classes Agentes* ($T_{Ag} \subseteq T$). Agentes são geralmente os atores

encontrados no *Diagrama de Casos de Uso*. É através do comportamento colaborativo dos Agentes que um problema de planejamento pode ser resolvido, pois faz com que os objetos recursos e o próprio ambiente sejam afetados de forma a levar todo o ambiente de um estado inicial a um estado objetivo. As classes que representam os objetos recursos são chamadas aqui de *Classes Recursos* ($T_{Rs} \subseteq T$). Na UML.P, *Classes Agentes* e *Classes Recursos* são identificadas através da utilização de *stereotypes*. Uma classe é identificada como sendo uma *Classe Agente* quando esta possui o *stereotype* `<<agent>>`. Já uma classe é identificada como sendo uma *Classe Recurso* quando esta possui o *stereotype* `<<entity>>` ou quando não possui nenhuma identificação de *stereotype* (*default*). Seguindo o exemplo do domínio *Logística* as principais classes, tanto agentes quanto recursos ($T = T_{Ag} \cup T_{Rs}$), são mostradas na Figura 27 onde T_{Ag} = {*Caminhão, Avião*} e T_{Rs} = {*Cidade, Local, Aeroporto, Pacote*}.

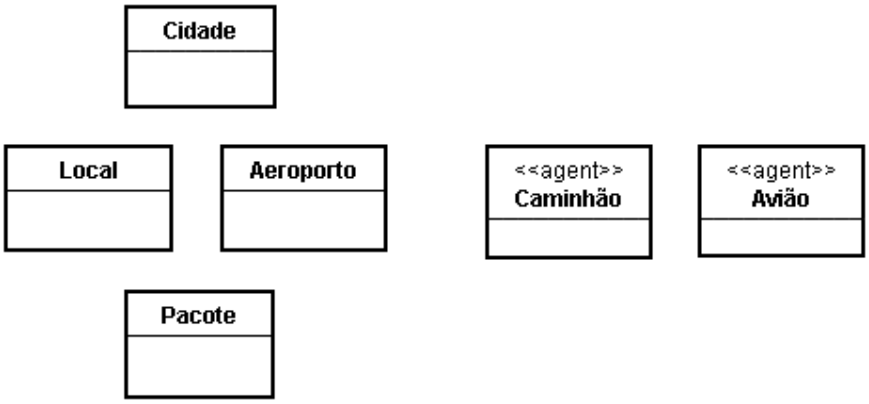


Figura 27 – As principais classes (agentes e recursos) do domínio clássico de planejamento *Logística*.

Associações entre as classes, atributos (*At*) e ações das classes fornecem uma excelente visão e noção da semântica do modelo. Qualquer relacionamento entre duas ou mais classes é considerado uma associação. No vocabulário de planejamento em IA, atributos e associações são geralmente chamados de predicados (como é o caso da PDDL). De certa forma associação também podem ser vistas como atributos onde seus tipos de valores são outras classes.

Associações tais como as associações simples, agregação e composição geralmente fazem com que o modelo do domínio se torne semanticamente mais próximo do domínio real. As associações simples apenas conectam classe identificando algum significado apropriado. Agregações e Composições já expressam significados como, por exemplo, “possui” (“pertence”) e “constituído de”, respectivamente. Essas associações possuem geralmente nomes, com uma direção semântica, *rolenames* e multiplicidades em ambos os lados. Como visto anteriormente, os *rolenames* representam as navegações entre as classes de uma associação, e as multiplicidades representam restrições (regras), indicando a maneira como os objetos se relacionam, que permitem o raciocínio sobre situações e estados válidos do modelo. No exemplo do domínio *Logística*, é possível citar regras de relação (elementos de *Ru*) através da multiplicidade como: um C*aminhão* pode *estar* em um e somente um *lugar* (*1*) em um determinada estado, já em um *lugar* podem *estar* nenhum, um ou muitos *veículos* (*0..**). A Figura 28 mostra o *Diagrama de Classes* do domínio *Logística* com as classes essenciais e as associações representando as principais características da estrutura estática do domínio real.

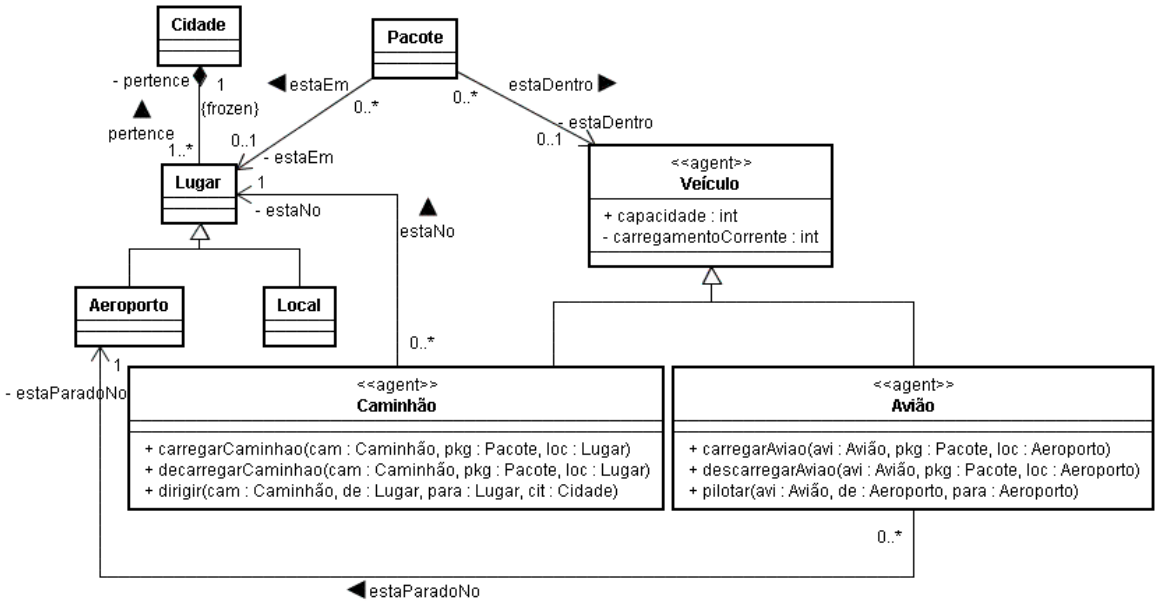


Figura 28 – *Diagrama de Classes* do domínio *Logística*.

O exemplo do diagrama apresentado na Figura 28 mostra as associações que fazem sentido no domínio real como, por exemplo, a associação "*estaEm*" entre as classes *Pacote* e *Lugar* traz um significado visual para usuários não especialistas em planejamento ou em modelagem.

Algumas informações são muito importantes no *Diagrama de Classe* e facilmente mapeadas no contexto de planejamento (UML.P). Por exemplo, os conceitos de *relações flexíveis* (associação) e *relações rígidas* (conceitos definidos por Ghallab et a. (2004)) são facilmente visualizadas no diagrama. Uma *relação flexível* representa as associações que mudam durante a evolução do ambiente (durante a vida dos objetos). Já a *relação rígida* representa as associações que não variam de estado para estado. Um bom exemplo de *relação flexível* é a associação "*estaEm*" entre as classes *Pacote* e *Lugar* já que um pacote pode ser levado de um lugar a outro na evolução dos estados do domínio. Um exemplo de *relação rígida* é a associação "*pertence*", marcada pela restrição *frozen* no diagrama da Figura 28, entre as classes *Lugar* e *Cidade*, pois esta representa a topologia do domínio que é possível se dizer que é imutável. Normalmente as associações de agregação e composição também são *relações rígidas* nos domínios de planejamento.

Durante a representação das classes no diagrama é possível notar que algumas delas possuem características e comportamentos comuns. Para melhorar o modelo, as classes podem ser organizadas utilizando o conceito de herança proveniente da orientação a objetos. O uso do conceito de herança faz com que as classes compartilhem estruturas em comum, através de uma hierarquia de classe. Heranças podem ser adicionadas através da generalização de classes, por exemplo, a classe *Veículo* é vista como uma superclasse de *Caminhão* e *Avião* onde estas duas classes herdam todas as características da classe *Veículo* incluído as associações. No caso, a classe *Pacote* pode se associar tanto com a classe *Caminhão* quanto com a classe *Avião* através da "*estaDentro*" devido às características herdadas por essas duas

classes da classe *Veículo*. As classes também herdam os atributos e métodos, como é o caso do atributo "*capacidade*", do tipo *Inteiro* (*0, 1, 2, 3,* ...), e do atributo "*carregamentoCorrente*" (representando um contador), também do tipo *Inteiro*, da classe *Veículo* representando o número máximo de pacotes que poderiam estar *dentro* do veículo, com isso, tanto o *Caminhão* quanto o *Avião* possuiriam este atributo. Como as associações podem ser vistas também como atributos (por exemplo, a associação *estaEm* é um atributo da classe *Pacote* e é do tipo *Lugar*), é possível perceber que o conjunto *At* é representado pelos elementos {*estaEm, estaDentro, pertence, estaNo, estaParadoNo, capacidade, carregamentoCorrente*}.

É importante notar que a estrutura estática do modelo, construída no *Diagrama de Classes*, não representa um domínio de planejamento em particular, pelo contrário, esta representa um enorme conjunto de domínios devido ao fato do diagrama representar parte de uma abstração do domínio.

Neste diagrama as ações (inicialmente representadas apenas pelos nomes e parâmetros) são atribuídas às respectivas *Classes Agentes*. Da análise de requisitos é possível inferir e extrair as ações essenciais presentes no domínio. Seguindo o exemplo do domínio *Logística*, como os agentes são representados pelas classes *Caminhão* e *Avião*, nelas são declaradas as ações que cada uma realizará no modelo do domínio conforme representado na Figura 28. Por exemplo, a classe *Avião* pode executar a ação "*carregaAviao"* que representa o fato do avião poder realizar o carregamento de um pacote para dentro de seus compartimentos. É importante definir também inicialmente os parâmetros desta ação, ou seja, quais as classes estarão se envolvendo nesta ação. No caso da ação "*carregaAviao"* as classes que se envolvem são: a própria classe *Avião;* a classe *Pacote*, já que esta será levada para dentro do veiculo; e a classe *Lugar*, pois o avião estará carregando o pacote de algum lugar

determinado. Portanto, a ação deve ser declarada da forma *carregaAviao(avi: Avião, pkg: Pacote, loc: Lugar)* como mostra a Figura 28.

Conforme pode se perceber, na UML.P não são utilizados os famosos métodos *getters* na *setters* nas classes (costume proveniente das práticas de modelagem na Engenharia de Software, como, por exemplo, *getCapacidade()*, *setCapacidade()*). Isto é devido ao fato desta proposta visar a construção do modelo do domínio o mais próximo do domínio real, fazendo com que muitos participantes possam intervir no modelo sendo gerado.

Algumas restrições e invariantes (elementos de *Ru*) podem ser adicionadas ao modelo através da OCL (OMG, 2003). Utilizando esta linguagem de restrições é possível representar regras que devem ser respeitadas durante a evolução dos objetos no sistema. Por exemplo, a restrição "a *capacidade* de um *veículo* nunca pode ser excedida" pode ser representada da seguinte maneira:

```
context Veículo inv:
   self.capacidade >= self.pacote->size()
```

Uma outra restrição pode ser adicionada à classe *Pacote*. Um objeto desta classe pode estar associada a um objeto do tipo *Veículo* através da associação "*estaDentro*" ou pode estar associada a um objeto do tipo *Lugar* através da associação "*estarEm*" mas nunca ambas no mesmo estado. Em OCL esta restrição poderia ser definida da seguinte maneira:

```
context Pacote inv:
   self.estaDentro->notEmpty() xor self.estaEm->notEmpty()
```

Estas restrições podem ser discutidas com os *stakeholders* e adicionados neste diagrama, já que representam propriedades, invariantes e regras do modelo do domínio de planejamento. É possível utilizar todo o potencial da OCL para tornar o modelo enriquecido de informação.

Todos os aspectos presentes no *Diagrama de Classe* (classes, agentes, atributos, associações, multiplicidades, ações, parâmetros e restrições) devem ser exaustivamente

discutidos e analisados para que se tenha um modelo “inicial” bem formado. Diz-se modelo “inicial” pois o processo de modelagem como um todo é um processo interativo, não inibindo possíveis modificações na estrutura estática durante as fases posteriores.

Para a modelagem da estrutura estática o uso da UML.P, especificamente no uso do *Diagrama de Classes*, é muito similar ao uso convencional da UML na Engenharia de Software, mas com algumas restrições e boas práticas. Essas restrições e boas práticas podem ser resumidas da seguinte maneira:

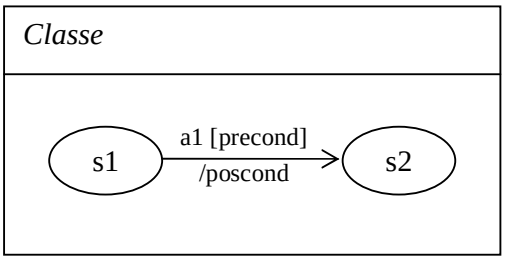
- Representação de classes essenciais com mapeamentos claros entre os modelos e os domínios reais;
- Uso dos *stereotypes* para separação dos agentes e recursos (`<<agent>>` e `<<entity>>`);
- Uso recomendado dos *rolenames* para navegações entre classes e uso das multiplicidades para a representação das regras de associação;
- Apenas as *Classes Agentes* possuem ações;
- A ações do modelo, bem como seus nomes, são facilmente identificadas no domínio real;
- Os parâmetros das ações são representados por todas as classes que participam das mesmas;
- Uso recomendado da OCL para declaração de restrições e invariantes da estrutura estática do modelo.

3.3.2.2.1.2. Características Dinâmicas

Para representar as características dinâmicas das ações que foram nomeadas no *Diagrama de Classes* é necessário um outro diagrama da UML que represente o dinamismo dos

modelos. Para tal, este trabalho utiliza o *Diagrama de Estados* e a OCL para modelar as pré e pós-condições das ações. De fato, outros diagramas da UML também são utilizados para representar os aspectos dinâmicos como, por exemplo, o *Diagrama de Seqüência* e o *Diagrama de Colaboração*, mas nesta proposta, inicialmente, apenas o *Diagrama de Estados* é utilizado, porém os demais diagramas não são descartados.

A utilização do *Diagrama de Estados* (*StateChart Diagram*) presente nesta proposta foi baseada no trabalho de D'Douza e Wills (1999) onde os autores apresentam uma abordagem de utilização da UML chamada de *Catalysis* (*Catalysis Approach*). Para D'Souza e Wills o *Diagrama de Estados* é uma ferramenta muito útil na modelagem. Para eles os *estados* e *transições* que aparecem neste diagrama são diretamente relacionados aos atributos e as ações representadas no *Diagrama de Classes*. Cada *estado* do diagrama é definido pelos valores dos atributos e associações relevantes para o *estado* em questão. O mesmo ocorre para as pré e pós-condições das ações (*transições* no diagrama). As definições dos *estados*, da pré e pós-condição das ações são realizadas através da OCL (OMG, 2003). A Figura 29 demonstra um exemplo da definição de *estados* no *Diagrama de Estados* de uma classe que possui um atributo (*atr1*) e duas associações (*assoc1* e *assoc2*) e esta pode executar uma ação *a1*.



Estado	atr1	assoc1	assoc2	Definição completa
s1	> 30	null	<> null	atr1>30 and assoc1=null and assoc2<>null
s2	= 0	y		atr1=0 and assoc1=y

Figura 29 – *Estados* no *Diagrama de Estados* definidos através dos valores dos atributos e associações com a OCL – Abordagem *Catalysis* (D'SOUZA; WILLS, 1999).

Seguindo ainda o exemplo da Figura 29, se a pré-condição (*precond*) da ação *a1* fosse definida como *x > 100* e a pós-condição (*poscond*) como *x = 0*, a representação da ação *a1* em OCL (conforme visto na Capítulo 2) seria:

```
context Classe::a1
     pre:
          -- expressões de s1
          atr1 > 30 and assoc1 = null and assoc2 <> null and
          -- expressões da precond
          x > 100
     post:
          -- expressões de s2
          atr1 = 0 and assoc1 = y and
          -- expressões da poscond
          x = 0
```

A abordagem utilizada no presente trabalho é a mesma daquela apresentada por D'Souza e Wills (1999) em relação a construção do *Diagrama de Estados*, ou seja, na UML.P os *estados* são também vistos como valores de atributos e associações relevantes, e as ações como senda aquelas presentes no *Diagrama de Classes.* No diagrama as expressões, tanto dos estados quando das pres e pós-condições, são representadas em OCL.

Como citado na revisão da literatura, em geral, toda classe do *Diagrama de Classes* que possui características dinâmicas possui seu próprio *Diagrama de Estados*, especialmente os *Agentes.* Para o exemplo do domínio *Logística* é possível dizer que apenas as classes *Pacote*, *Avião* e *Caminhão* possuirão *Diagramas de Estados* próprios, já as demais classes não possuem características dinâmicas significativas ao modelo, pois estas formam apenas a topologia do domínio.

Neste trabalho, cada *Diagrama de Estados* descreve as mudanças causadas pelas ações sobre uma única classe fazendo com que a construção do diagrama seja centrada na classe em questão. O diagrama representa os possíveis estados de um objeto da classe, assim como as ações que o afetam (essas ações podem ser realizadas por outras classes - *Agentes*). Uma ação

pode afetar diversas classes fazendo com que a mesma apareça em diversos *Diagramas de Estados*. De fato, a modelagem da ação (suas pré e pós-condições) é o resultado da união das representações dos *Diagramas de Estados* onde esta aparece. Em cada diagrama onde uma mesma ação aparece esta possuirá pré e pós-condições referentes ao contexto do diagrama. Este conceito será melhor visualizado nos exemplos descritos nos próximos parágrafos.

De um modelo geral, para a construção dos *Diagramas de Estados* do modelo do domínio de planejamento em UML.P algumas regras são necessárias para que a modelagem das ações seja disciplinada e estruturada. Algumas regras são:

- Existe um diagrama para cada classe que possuir características dinâmicas;
- Os *estados* no diagrama representam os possíveis estados de um objeto da classe do diagrama. *Estados* são definidos por expressões em OCL (OMG, 2003) que envolvam os atributos e associações relevantes da classe;
- As pré e pós-condições representadas nas *transições* são expressas também em OCL;
- Os estados em um diagrama são mutuamente exclusivos;
- O *Diagrama de Estados* representa como as ações afetam um único objeto da classe. As ações devem afetar diretamente tal objeto;
- Uma mesma ação pode estar presente em diversos *Diagrama de Estados*, pois a mesma pode afetar mais de uma classe de objetos;

Para exemplificar a construção dos *Diagramas de Estados* na modelagem das características dinâmicas e, conseqüentemente, das ações será utilizado o domínio *Logística*, cuja estrutura estática foi definida nos tópicos anteriores.

Iniciando pela classe *Pacote*, pode-se modelar o comportamento desta classe levando em consideração os seguintes pontos:

- Supõe-se que um objeto do tipo *Pacote* pode ser encontrado em três estados possíveis, sendo eles "*Pacote dentro de um Caminhão*", "*Pacote dentro de um Avião*" e "*Pacote em um lugar*" (seja um aeroporto ou um local). Um objeto do tipo *Pacote* sempre inicia no estado "*Pacote dentro de um Avião*" (default) caso seu estado inicial não seja especificado;
- As ações que podem afetar um objeto do tipo *Pacote* são: *carregaCaminhao* e *descarregaCaminhao* (realizadas pelo *Caminhão*), *carregaAviao* e *descarregaAviao* (realizadas pelo Avião);
- O *Diagrama de Casos de Uso* deve ser utilizado para levantar as pré e pós-condições das ações que afetam os objetos da classe *Pacote*;
- As associações relevantes para os estados são: *estaEm* e *estaDentro*.

Focalizando exclusivamente na classe *Pacote* é possível traçar as transições entre os estados (as ações) e suas respectivas pré e pós-condições no contexto *Pacote.* A Figura 30 apresenta o *Diagrama de Estados* da classe *Pacote*.

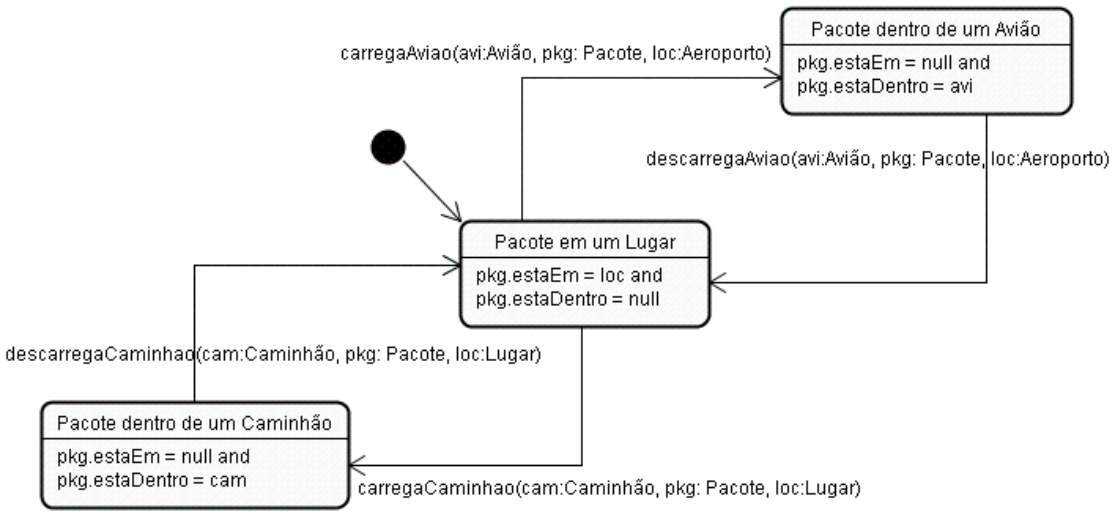


Figura 30 – *Diagrama de Estados* da classe *Pacote*.

Em relação à classe *Caminhão* alguns pontos relevantes são:

- Supõe-se que um objeto do tipo *Caminhão* pode ser encontrado em apenas um estado relevante, sendo este "*Caminhão parado em um lugar*";
- As ações que podem afetar um objeto do tipo *Caminhão* são: *carregaCaminhao*, *descarregaCaminhao* e *dirigir* (realizadas pela própria classe *Caminhão*);
- O *Diagrama de Casos de Uso* deve ser utilizado para levantar as pré e pós-condições das ações que os objetos da classe *Caminhão* realizam;
- A associação relevante para o estado é: *estaNo*. Neste caso de exemplo os atributos *capacidade* e *carregamentoCorrente* não são relevantes para definir os estados.

A Figura 31 apresenta o *Diagrama de Estados* da classe *Caminhão*. Esta abordagem de construção do *Diagrama de Estados* com a OCL (OMG, 2003) tende a facilitar a modelagem das pré e pós-condições das ações que acaba sendo uma das partes mais complicada na modelagem de domínios de planejamento, a modelagem de ações.

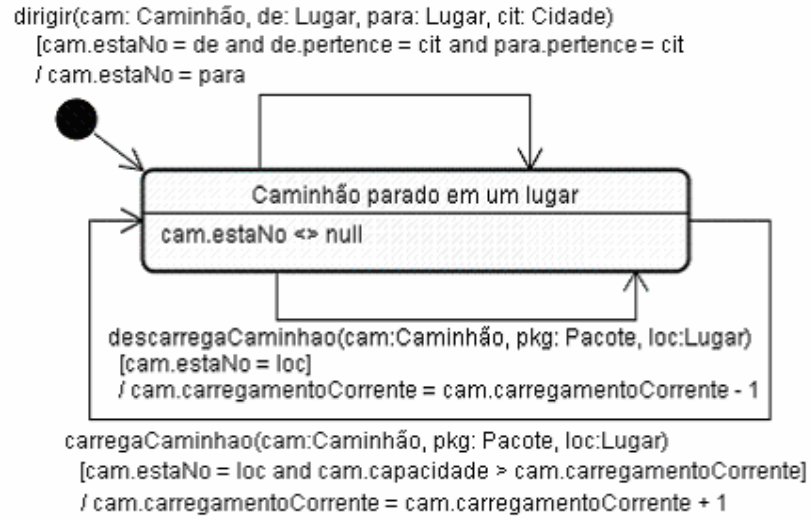


Figura 31 – *Diagrama de Estados* da classe *Caminhão*.

Já em relação à classe *Avião* alguns pontos relevantes são:

- Supõe-se que um objeto do tipo *Avião* pode também ser encontrado em apenas um estado relevante, sendo este "*Avião parado em um aeroporto*";
- As ações que podem afetar um objeto do tipo *Avião* são: *carregaAviao*, *descarregaAviao* e *pilotar*(realizadas pela própria classe *Avião*);
- O *Diagrama de Casos de Uso* deve ser utilizado para levantar as pré e pós-condições das ações que os objetos da classe *Avião* realizam;
- A associação relevante para o estado é: *estaParadoNo*. Novamente, neste caso de exemplo os atributos *capacidade* e *carregamentoCorrente* não são relevantes para definir os estados.

A Figura 32 apresenta o *Diagrama de Estados* da classe *Avião*.

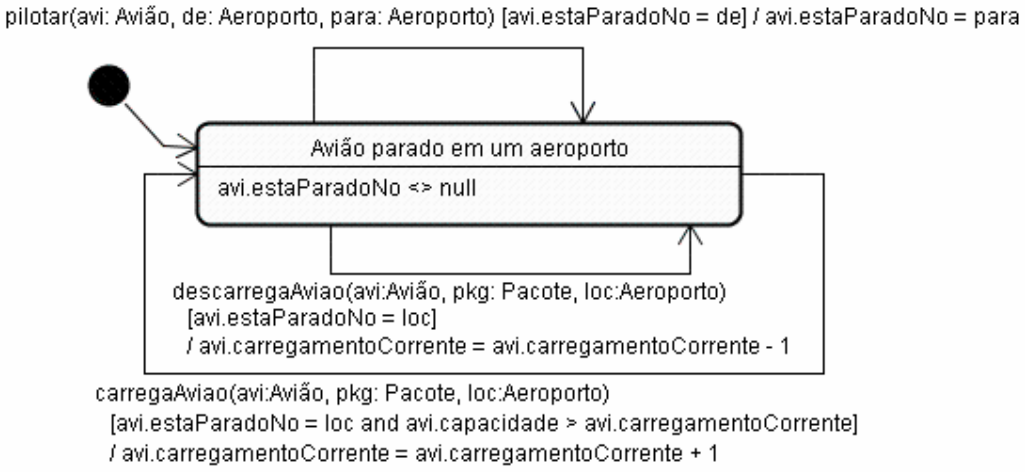


Figura 32 – *Diagrama de Estados* da classe *Avião*.

De fato, a união das expressões da OCL declaradas em todos os diagramas forma a definição completa de cada ação. Tomando como base a Figura 30, Figura 31 e Figura 32 é possível representar em OCL as ações resultantes da união dos diagramas. A seguir são

representadas as ações *dirigir* (realizada pelo *Caminhão*) e *carregarAviao* (realizada pelo *Avião*) em OCL na abordagem UML.P.

```
context Caminhão:: dirigir(cam: Caminhão, de:Lugar, para:Lugar, cit:Cidade)
   pre:
          -- contexto do Caminhão
          -- O caminhão deve estar no lugar de partida (de) que pertence à mesma
          --  cidade do lugar de destino (para)
          cam.estaNo <> null and cam.estaNo = de and de.pertence = cit and
          para.pertence = cit and
   post:
          -- contexto do Caminhão
          cam.estaNo = para



context Avião:: carregaAviao(avi:Avião, pkg:Pacote, loc:Aeroporto)
   pre:
          -- contexto do Avião
          -- O Avião deve estar em um aeroporto (loc) e sua capacidade deve ser
          --  maior que seu carregamento corrente para que um pacote seja
          --  carregado
          avi.estaParadoNo <> null and avi.estaParadoNo = loc and avi.capacidade
          > avi.carregamentoCorrente and

          -- contexto do Pacote
          -- o Pacote deve também estar no mesmo local
          pkg.estaEm = loc and pkg.estaDentro = null

   post:
          -- contexto do Caminhão
          avi.carregamentoCorrente = avi.carregamentoCorrente + 1 and

          -- contexto do Pacote
          pkg.estaEm = null and pkg.estaDentro = avi
```

No exemplo acima da ação *carregarAviao,* representada em OCL, é possível visualizar quais expressões são provenientes do *Diagrama de Estados* da classe *Avião* e quais são provenientes do *Diagrama de Estados* da classe *Pacote*. Esta separação é ilustrada pelo comentário "`contexto do Avião`" para a classe *Avião* e pelo comentário "`contexto do Pacote`" para a classe *Pacote*, tanto na pré-condição (**`pre`**) quanto na pós-condição (**`post`**).

Como visto na modelagem da estrutura estática e das características dinâmicas, os elementos dos conjuntos A,, *Ru*, e *Ac* de um domínio $\mathcal{D}$ foram representados e identificados. De fato, os elementos de S podem ser inferidos, por exemplo, do *Diagramas de Estados* construídos.

3.3.2.2.1.3. Agentes e Recursos

Neste trabalho os objetos Agentes e os Recursos do domínio (*Ag* e *Rs*) são representados utilizando o *Diagrama de Objetos*. É neste diagrama que os agentes e os recursos do domínio, com seus respectivos tipos e nomes, são declarados, bem como as características constantes (imutáveis) do domínio.

Para exemplificar a declaração dos Agentes (*Ag*) e Recursos (*Rs*), a Figura 33 demonstra a representação tanto de Agentes quanto de Recursos de um domínio *Logística* (representado na Figura 24) com o *Domínio Abstrato* representado nas secções anteriores. Neste domínio existem duas cidades *sãopaulo* e *riodejaneiro* e cada uma das cidades possui dois *Locais* e um *Aeroporto* (estes objetos constituem a topologia do problema). Existem também quatro pacotes *pkg1, pkg2, pkg3* e *pkg4* a serem entregues e quatro agentes *caminhão1*, *caminhão2*, *avião1* e *avião2* conforme mostra a Figura 33.

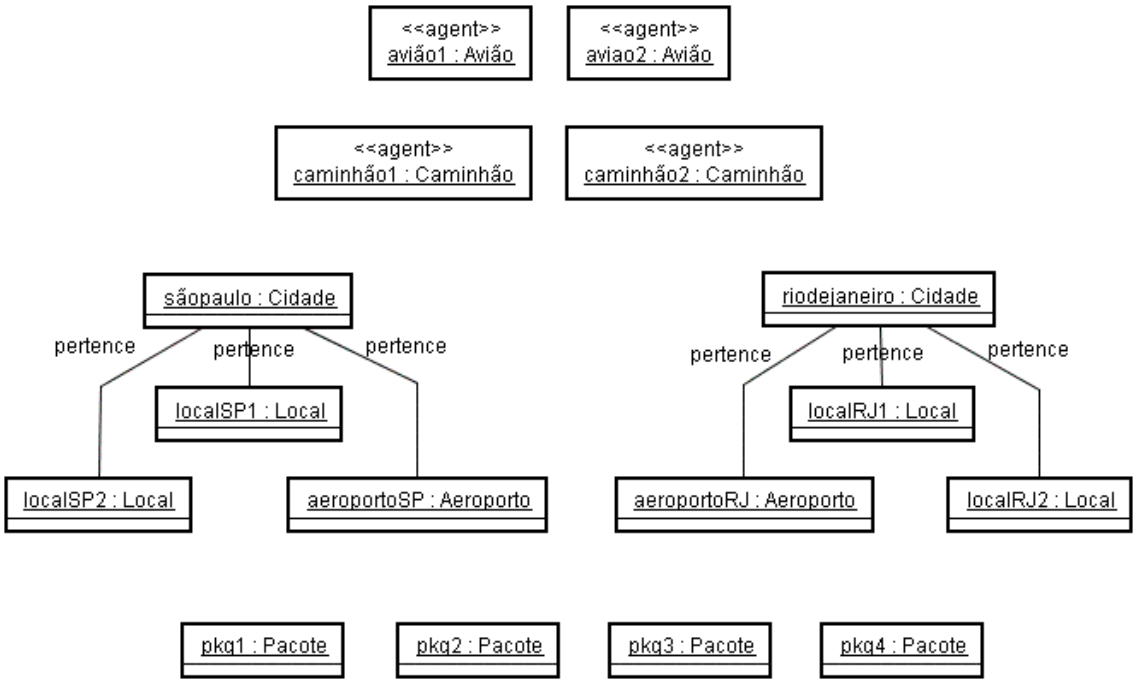


Figura 33 – Agentes e Recursos declarados através do *Diagrama de Objetos.*

Na Figura 33 tem-se que o conjunto de Agentes é *Ag* = {*avião1, avião2, caminhão1, caminhão2*}. Já os Recursos são *Rs* = {*pkg1, pkg2, pkg3, pkg4, sãopaulo, riodejaneiro, localSP1, localSP2, localRJ1, localRJ2, aeroportoSP, aeroportoRJ*}. Nesta figura é também possível visualizar algumas invariantes, como por exemplo, o fato do recurso *localSP1* pertencer (relação *pertence* com significado físico) à cidade *sãopaulo*. Estas são características imutáveis do domínio.

A utilização do *Diagrama de Objetos* para representação dos Agentes e Recursos é também uma característica presente na UML.P.

3.3.2.2.2. Modelagem do Problema de Planejamento

É possível modelar problemas de planejamento quando a modelagem do domínio foi concluída através dos *Diagramas de Classes, Estados* e *Objetos*. Conforme visto anteriormente, um problema de planejamento $\mathscr{P}$ referente a um modelo de domínio $\mathscr{D}$ de planejamento é geralmente caracterizado por uma situação onde dois estados são conhecidos: o estado inicial (S_0) e o estado objetivo (S_G). O diagrama utilizado para descrever tais estados é também o *Diagrama de Objetos* que neste contexto é chamado de *Snapshot* (D'SOUZA; WILLS, 1999).

3.3.2.2.2.1. Snapshots

O *Snapshot* representa uma fotografia de um determinado estado ou momento dos objetos agentes e recursos do domínio. Esta fotografia representa como os objetos existentes no domínio se relacionam e quais os valores de seus atributos em um determinado estado ou instante (se o tempo for considerado) que, no caso de planejamento, os mais importantes são o estado inicial ou o estado objetivo (final) (D'SOUZA; WILLS, 1999).

De fato, um problema de planejamento é composto por dois *Snapshots*. O primeiro descreve o arranjo dos objetos e seus atributos no estado inicial, e o segundo descreve o arranjo dos objetos e seus atributos no estado objetivo. A representação do *Snapshot* deve seguir rigorosamente a regras e restrições definidas no em $\mathcal{D}$, principalmente as regras definidas no *Diagrama de Classes*. As características provenientes das associações, multiplicidades, atributos, restrições e classes devem ser respeitadas durante a definição dos problemas de planejamento de modo a construir estados válidos em relação à estrutura estática definida (VAQUERO; TONIDANDEL; SILVA, 2005).

Um exemplo simples de um problema de planejamento do domínio *Logística* apresentado na Figura 33 poderia ser descrito da seguinte forma:

- **Estado Inicial**: O *caminhão1*, com *capacidade* para 4 pacotes e com nenhum carregamento, sem encontra no *localSP1* da cidade *saopaulo*. Os pacotes *pkg1* e *pkg2* se encontram no *localSP2* e o *avião1* (com *capacidade* para 10 pacotes e com nenhum carregamento) se encontra no *aeroportoSP1* também da cidade *sãopaulo*. Já o *caminhão2*, com *capacidade* para 2 pacotes e com nenhum carregamento, *se encontra no localRJ1 da cidade riodejaneiro.* O *pkg3* se encontra no *localRJ2* e o pacote *pkg4* e *avião2* (com *capacidade* para 10 pacotes e com nenhum carregamento) se encontram no *aeroportoRJ* também da cidade *riodejaneiro*.
- **Estado Objetivo**: no estado desejado os pacotes *pkg1* e *pkg2* devem ser entregues no *localRJ1* da cidade *riodejaneiro* e os pacotes *pkg3* e *pkg4* devem ser entregues no *localSP1* da cidade s*ãopaulo*.

Para uma melhor visualização destes estados a Figura 34 e a Figura 35 ilustram as descrições acima.

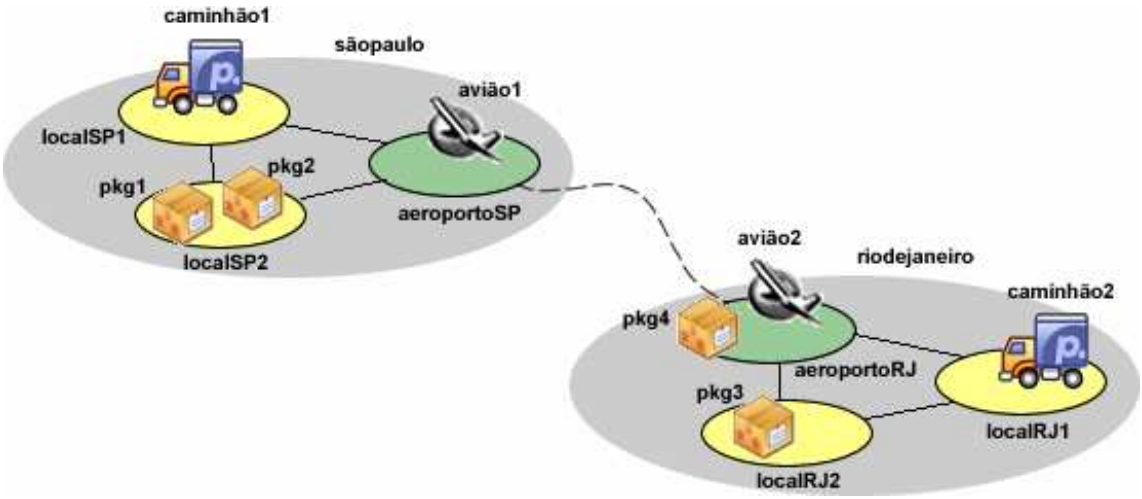


Figura 34 – Ilustração do Estado Inicial de um problema de planejamento no domínio *Logística*.

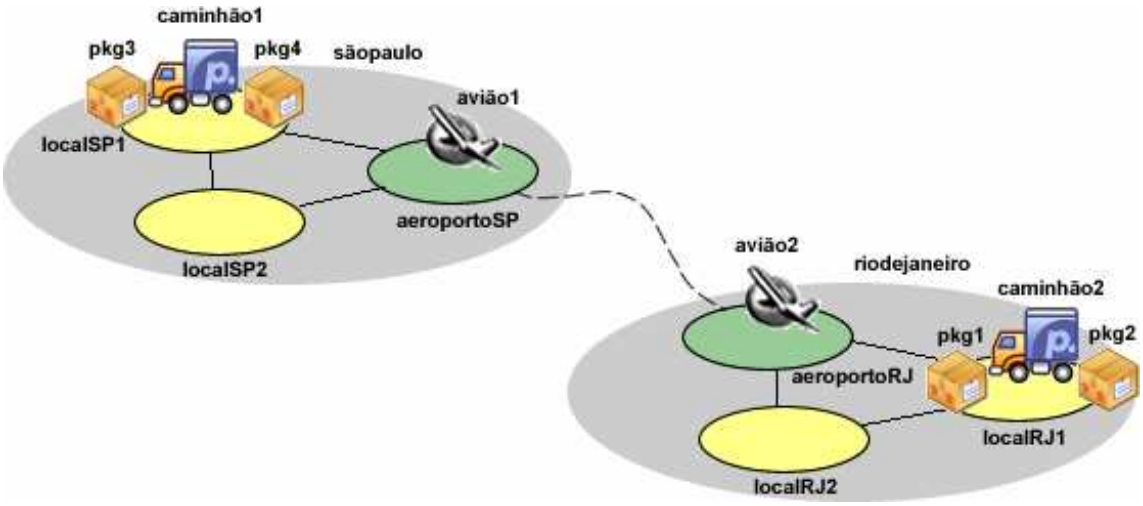


Figura 35 – Ilustração do Estado Objetivo de um problema de planejamento no domínio *Logística*.

A Figura 36 (Snapshot inicial) e a Figura 37 (Snapshot objetivo) demonstram os estados inicial e objetivo utilizando o *Diagrama de Objetos*. Com isso o planejador pode iniciar o processo de planejamento onde se buscam seqüências de ações para se alcançar o estado objetivo a partir do estado inicial utilizando as ações disponíveis para o problema.

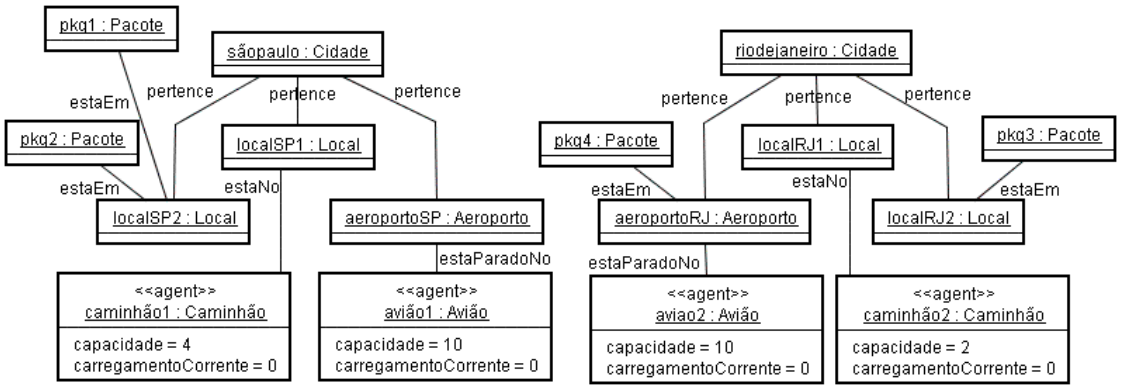


Figura 36 – Snapshot inicial de um problema de Logística.

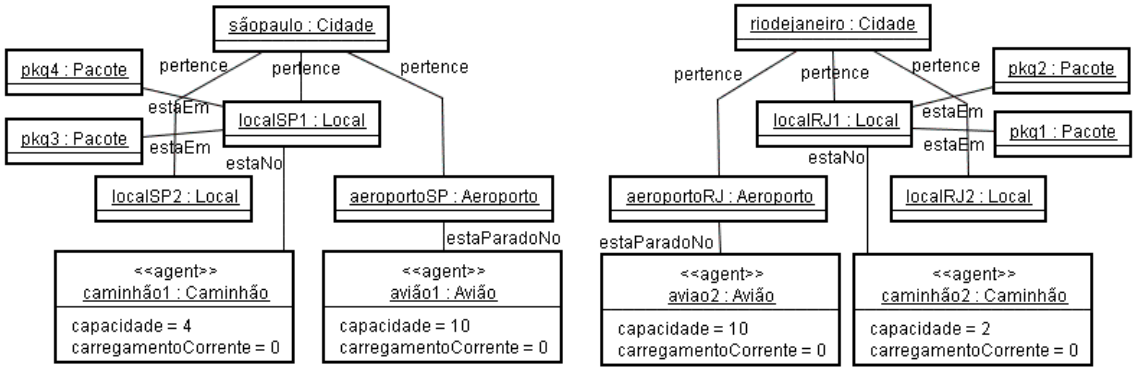


Figura 37 – Snapshot objetivo de um problema de Logística.

A utilização do *Diagrama de Objetos* para representação do problema de planejamento é também uma característica presente na UML.P.

3.3.3. Análise do Modelo do Domínio

O processo de *Análise do Modelo do Domínio* está diretamente relacionado com as atividades de *Verificação*, *Validação* e *Refinamento* do modelo do domínio de planejamento que está sendo construído. O processo de análise do modelo leva o projetista a investigar as características do modelo e confrontá-las com as características do domínio real. A análise

pode ser realizada sobre grupos de características especificas do modelo como, por exemplo, características estáticas, funcionais e dinâmicas (RUMBAUGH et al., 1991), e esses grupos de características podem ser analisados tanto isoladamente quanto em conjunto. Para os domínios de planejamento, em geral, os maiores desafios se encontram na análise das características dinâmicas, ou seja, na verificação do comportamento dos objetos no domínio.

De fato, erros, inconsistências e incoerências encontrados nesta etapa de análise podem economizar tempo e recursos em fases posteriores como, implementação e testes. Mesmo assim, poucos métodos e ferramentas de análise de modelos de domínio de planejamento são encontrados na literatura. As poucas ferramentas disponíveis são consideradas apenas verificadores de sintaxe ou removedores de inconsistência (*Debugging*) (MCCLUSKEY, 2002).

Atualmente, muitos pesquisadores utilizam o próprio planejador como ferramenta de análise (principalmente para verificação) do modelo do domínio de planejamento devido, justamente, à escassez de métodos e ferramentas com este foco, porém os planejadores não foram projetados para este fim. Este processo de análise utilizando o planejador acaba sendo um processo de refinamento do modelo baseado em tentativa e erro, onde o projetista faz com que o planejador resolva alguns problemas de planejamento (geralmente aleatórios) raciocinando sobre o modelo do domínio em questão. A utilização de planejadores como primeira alternativa na análise do modelo acaba fazendo com que processos de verificação sejam exaustivos, pois muitas vezes o projetista se depara com conflitos onde não se sabe se o planejador não possui capacidade para tratar o modelo ou se realmente o modelo não está correto.

Diante deste cenário, a *Análise do Modelo do Domínio* apresentada neste trabalho foca especificamente nos aspectos dinâmicos do modelo do domínio onde justamente são encontrados grandes desafios. A presente proposta de análise visa principalmente a

verificação das características dinâmicas modeladas nos *Diagramas de Estados*, conforme proposto nos tópicos anteriores. Os principais objetivos desta análise são: verificar como as ações mudam os estados dos objetos das classes; verificar a evolução dos estados das classes; e verificar as interações entre as classes durante a realização das ações.

O uso dos diagramas da UML que representam as características dinâmicas para o processo de análise não é suficiente, pois, além dos diagramas serem semi-formais, geralmente o projetista usa mais de um diagrama para representar os aspectos dinâmicos, o que dificulta , por exemplo, garantir a coerência do comportamento dos objetos em todos os diagramas (DÖLL, 2003). É preciso verificar se as informações representadas em diversos diagramas são coerentes entre si. Infelizmente, não existe na UML um diagrama que represente todo o comportamento dinâmico do modelo do domínio através, por exemplo, de um modelo de estados global. De fato, enumerar todos os estados de um domínio em um único *Diagrama de Estados*, por exemplo, seria inviável para a maioria das aplicações de planejamento devido à explosão combinatória de estados possíveis.

Devido ao fato dos diagramas da UML não serem suficientes para a Análise do Modelo do Domínio nos aspectos dinâmicos, o presente projeto propõe o uso das Redes de Petri (MURATA, 1989) na realização desta tarefa, onde os *Diagramas de Estados* gerados na modelagem são representados em RdP para que as verificações (com eventuais simulações) sejam feitas. A grande vantagem do uso das Redes de Petri no *Ambiente* proposto é que esta provê um maior formalismo para o modelo, provê mecanismos de simulação e técnicas de análise como, por exemplo, identificação de invariantes, *deadlock* entre outras (MURATA, 1989). Alguns trabalhos (mais direcionados à engenharia de software) mostram o potencial da aplicação das RdP para verificação e validação dos aspectos dinâmicos de modelos gerados em UML, muitos deles utilizando o *Diagrama de Estados* como referência (WATANABE et

al., 1997) (CHEUNG; CHOW; CHEUNG, 1998) (PALUDETTO; DELATOUR, 1999) (BARESI; PEZZÈ, 2001) (SALDHANA; SHATZ, 2000) (DÖLL, 2003).

O tópico a seguir apresenta como os *Diagramas de Estados* são representados em Redes de Petri onde o processo de análise (neste trabalho restrito apenas a atividades de verificação) é realizado.

3.3.3.1. Análise dos Diagramas de Estados através das Redes de Petri

Como visto na proposta de modelagem com a UML.P, cada classe do *Digrama de Classes*, com características dinâmicas, possui um *Diagrama de Estados* que representa não só os estados possíveis de um objeto daquela classe, mas também como as ações afetam e modificam o estado do objeto. Com esta construção dos *Diagramas de Estados* para cada classe, é possível visualizar o domínio (ainda no nível abstrato das classes) modularmente, ou seja, o comportamento de cada classe do domínio representa um módulo e a união dos módulos fornece uma visão geral do comportamento do domínio. Esta abordagem de divisão de contexto (divisão por módulo) realizado na UML na criação dos *Diagramas de Estados* pode favorecer tanto o processo de modelagem quanto o de análise.

O processo de análise proposto neste trabalho visa estudar cada módulo (proveniente do *Diagrama de Estados* de uma classe) tanto separadamente quando em conjunto. Para isso, cada módulo é representado por uma Rede de Petri lugar/transição (estendida, conforme será visto nos próximos tópicos) através dos conceitos de modularidade provenientes da *PNML modular* (KINDLER; WEBER, 2001). Como visto no Capitulo 2, na seção da *PNML modular*, as redes podem ser visualizadas e analisadas isoladamente ou em conjunto através de uma representação modular (KINDLER; WEBER, 2001). Neste trabalho, a passagem de cada *Diagrama de Estados* para uma Rede de Petri não leva em consideração as expressões

em OCL, e sim apenas os elementos estados e transições (ações) do diagrama, pois o foco principal num primeiro momento é verificar a coerência do fluxo de ações e os estados. Assim, a presença da OCL no processo de análise não é tratada neste trabalho, ficando para futuros trabalhos.

O processo de análise dos *Diagramas de Estados* através de Redes de Petri proposto neste trabalho é dividido em duas análises: *Análise Modular*, *Análise de Interfaces* (análise das interfaces entre os módulos). Não serão tratadas aqui análises com as instâncias de agentes e de recursos, pois esta proposta de análise coloca foco em níveis mais abstratos do modelo como é o caso do *Diagrama de Estados* que representa o comportamento de um conjunto de objetos. A Figura 38 demonstra o fluxo de execução das análises.

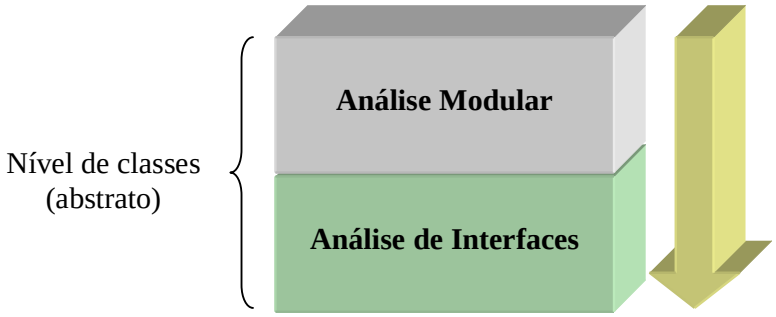


Figura 38 – Principais análises realizadas sobre os aspectos dinâmicos do modelo.

Vale citar que o processo de análise presente neste trabalho contempla apenas *Diagrama de Estados* na forma de uma máquina de estados (ou seja, cada transição do diagrama deve possuir exatamente um estado de saída e um estado de entrada). Nesta forma, os elementos do *Diagrama de Estados* (estados e ações) são facilmente representados em Redes de Petri.

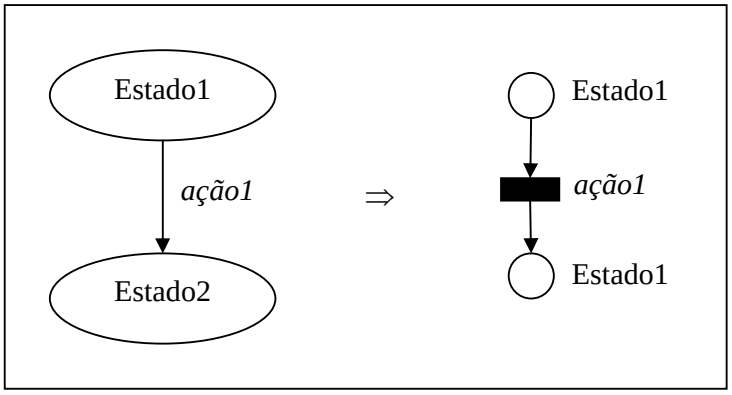


Figura 39 – Elementos do *Diagrama de Estados* representados em Redes de Petri.

É importante mencionar também que as análises (tanto *Modular* quanto de *Interfaces*, ou ambas) podem ser realizadas durante o processo de modelagem para que eventuais refinamentos sejam efetuados.

Nos tópicos a seguir será explicada cada análise identificada na Figura 38, bem como o processo de passagem dos *Diagramas de Estados* para as Redes de Petri. Para demonstrar as análises, o modelo do domínio *Logística*, construído anteriormente, será utilizado como caso de exemplo.

3.3.3.1.1. Análise Modular

A *Análise Modular* visa verificar cada módulo de forma isolada, mas levando em consideração as dependências entre os outros módulos. O objetivo é analisar cada parte (ou seja, cada comportamento de classe) dos aspectos dinâmicos do modelo para que inicialmente tenha-se um refinamento localizado. A análise dos módulos através de uma RdP faz com que o projetista verifique características como, por exemplo, paralelismo, concorrências e eventuais *deadlocks* indesejáveis.

Para que a representação de um módulo *m*, já representado por um *Diagrama de Estados* da classe *C*, em RdP seja realizada, levando em consideração os conceitos da *PMNL modular*, as seguintes regras são necessárias:

i. Todas as transições *t* em *m* que representam ações a que não são realizadas pela classe *C* são consideradas aqui *transições importadas* de outros módulos; A execução destas *transições importadas* depende pelo menos da classe que a executa (a classe agente). O elemento *transição importada* em uma Rede de Petri é ilustrado em (a) da Figura 40;

ii. Todas as transições *t* em *m* que representam ações a realizadas pela própria classe *C* (isto é, a classe *C* é agente destas ações), mas que suas execuções dependam de outros módulos (de outras classes), são consideradas aqui *transições exportadas* (t_e) de outros módulos. As *transições exportadas* estarão presentes em outros módulos, pois estas devem afetar o comportamento de outras classes. O elemento *transição exportada* em uma Rede de Petri é ilustrado em (b) da Figura 40;

iii. Todas as transições *t* em *m* que representam ações a realizadas pela classe *C* (isto é, a classe *C* é agente destas ações), e que sua execução dependa exclusivamente da classe *C* são consideradas aqui *transições normais.* As transições normais aparecem apenas no módulo *M*. O elemento *transição normal* em uma Rede de Petri é ilustrado em (c) da Figura 40;

iv. Para representar as dependências de outros módulos tanto para as *transições importadas* quanto para as *transições exportadas*, são adicionados lugares (chamados aqui de *Gates* conforme trabalho (SILVA; DEL FOYO, 2003)) e arcos que habilitam ou desabilitam as transições em questão. Ou seja, para cada transição que possui dependência de outros módulos é adicionado um *Gate* e um arco habilitador direcionado a esta transição. O elemento *Gate* e seu arco habilitador para um transição em uma Rede de Petri são ilustrados em (d) da Figura 40;

v. As *transições importadas, exportadas* e *normais* possuem regras de disparo exatamente iguais a uma Rede de Petri transição/evento. Aqui elas são apenas identificadas visualmente para separar contextos.

vi. Estados de *m* são representados com lugares da RdP, inclusive o estado final (elemento que representa o estado final do objeto no *Diagrama de Estados*). O elemento *estado* em uma Rede de Petri é ilustrado em (e) da Figura 40.

vii. Cada módulo possui apenas uma marca e esta marca é posicionada inicialmente no lugar que representa o estado inicial de um objeto da classe *C*.

viii. A Rede de Petri representando um módulo deve ser conservativa (MURATA, 1989), pois um objeto da classe *C* deve se encontrar sempre em um dos lugares (estados) da rede.

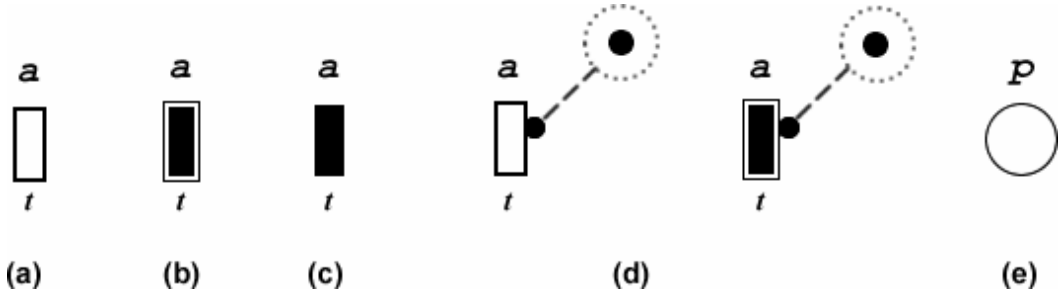


Figura 40 – Elementos da Rede de Petri da representação de um módulo.

Levando em consideração as regras acima citadas, a passagem de um módulo representado em *Diagrama de Estados* para uma RdP se torna uma tarefa simples. Seguindo o exemplo do domínio *Logística* os módulos em Rede de Petri provenientes dos *Diagramas de Estados*, por exemplo, da classe *Pacote* e da classe *Avião* são apresentados na Figura 41 e na Figura 42.

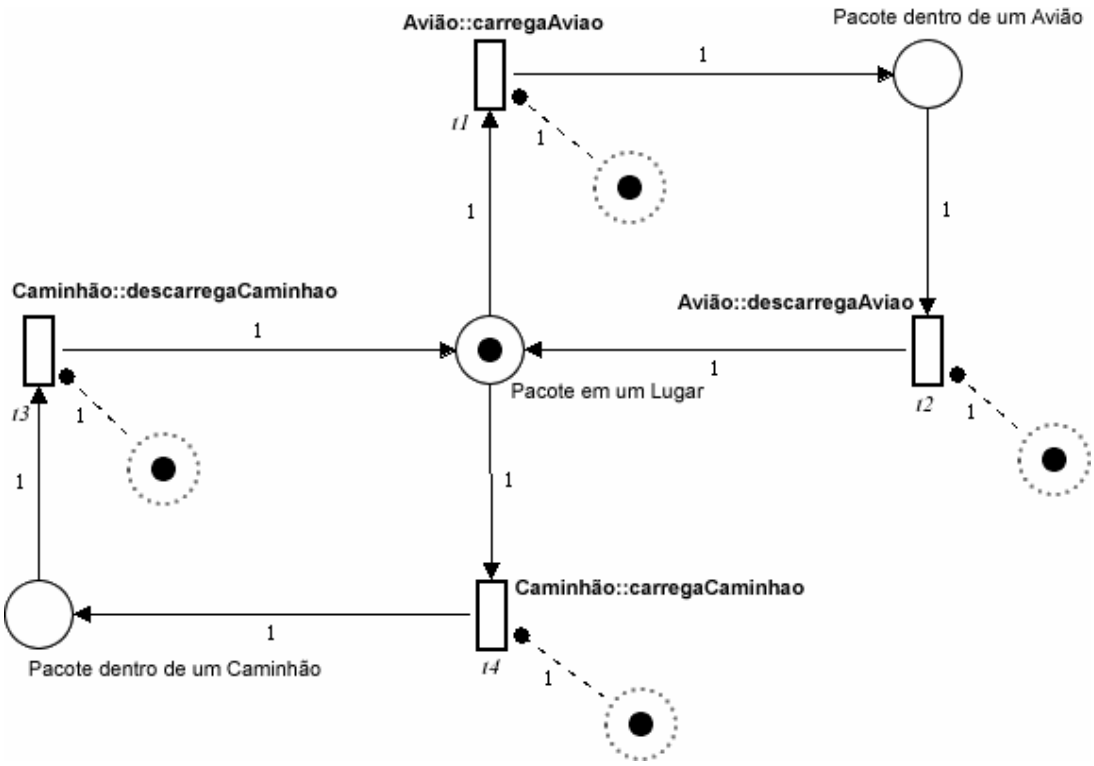


Figura 41 – Módulo da classe *Pacote* em Rede de Petri.

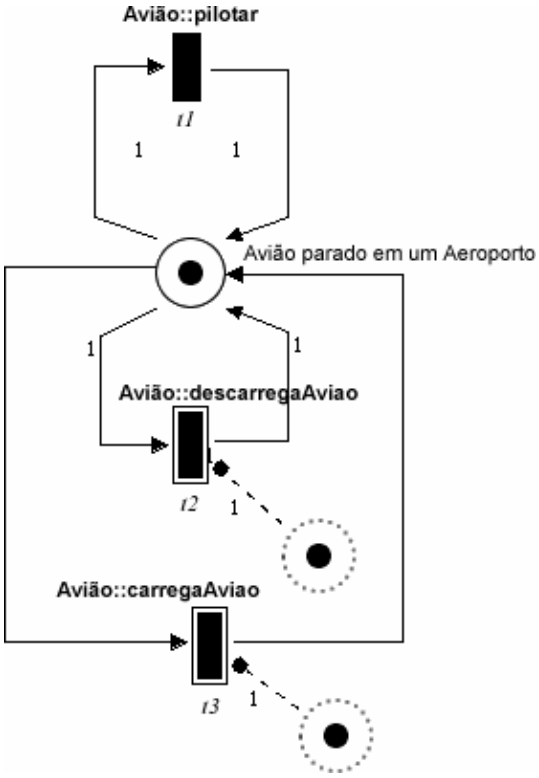


Figura 42 – Módulo da classe *Avião* em Rede de Petri.

Para analisar os módulos, o projetista pode verificar o fluxo das marcas em cada módulo, verificar a existências de *deadlocks* indesejáveis, verificar a coerência das concorrências e dos paralelismos e também manipular os *Gates* para analisar o comportamento do módulo caso as condições externas do mesmo sejam modificadas. A manipulação dos *Gates* permite ao projetista habilitar ou desabilitar as condições de transições com dependências externas ao módulo e assim verificar o comportamento da marca como, por exemplo, o projetista pode visualizar o que acontece no módulo do *Pacote* quando o avião não está disponível para carregá-lo. Com as verificações, o projetista pode refinar o modelo, caso seja necessário, utilizando o *Ambiente*, onde o mesmo pode entrar e sair dos contextos de modelagem e análise quando desejado.

Para o exemplo do domínio *Logística* a passagem do módulo representado no *Diagrama de Estados* para uma Redes de Petri (estendida devido aos arcos habilitadores (PETERSON, 1981)) acaba sendo um processo trivial (observadas as restrições), mas para domínios mais complexos, onde as classes possuem comportamentos mais complexos, a análise através das Rede de Petri se torna extremamente vantajosa.

Algumas técnicas de análise de propriedades de Redes de Petri estão disponíveis na literatura que podem enriquecer a presente proposta de análise como, por exemplo, as técnicas tradicionais de análise de propriedades (invariantes, árvore de alcançabilidade, análise de *deadlocks*, entre outras) disponíveis em (MURATA, 1989) e outras técnicas mais elaboradas como a análise modular de redes disponíveis em (CHRISTENSEN; PETRUCCI, 2000). Estas técnicas de análises de RdP não serão abordadas neste trabalho mas podem ser livremente utilizadas nas redes geradas na presente análise modular.

Um ponto importante a ser citado é que algumas técnicas de análise de Redes de Petri utilizam a *Equação de Estados* (MURATA, 1989) e esta exige que a rede seja pura, isto é, sem a presença de *self-loop* (como é o caso da transição *pilotar* na Figura 42). Estes casos são

resolvidos de maneira simples, bastando transformar um *self-loop* em um *loop*, ou seja, introduzir um lugar e uma transição conforme ilustrado na Figura 43 (MURATA, 1989).

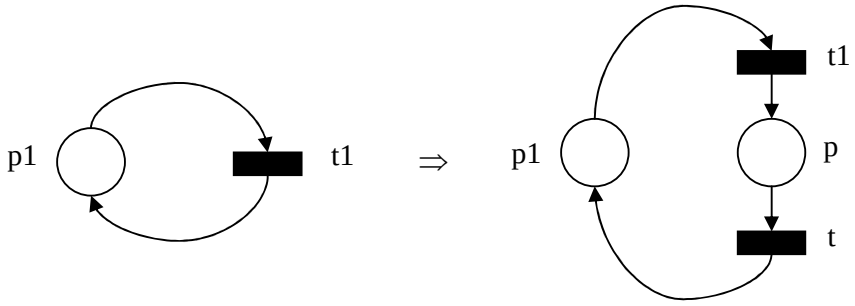


Figura 43 – Transformação de um *sef-loop* em um loop (MURATA, 1989).

3.3.3.1.2. Análise de Interfaces

A *Análise de Interface* visa verificar as interações e inter-relações entre as partes (os módulos) do modelo em relação aos aspectos dinâmicos. Nesta análise o projetista pode verificar pelo menos 2 (dois) módulos juntos, ou até mesmo todos, fazendo com que o mesmo visualize as interdependências entre as classes durante a execução de ações do domínio.

A representação dos módulos (já representados em *Diagramas de Estados*) a serem analisados em Redes de Petri (no caso, uma única RdP) segue as mesmas regras da análise modular, porém, como neste caso existem mais de um módulo envolvido, algumas regras adicionais são necessárias. As regras que serão mostradas a seguir são restritas ao caso onde uma ação a qualquer em um *Diagrama de Estados* qualquer d (pertencentes ao conjunto de diagramas/módulos a serem analisados) nunca aparece mais de uma vez no diagrama d em questão. O caso onde isso não ocorre, onde uma ação aparece mais de uma vez em um mesmo módulo, será apenas discutido ao final deste tópico. Assim, seja $\mathtt{M} \subseteq \mathbf{M}$ o conjunto de módulos a ser analisado (onde $\mathbf{M}$ é o conjunto de todos os módulos do domínio), e $m \in \mathtt{M}$ um

módulo que representa o comportamento de uma classe *C,* o conjunto de módulos $\mathtt{M}$ a ser analisado é representado em RdP com as seguintes regras:

i. Cada módulo é inicialmente representado da mesma maneira daquela apresentada na *Análise Modular*, mas sem a representação dos *Gates*.

ii. As *transições importadas* e *exportadas* (t_i e t_e), presentes nos módulos de $\mathtt{M}$ e que representam uma mesma ação a, são fundidas em apenas uma transição. Esta transição, resultante da fusão, será uma *transição exportada* se e somente se existir uma outra transição em um módulo externo $m_{ex} \notin \mathtt{M}$ ($m_{ex} \in \mathbf{M}$) que represente a mesma ação a. Caso contrário, a transição resultante da fusão será uma *transição normal*. Esta regra onde as ações são fundidas só é válida quando para todos os módulos $m \in \mathtt{M}$, uma ação a qualquer só aparece no máximo uma vez em cada m.

iii. Para representar as dependências de outros módulos que não estão presentes em $\mathtt{M}$, tanto para as *transições importadas* quanto para as *transições exportadas*, são adicionados os *Gates* e seus respectivos arcos habilitadores. Ou seja, se uma ação a está representada nos módulos de $\mathtt{M}$ e também em módulos não existentes em $\mathtt{M}$, esta ação a possui dependências externas ao conjunto $\mathtt{M}$ que serão representadas através de um *Gate*;

iv. Novamente, as *transições importadas, exportadas* e *normais* possuem regras de disparo exatamente iguais a uma Rede de Petri transição/evento;

v. Todos os estados de todos os módulos de $\mathtt{M}$ são representados através lugares da RdP, inclusive o estado final (elemento que representa o estado final do objeto no *Diagrama de Estados*).

vi. A Rede de Petri representando a união dos módulos deve ser conservativa (MURATA, 1989).

Para exemplificar a *Análise de Interfaces* proposta, levando em consideração as regras acima listadas, a Figura 44 demonstra a Rede de Petri para analisar as inter-relações entre as classes *Pacote* e *Avião*.

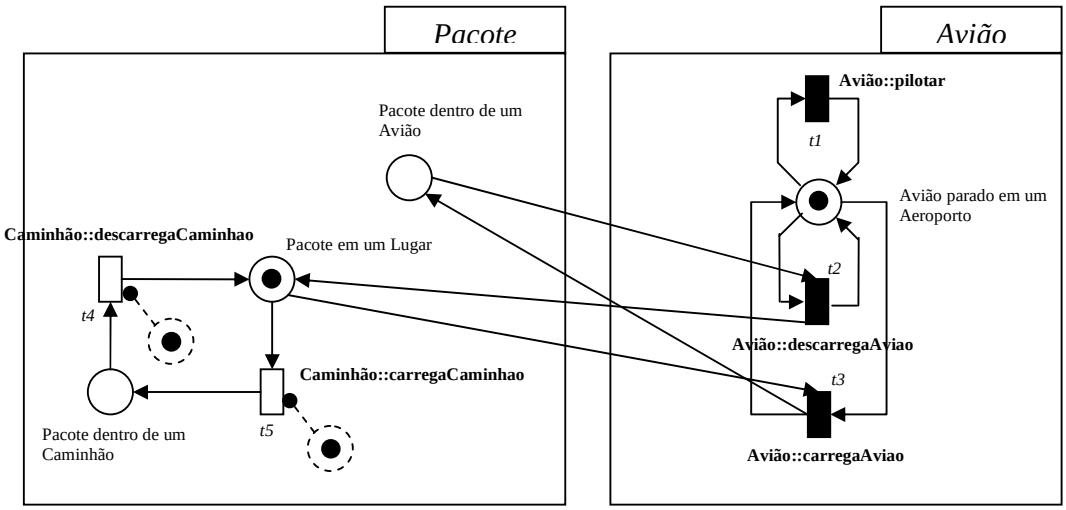


Figura 44 – *Análise de Interfaces* do comportamento das classes *Pacote* e *Avião*.

O exemplo da Figura 44 mostra como as ações são fundidas, como é o caso da ação *carregaAviao* (da classe Avião) que estava presente tanto no módulo da classe *Pacote* quanto da classe agente *Avião*. Com essa representação em RdP é possível verificar as dependências da ocorrência de uma ação (abstração das pré-condições) que, no caso da ação *carregaAviao* seriam "*Pacote em um Lugar*" (seja este um local ao um aeroporto) e "*Avião parado em um Aeroporto*", bem como seus efeitos (de uma forma abstrata) que são "*Pacote dentro de um Avião*" e "*Avião parado em um Aeroporto*".

Um caso particular da *Análise de Interfaces*, novamente considerando as regras acima citadas, é quando todos os módulos do domínio (conjunto `M`) são representados na rede. Neste caso é possível verificar todas as interações entre todos os comportamentos das classes. A Figura 45 ilustra a rede para a análise de todos os módulos do domínio *Logística*.

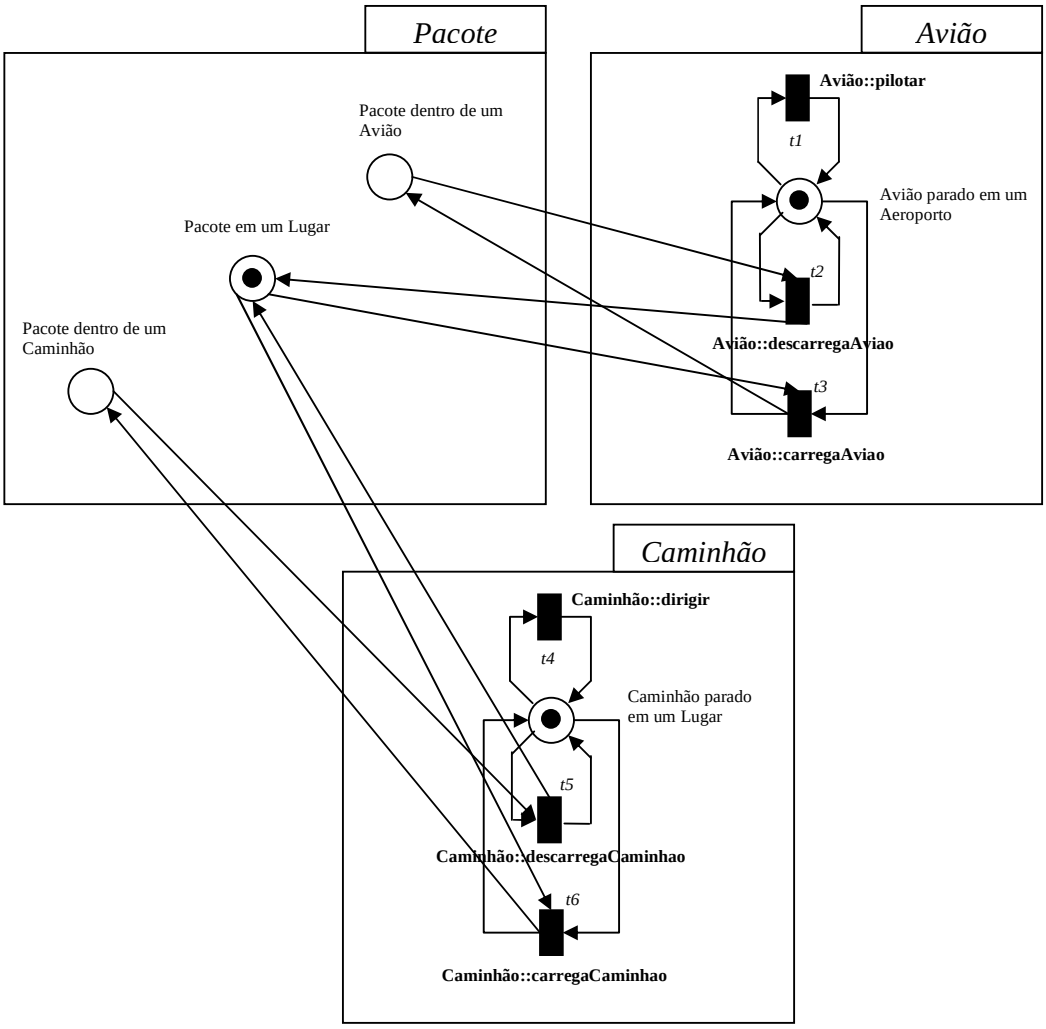


Figura 45 – *Análise de Interfaces* de todos os módulos do domínio *Logística*.

É possível perceber que tanto na Figura 44 quanto na Figura 45 as ações não se repetem em um mesmo módulo, como, por exemplo, a ação *carregaCaminhao* aparece no máximo uma vez em cada módulo. Caso uma ação aparece mais de ume vez em um ou mais módulos a serem analisados, a regra de fusão das transições (exportadas e/ou importadas), conforme apresentadas anteriormente (regra ii), são diferentes nestes casos. A Figura 46 demonstra um caso de exemplo onde dois módulos, *M1* e *M2*, devem ser analisados pela *Análise de Interfaces* e a ação *a* que aparece duas vezes no *M1*.

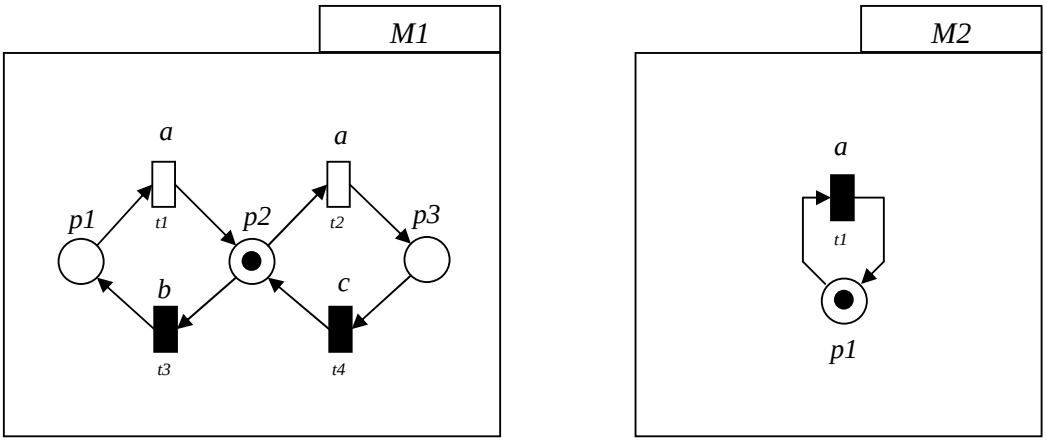


Figura 46 – Caso onde ações se repetem em um mesmo módulo na *Análise de Interfaces*.

Neste trabalho, estes casos são tratados da seguinte maneira:

- Seja $T_a^{mi} = \{ t1_a^{mi}, t2_a^{mi}, \ldots tn_a^{mi} \}$ o conjunto de transições de um módulo $m_i \in \mathtt{M}$ que representam uma mesma ação *a*, onde $\mathtt{M} = \{m1, m2, ..., mn\}$ é o conjunto dos módulos a serem analisados. O conjunto T de transições, presentes na Rede de Petri resultante da união dos módulo, que representam a ação *a* é o produto cartesiano de todos os conjuntos T_a^{mi} (com i = 1, 2, ..., n), ou seja, $\mathtt{T} = T_a^{m1} \times T_a^{m2} \times \ldots \times T_a^{mn}$. Por exemplo, se a análise é feita sobre três módulos (*m1, m2* e *m3*), onde $T_a^{m1} = \{ t1_a^{m1}, t2_a^{m1} \}$, $T_a^{m2} = \{ t1_a^{m2} \}$ e $T_a^{m3} = \{ t1_a^{m3}, t2_a^{m3} \}$, a rede resultante teria 4 transições que representariam *a* ação que ligam seus respectivos lugares convenientemente, ou seja, o conjunto $\mathtt{T} = T_a^{m1} \times T_a^{m2} \times T_a^{m3}$ teria 4 elementos onde $\mathtt{T} = \{t_(t1_a^{m1}, t1_a^{m2}, t1_a^{m3}), t_(t1_a^{m1}, t1_a^{m2}, t2_a^{m3}), t_(t2_a^{m1}, t1_a^{m2}, t1_a^{m3}), t_(t2_a^{m1}, t1_a^{m2}, t2_a^{m3})\}$.

No caso dos módulos da Figura 46, tem-se que T_a^{M1} possui dois elementos e T_a^{M2} possui apenas um elemento. Para melhor visualizar a regra acima citada, a Figura 47 demonstra o resultado da união dos módulos *M1* e *M2* para a *Análise de Interfaces*. Nesta figura o conjunto

T é representado pelos elementos $t_{(}t1_{a}^{M1},t1_{a}^{M2})$ (apelidado de *a[1]*) e $t_{(}t2_{a}^{M1},t1_{a}^{M2})$ (apelidado de *a[2]*).

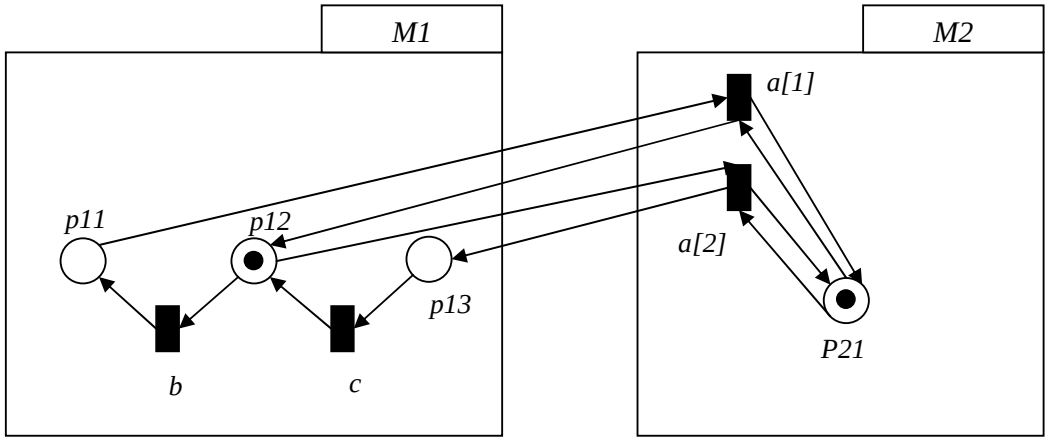


Figura 47 – *Análise de Interfaces* para o caso onde há repetições de ações nos módulos.

Mais uma vez, refinamentos podem ser realizados no modelo através desta análise, bem como o uso de técnicas de análise de RdP sobre as redes geradas. O projetista pode também habilitar e desabilitar os *Gates* para simular o comportamento dos módulos em relação aos módulos não incluídos.

3.3.3.2. Alguns Benefícios

Acredita-se que o fato do *Ambiente* trabalhar com diversos contextos contribui para o projetista visualizar o seu modelo através de diversos pontos de vistas e ferramentas, o que auxilia na formação do modelo e no estudo do problema de planejamento. O uso das Redes de Petri traz um ponto de vista interessante aos aspectos dinâmicos já que este formalismo foi criado justamente para modelar e estudar características dinâmicas.

Como visto, com a análise proposta o projetista pode refinar o seu modelo através de sucessivas verificações. Verificações que podem ser realizadas através dos módulos ou uma

combinação destes. Quando o domínio é realmente complexo e este possui proporções maiores, as análises tanto *Modulares* quanto de *Interfaces* se tornam boas alternativas durante um processo de análise do domínio como um todo.

Acredita-se também que a *Análise do Modelo do Domínio* pode trazer, além do refinamento do modelo de planejamento através de uma visão geral do domínio, informações adicionais relevantes (extraídas do modelo automaticamente ou não) para os algoritmos de planejamento. Essas informações dinâmicas podem contribuir para o processo de planejamento automático, no que diz respeito ao processo de busca por soluções, restringindo, por exemplo, espaços de busca que levam a soluções incorretas. Eventualmente essas informações podem ser expressas na forma de restrições tanto do domínio quanto do problema.

3.3.4. Testes com Planejadores

O *Ambiente* proposto visa também contemplar testes dos modelos elaborados com planejadores. Geralmente os planejadores utilizados para testes são aqueles disponíveis na literatura (muitos deles apresentados no Capítulo 2).

Os testes com planejadores têm por objetivo principal verificar possíveis planos que solucionam os problemas $\mathscr{P}$ de um domínio $\mathscr{D}$ e, conseqüentemente, realizar possíveis refinamentos no modelo como, por exemplo, adição de restrições, tanto no domínio quanto no problema. Eventualmente, a análise de planos gerados por um planejador pode indicar a necessidade de restrições no modelo que evitem, por exemplo, estados indesejáveis que só foram vistos durante os testes ou pode apresentar a falta ou o excesso de recursos no problema.

Conforme visto no Capítulo 2, uma das linguagens mais utilizadas no processo de planejamento automático através de planejadores é a PDDL. Assim, o *Ambiente* proposto

neste trabalho deve também permitir que o projetista realize testes com planejadores através da PDDL (ou eventualmente outra linguagem pertinente). De fato, o *Ambiente* deve prover mecanismos de passagem de uma representação a outra de forma transparente ao projetista.

Para o exemplo do domínio *Logística,* uma representação simplificada em PDDL do modelo do domínio gerado em UML.P (mecanismo de tradução que será proposto e apresentado no próximo capítulo), apenas para ilustração dos testes, seria a seguinte:

```
(define (domain Logistica)
    (:requirements :typing :fluents :negative-preconditions)
    (:types
      Veiculo - object
      Caminhao - Veiculo
      Aviao - Veiculo
      Lugar - object
      Local - Lugar
      Aeroporto - Lugar
      Pacote - object
      Cidade - object
    )

    (:predicates
      (estaDentro ?pac - Pacote ?vei - Veiculo)
      (estaEm ?pac - Pacote ?lug - Lugar)
      (pertence ?lug - Lugar ?cid - Cidade)
      (estaNo ?cam - Caminhao ?lug - Lugar)
      (estaParadoNo ?avi - Aviao ?aer - Aeroporto)
    )

    (:functions
      (capacidade ?vei - Veiculo)
      (carregamentoCorrente ?vei - Veiculo)
    )


    (:action dirigir
     :parameters (?cam - Caminhao ?de - Lugar ?para - Lugar ?cit - Cidade)
     :precondition
       (and
         (estaNo ?cam ?de)
         (pertence ?de ?cit)
         (pertence ?para ?cit)
       )
     :effect
       (and
         (not (estaNo ?cam ?de))
         (estaNo ?cam ?para)
       )
    )

    (:action carregaCaminhao
     :parameters (?cam - Caminhao ?pkg - Pacote ?loc - Lugar)
     :precondition
       (and
         (estaNo ?cam ?loc)
         (> (capacidade ?cam) (carregamentoCorrente ?cam))
         (estaEm ?pkg ?loc)
       )
     :effect
       (and
         (increase (carregamentoCorrente ?cam) 1)
         (not (estaEm ?pkg ?loc))
         (estaDentro ?pkg ?cam)
       )
    )
```

```
    (:action descarregaCaminhao
     :parameters (?cam - Caminhao ?pkg - Pacote ?loc - Lugar)
     :precondition
       (and
         (estaNo ?cam ?loc)
         (estaDentro ?pkg ?cam)
       )
     :effect
       (and
         (decrease (carregamentoCorrente ?cam) 1)
         (estaEm ?pkg ?loc)
         (not (estaDentro ?pkg ?cam))
       )
    )

    (:action pilotar
     :parameters (?avi - Aviao ?de - Aeroporto ?para - Aeroporto)
     :precondition
       (and
         (estaParadoNo ?avi ?de)
       )
     :effect
       (and
         (not (estaParadoNo ?avi ?de))
         (estaParadoNo ?avi ?para)
       )
    )

    (:action carregaAviao
     :parameters (?avi - Aviao ?pkg - Pacote ?loc - Aeroporto)
     :precondition
       (and
         (estaParadoNo ?avi ?loc)
         (> (capacidade ?avi) (carregamentoCorrente ?avi))
         (estaEm ?pkg ?loc)
       )
     :effect
       (and
         (increase (carregamentoCorrente ?avi) 1)
         (not (estaEm ?pkg ?loc))
         (estaDentro ?pkg ?avi)
       )
    )

    (:action descarregaAviao
     :parameters (?avi - Aviao ?pkg - Pacote ?loc - Aeroporto)
     :precondition
       (and
         (estaParadoNo ?avi ?loc)
         (estaDentro ?pkg ?avi)
       )
     :effect
       (and
         (decrease (carregamentoCorrente ?avi) 1)
         (estaEm ?pkg ?loc)
         (not (estaDentro ?pkg ?avi))
       )
    )
)
```

A representação do problema descrito anteriormente na modelagem do problema seria em PDDL da seguinte forma:

```
(define (problem Logistica_4_pacotes)
    (:domain Logistica)
    (:objects
      caminhao1 - Caminhao
      caminhao2 - Caminhao
      localSP1 - Local
      aviao1 - Aviao
      aviao2 - Aviao
      localRJ1 - Local
      aeroportoSP - Aeroporto
      aeroportoRJ1 - Aeroporto
      saopaulo - Cidade
      riodejaneiro - Cidade
      pkg1 - Pacote
      pkg2 - Pacote
      localSP2 - Local
      localRJ2 - Local
      pkg3 - Pacote
      pkg4 - Pacote
    )

    (:init
      (pertence aeroportoSP saopaulo)
      (pertence aeroportoRJ1 riodejaneiro)
      (pertence localSP1 saopaulo)
      (pertence localRJ1 riodejaneiro)
      (pertence localSP2 saopaulo)
      (pertence localRJ2 riodejaneiro)
      (estaNo caminhao1 localSP1)
      (estaEm pkg1 localSP2)
      (estaEm pkg2 localSP2)
      (estaParadoNo aviao1 aeroportoSP)
      (estaParadoNo aviao2 aeroportoRJ1)
      (estaEm pkg3 localRJ2)
      (estaEm pkg4 aeroportoRJ1)
      (estaNo caminhao2 localRJ1)
      (= (capacidade caminhao1) 4)
      (= (carregamentoCorrente caminhao1) 0)
      (= (capacidade caminhao2) 2)
      (= (carregamentoCorrente caminhao2) 0)
      (= (capacidade aviao1) 10)
      (= (carregamentoCorrente aviao1) 0)
      (= (capacidade aviao2) 10)
      (= (carregamentoCorrente aviao2) 0)
    )

    (:goal
      (and
        (estaNo caminhao1 localSP1)
        (estaParadoNo aviao1 aeroportoSP)
        (estaParadoNo aviao2 aeroportoRJ1)
        (estaEm pkg3 localSP1)
        (estaEm pkg4 localSP1)
        (estaEm pkg1 localRJ1)
        (estaEm pkg2 localRJ1)
        (estaNo caminhao2 localRJ1)
      )
    )
)
```

Tendo o modelo representado em PDDL é possível selecionar um planejador, como o Metric-FF (HOFFMANN, 2003), para que este resolva o problema de planejamento e gere um plano-solução a ser analisado. No caso do domínio e do problema representados acima em PDDL o planejador Metric-FF fornece o seguinte plano:

```
 0: CARREGAAVIAO AVIAO2 PKG4 AEROPORTORJ1
 1: PILOTAR AVIAO2 AEROPORTORJ1 AEROPORTOSP
 2: DESCARREGAAVIAO AVIAO2 PKG4 AEROPORTOSP
 3: DIRIGIR CAMINHAO2 LOCALRJ1 LOCALRJ2 RIODEJANEIRO
 4: CARREGACAMINHAO CAMINHAO2 PKG3 LOCALRJ2
 5: DIRIGIR CAMINHAO2 LOCALRJ2 AEROPORTORJ1 RIODEJANEIRO
 6: DESCARREGACAMINHAO CAMINHAO2 PKG3 AEROPORTORJ1
 7: DIRIGIR CAMINHAO1 LOCALSP1 LOCALSP2 SAOPAULO
 8: CARREGACAMINHAO CAMINHAO1 PKG1 LOCALSP2
 9: CARREGACAMINHAO CAMINHAO1 PKG2 LOCALSP2
10: DIRIGIR CAMINHAO1 LOCALSP2 AEROPORTOSP SAOPAULO
11: DESCARREGACAMINHAO CAMINHAO1 PKG1 AEROPORTOSP
12: DESCARREGACAMINHAO CAMINHAO1 PKG2 AEROPORTOSP
13: CARREGAAVIAO AVIAO2 PKG1 AEROPORTOSP
14: CARREGAAVIAO AVIAO2 PKG2 AEROPORTOSP
15: CARREGACAMINHAO CAMINHAO1 PKG4 AEROPORTOSP
16: DIRIGIR CAMINHAO1 AEROPORTOSP LOCALSP1 SAOPAULO
17: DESCARREGACAMINHAO CAMINHAO1 PKG4 LOCALSP1
18: PILOTAR AVIAO2 AEROPORTOSP AEROPORTORJ1
19: CARREGAAVIAO AVIAO2 PKG3 AEROPORTORJ1
20: DESCARREGAAVIAO AVIAO2 PKG1 AEROPORTORJ1
21: DESCARREGAAVIAO AVIAO2 PKG2 AEROPORTORJ1
22: CARREGACAMINHAO CAMINHAO2 PKG1 AEROPORTORJ1
23: CARREGACAMINHAO CAMINHAO2 PKG2 AEROPORTORJ1
24: DIRIGIR CAMINHAO2 AEROPORTORJ1 LOCALRJ1 RIODEJANEIRO
25: DESCARREGACAMINHAO CAMINHAO2 PKG1 LOCALRJ1
26: DESCARREGACAMINHAO CAMINHAO2 PKG2 LOCALRJ1
27: PILOTAR AVIAO2 AEROPORTORJ1 AEROPORTOSP
28: DESCARREGAAVIAO AVIAO2 PKG3 AEROPORTOSP
29: PILOTAR AVIAO2 AEROPORTOSP AEROPORTORJ1
30: DIRIGIR CAMINHAO1 LOCALSP1 AEROPORTOSP SAOPAULO
31: CARREGACAMINHAO CAMINHAO1 PKG3 AEROPORTOSP
32: DIRIGIR CAMINHAO1 AEROPORTOSP LOCALSP1 SAOPAULO
33: DESCARREGACAMINHAO CAMINHAO1 PKG3 LOCALSP1
```

O plano gerado por um planejador geral nas fases de testes pode auxiliar o projetista, por exemplo, a avaliar como o problema pode ser resolvido (no caso, não considerando tempo de resposta do planejador), avaliar também como os recursos e agentes podem estar sendo utilizados bem como verificar o modelo e seu comportamento. Por exemplo, no plano gerado pelo planejador Metric-FF, o agente *aviao1* não é utilizado para resolver o problema, isto, dependendo do ponto de vista, pode indicar uma má utilização dos recursos disponíveis ou eventualmente um excesso de recursos. Esta avaliação depende muito do interesse do projetista.

Mesmo o plano acima ilustrado sendo simples (devido o problema ser simples), no caso com 33 ações seqüenciais, tem-se uma boa visão do que o planejador pode fornecer durante um processo de testes com planejadores.

Vale lembrar que um mesmo problema pode ser resolvido de maneiras diferentes dependendo da técnica de planejamento (planejador) sendo utilizada durantes os testes. Assim é recomendado que o projetista utilize diferentes técnicas de planejamento para testar, refinar e novamente verificar seu modelo.

3.3.5. Escolha da Técnica de Planejamento

A escolha do planejador adequado para resolver os problemas de planejamento depende tanto das características do planejador quanto do domínio e do problema a ser resolvido. Características do domínio como, ações com tempo, restrições numéricas, sistema observável ou parcialmente observável, existência de eventos exógenos, funções de probabilidade, são parâmetros fundamentais para a tomada de decisão sobre a escolha do planejador mais adequado para resolver o problema de planejamento. Hoje existem diversos planejadores disponíveis, muitos destes não são capazes de resolver problemas mais complexos devido a suas limitações, outros possuem grandes potenciais na solução de problemas. Vale lembrar que existem planejadores dedicados a certos problemas, ou seja, muitas das informações e restrições do domínio já fazem parte do algoritmo, fazendo com que este evite buscas exaustivas sem sucesso. Como visto, a área de pesquisa de Planejamento Automático focaliza esforços em planejadores gerais, isto é, independente do domínio. Este fato faz com que o modelo deva possuir todas as informações necessárias para que o algoritmo de planejamento não forneça resultados insatisfatórios em tempos de resposta inviáveis. De fato, planejadores gerais são mais recomendados durante os testes do modelo.

Hoje não existem mecanismos para se avaliar qual o planejador é mais adequado para um determinado domínio. Esse assunto vem sendo bastante discutido na comunidade de Planejamento Automático, onde existe um grande interesse em se mapear características comuns de domínios distintos com os respectivos planejadores que possuem ótimas performances diante destas características (MCCLUSKEY et al., 2003) (VAQUERO; TONIDANDEL; SILVA, 2005).

Dentro do ciclo de vida de projeto de planejamento, e conseqüentemente dentro do *Ambiente*, a escolha de um planejador disponível seria a primeira opção como ponto de partida para o desenvolvimento de um planejador dedicado. Caso não haja planejadores que contribuam no desenvolvimento do domínio de planejamento em questão é recomendado o desenvolvimento de planejadores dedicados desenvolvidos desde de o começo, e eventualmente, aproveitando-se os avanços já alcançados nesta área de pesquisa.

3.4. Ferramentas de Suporte a Engenharia do Conhecimento

Ferramentas de suporte ao projetista durante a modelagem do domínio de planejamento eram extremamente escassas antes da primeira Competição Internacional de Engenharia do Conhecimento para Planejamento e Escalonamento (ICKEPS) em 2005. Antes do ICKEPS 2005 apenas alguns trabalhos eram direcionados a elaboração e desenvolvimento dessas ferramentas. Um deles, o principal nesta área, foi o trabalho do pesquisador McCluskey et at. (2003) onde conceitos e sugestões de processos de aquisição e modelagem de domínios de planejamento eram propostos através de um ambiente de planejamento idealizado. De fato, este trabalho vem guiando a área da EC para Planejamento. Como fruto dessa linha de pesquisa, McCluskey juntamente com Simpson idealizaram plataformas e ferramentas que contribuíssem nos processos de modelagem, verificação e validação de modelos. A

ferramenta pioneira, desenvolvida dentro deste ideal, se chamou GIPO (*Graphical Interface for Planning with Objects*) (SIMPSON et al., 2001).

A ferramenta GIPO é uma das únicas com foco direcionado aos processos de aquisição de conhecimento e modelagem do domínio. Esta também possui uma abordagem orientada a objetos durante a modelagem o que vem mostrando ser uma abordagem interessante no contexto de planejamento. GIPO se tornou uma ferramenta de referência nesta área, mas a mesma possuía algumas restrições tais como: a utilização de uma linguagem não tão conhecida fora do âmbito acadêmico, a OCL (*Object Centered Language* (SIMPSON et al., 2001)); projetistas sem conhecimento em planejamento automático ou conceitos de IA podem ter dificuldades no uso da mesma; a ferramenta não contempla as fases de requisitos no ciclo de vida de projeto.

Ferramentas como "*Common Process Editor*" (TATE; POLYAK; JARVIS, 1998) e "*Act Editor*" (MYERS; WILKINS, 1997) também se destacaram antes da competição, mas estas eram ferramentas específicas a alguns domínios. Muitas ferramentas (a maioria) disponíveis para a aquisição e validação de modelos são consideradas apenas verificadores de sintaxe ou removedores de inconsistência (*Debugging*) (MCCLUSKEY, 2002).

De fato, para lidar com problemas reais de planejamento são realmente necessários ambientes, ferramentas e métodos que auxiliem o projetista e toda sua equipe a desenvolver um ciclo de vida de projeto com ótimos resultados. As ferramentas podem contribuir para a síntese de modelos, análises automáticas de domínios, correções de sintaxe, remover redundâncias, visualização do modelo, verificação, validação entre outras contribuições. Essas contribuições são similares àquelas encontradas em ferramentas de apoio à modelagem e desenvolvimento de softwares ou de sistemas de engenharia em geral.

Diante deste cenário o desenvolvimento de uma ferramenta que considera o *Ambiente* de modelagem e análise de domínios de planejamento proposto, bem como o ciclo de vida de

projeto, se fez necessário. Como resultado do presente trabalho de pesquisa, foi desenvolvido um protótipo de ferramenta que contempla os conceitos apresentados, principalmente os conceitos do *Ambiente* com o processos de análise de requisitos, modelagem, verificação e análise de domínio. A ferramenta desenvolvida foi chamada de itSIMPLE (*Integrated Tools Software Interface for Modeling Planning Environments*) (VAQUERO; TONIDANDEL; SILVA, 2005) e esta participou da competição ICKEPS 2005, juntamente com outras sete ferramentas, onde obteve ótimos resultados. O capítulo a seguir descreve esta ferramenta, fruto do presente trabalho de pesquisa, que implementa o *Ambiente* proposto neste capítulo visando auxiliar projetistas nos processos de design de domínios de planejamento.

Capítulo 4

4. Ferramenta de Suporte a Modelagem e Análise de Domínios de Planejamento

No capítulo anterior foram discutidos os conceitos referentes ao *Ambiente* integrado proposto, que contempla os aspectos do ciclo de vida de um projeto, com foco principalmente nos processos de análise de requisitos, na modelagem e análise de domínios de planejamento. Uma vez apresentados esses processos e estudadas as ferramentas disponíveis com fins similares no processo de design, o presente projeto de pesquisa concentrou atenção no desenvolvimento de uma ferramenta que concebesse o *Ambiente* e que fornecesse suporte, principalmente, à modelagem e análise de domínios de planejamento tanto simples quanto reais para o projetista e toda sua equipe.

Neste capítulo são apresentadas as características e funcionalidades da ferramenta protótipo desenvolvida neste projeto que foi nomeada de itSIMPLE (*Integrated Tools Software Interface for Modeling PLanning Environments*). Vale citar que esta ferramenta já vem sendo bem aceita pela comunidade de pesquisa de Planejamento Automático, haja visto a participação nos principais congressos da área como, por exemplo, ICAPS 2005 e 2006 (VAQUERO; TONIDANDEL; SILVA, 2005) (VAQUERO et al., 2006) e principalmente a participação na competição na área de Engenharia do Conhecimento aplicada ao Planejamento Automático (ICKEPS) ocorrida durante o ICAPS 2005 (VAQUERO; TONIDANDEL; SILVA, 2005) onde ganhou reconhecimento.

O capítulo inicia com uma breve descrição da ferramenta itSIMPLE com seus principais objetivos. Em seguida é apresentada a arquitetura da ferramenta em relação às linguagens utilizadas. O ambiente de análise de requisitos, modelagem e análise da ferramenta é apresentado seguido dos mecanismos de traduções dos modelos tanto para a linguagem PDDL quanto para a linguagem PNML (no caso para as visualizações em Redes de Petri). Vale lembrar que a tradução para PDDL no *Ambiente* é feita com o objetivo de disponibilizar o modelo gerado na ferramenta para que os planejadores gerais possam entender o mesmo, já que a PDDL é atualmente o idioma dos planejadores, para os eventuais testes com os planejadores.

4.1. O itSIMPLE

O itSIMPLE - *Integrated Tools Software Interface for Modeling PLanning Environments* - é um projeto de um ambiente integrado que possui como principal objetivo minimizar os problemas encontrados durante o ciclo de vida de projeto de aplicações reais de planejamento, principalmente nas fases de requisitos, modelagem e análise, onde os diferentes pontos de vista dos participantes devem ser levados em consideração. O software itSIMPLE, desenvolvido em Java™ (GOSLING et al., 2000), foi projetado para que o usuário possa realizar processos disciplinados das fases iniciais de projetos com o objetivo de criar modelos, ricos em conhecimento, de diversos domínios de planejamento reais. A ferramenta desenvolvida é independente de qualquer técnica ou algoritmo de planejamento existente, isto é, os processos de modelagem e verificação não dependem de uma técnica específica de planejamento (planejador). De fato, os algoritmos de planejamento são vistos como componentes presentes neste ambiente que podem ser escolhidos (automaticamente ou não) livremente.

Através do itSIMPLE o projetista pode especificar, modelar e analisar domínios utilizando linguagens conhecidas sem a necessidade de profundos conhecimentos na área de Planejamento Automático. Linguagens e formalismos como, inicialmente, a UML, XML, Redes de Petri e PDDL são integradas em uma única ferramenta (*Ambiente*) de modo a proporcionar ao usuário uma grande flexibilidade e facilidade na mudança de contexto (modelagem, análise, entre outros), possibilitando o uso do potencial de cada linguagem e enriquecendo assim os modelos e as respectivas análises. Linguagens visuais como a UML e Redes de Petri propiciam uma maior uniformidade nos conceitos do modelo perante os participantes (*stakeholders*, especialistas em planejamento, especialistas do domínio, usuários, etc), com diferentes pontos de vista, devidos aos diagramas e as animações.

O itSIMPLE foi projetado também para facilitar a reutilização e a manutenção dos modelos já desenvolvidos. Com a ferramenta, o projetista pode utilizar partes de domínios já modelados durante a elaboração de projeto, pois ela permite o projetista visualizar e trabalhar com diversos domínios ao mesmo tempo como será visto nos próximos tópicos. A manutenção dos modelos faz com que o conhecimento declarado nos modelos seja enriquecido gradativamente através da ferramenta e eventualmente reutilizado.

4.2. A Arquitetura do itSIMPLE

Visando a facilidade e o poder de representação da ferramenta, a arquitetura do itSIMPLE foi desenvolvida para incorporar um conjunto de linguagens e formalismos capazes de lidar com os requisitos e com os processos provenientes da Engenharia do Conhecimento para planejamento.

Entre as diversas linguagens de análise e modelagem disponíveis, o projeto itSIMPLE utiliza, como forma de representação inicial do conhecimento, a linguagem semi-formal UML (OMG, 2001), que é uma linguagem diagramática bem conhecida, comumente utilizada nas

áreas das Engenharias de Requisitos e de Software. Como visto, a UML é muito utilizada na especificação e modelagem em uma grande variedade de aplicações. Acredita-se, neste trabalho, que muitos engenheiros são, de alguma forma, familiarizados com esta linguagem. De fato, a UML é apropriada na elaboração dos primeiros modelos do domínio de planejamento.

Devido ao fato dos aspectos dinâmicos serem fundamentais em domínios de planejamento, linguagens e formalismos que provêem representações e mecanismos de análise para tais aspectos podem realmente contribuir para uma melhor e mais correta modelagem e análise. A arquitetura elaborada utiliza as Redes de Petri (MURATA, 1989) para a análise dinâmica onde são realizadas verificações do modelo do domínio para seu refinamento, conforme visto no Capítulo 3. De fato, as Redes de Petri também são muito utilizadas, principalmente na indústria, na modelagem de processos de manufatura (MURATA, 1989).

Como a comunidade de Planejamento Automático padronizou, através da PDDL (MCDERMOTT, 1998), a linguagem de definição de modelos do domínio de planejamento para servir como dados de entrada para os planejadores, a ferramenta itSIMPLE integra mecanismos para lidar com esta linguagem, principalmente para eventuais teste dos modelos com planejadores gerais.

Com o objetivo de armazenar e estruturar toda informação e conhecimento disponibilizado pelo projetista através de todas as linguagens utilizadas (UML, RdP e PDDL), a ferramenta itSIMPLE utiliza e linguagem XML (BRAY et al., 2004) que é vastamente utilizada, por exemplo, em sistemas de transição de dados, aplicações de Internet e sistemas de troca de informação. Esta linguagem é de fácil manipulação, onde qualquer *browser* de Internet ou interpretador de XML pode ser utilizado para acessar os dados do modelo do domínio. Outro ponto importante derivado da utilização da XML é que a UML, as RdP e a

PDDL possuem representações em XML como XMI (OMG, 2005) (esta não é utilizada neste trabalho), PNML (WEBER; KINDLER, 2002) e XPDDL (GOUGH, 2004), respectivamente, bem como muitas outras linguagens, o que contribui para a integração.

A representação dos modelos em XML, integrador no *Ambiente* em relação a linguagens, é chamada aqui de *Representação Core*, conforme mencionada no Capítulo 3. Seguindo o princípio de portabilidade, a *Representação Core* é utilizada para realizar a integração entre as representações da UML, RdP e PDDL através de funções de tradução que mapeiam o conhecimento de uma linguagem à outra (mapeamentos que serão apresentados neste capítulo). A tradução entre representações é realizada, não apenas por conveniência, mas também para que torne possível o uso das capacidades de outras linguagens e ferramentas com o objetivo de se realizar, por exemplo, verificações e testes.

De fato, todos os processos de modelagem, verificações e traduções entre linguagens são realizados sobre a estrutura em XML da *Representação Core*. No caso da UML, toda a informação representada nos diagramas é diretamente estruturada e armazenada na *Representação Core*. As análises com as RdP, tanto na análise de requisitos propostas em (SANTOS; SILVA, 2004) quanto na análise das características propostas neste trabalho, são possíveis através da PNML (e também a *PNML Modular*) inferida da *Representação Core*, ou seja, dados extraídos da *Representação Core* do modelo são representados em PNML possibilitando a visualização das redes. Já os testes com planejadores podem ser realizados com a PDDL, pois dados do modelo são extraídos da *Representação Core* e representados primeiramente em XPDDL e conseqüentemente em PDDL. Esta estrutura integrada permite que o projetista exporte representações de um mesmo modelo para que estas sejam utilizadas eventualmente em outras ferramentas com objetivos diversos como, por exemplo, análises mais específicas, simuladores, entre outros. Como visto no Capítulo 3, os processos de análise e testes podem providenciar refinamentos no modelo do domínio, e este é o objetivo principal

das traduções, principalmente para as RdP e para a PDDL. Considerando todos estes aspectos, a arquitetura integrada proposta do itSIMPLE é demonstrada na Figura 48.

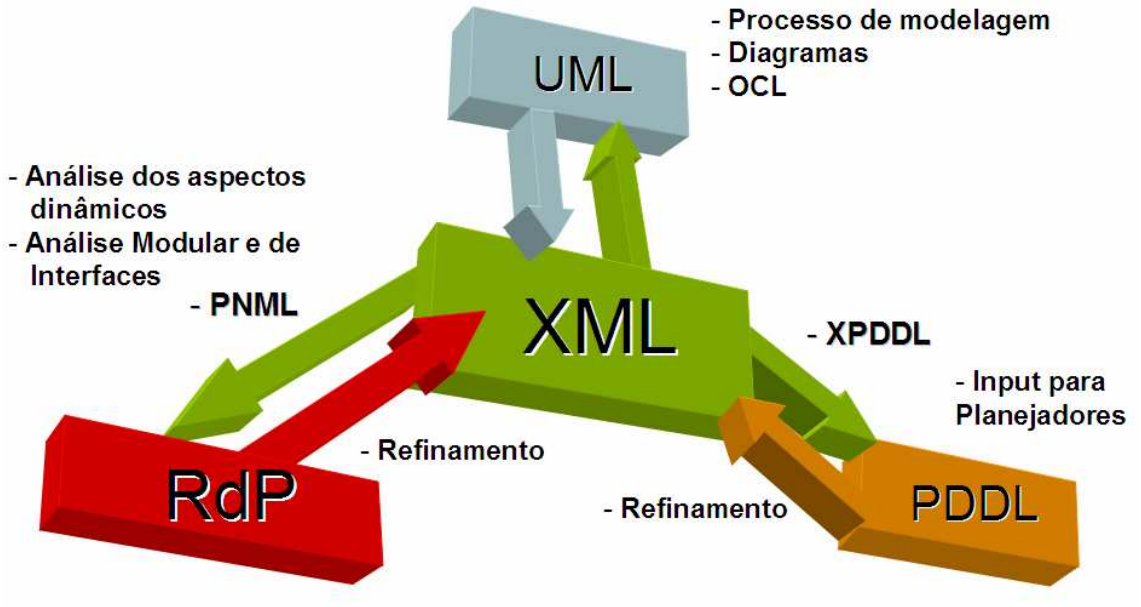


Figura 48 – A arquitetura das linguagens integradas utilizadas no itSIMPLE.

A *Representação Core* utilizada neste trabalho espelha os elementos da UML, isto é, a estruturação XML para armazenar o modelo reflete bastante a estrutura dos diagramas e dos elementos da linguagem UML. Baseada nesta estruturação é que os processo de tradução são realizados. A Figura 49 demonstra a estrutura de *tags* da XML (*Representação Core*), de forma simplificada, para um projeto (modelagem e análise de um domínio de planejamento) desenvolvido no itSIMPLE.

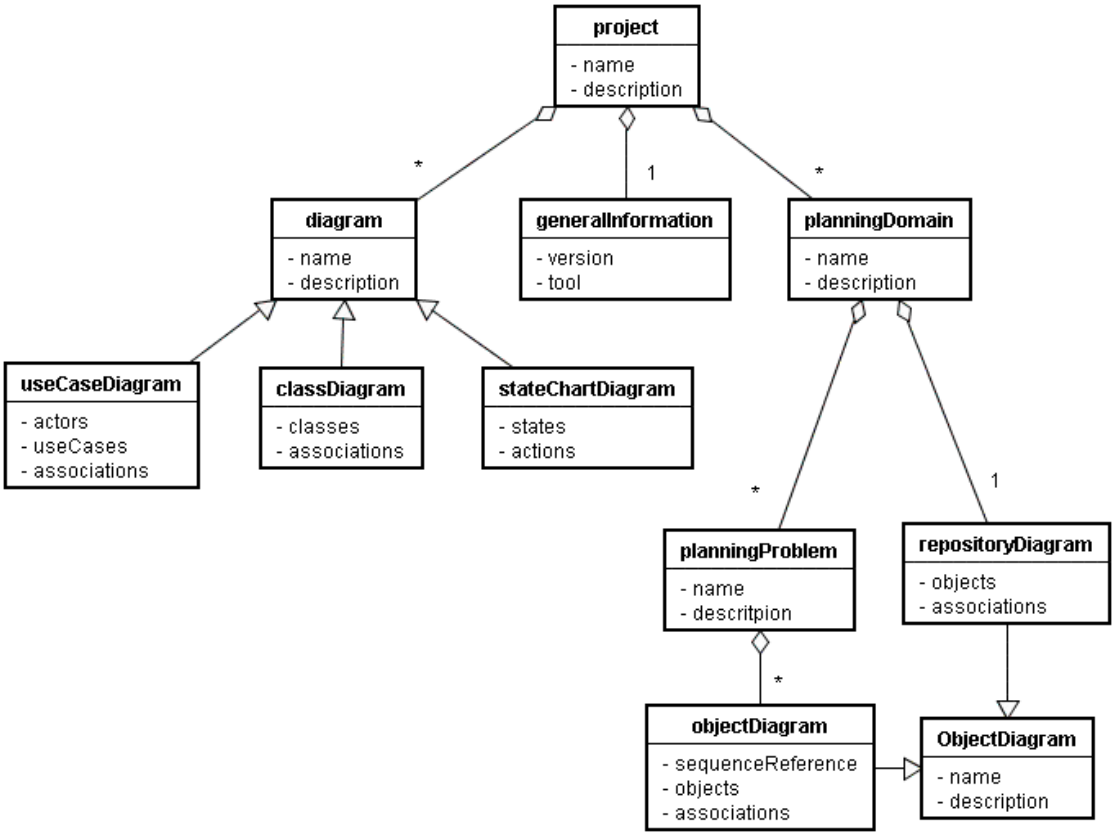


Figura 49 – Estrutura simplificada da *Representação Core* (XML) para um projeto no itSIMPLE.

Com esta estrutura da *Representação Core* de um projeto, o projetista pode modelar, para um mesmo conjunto de características abstratas do modelo do domínio (os conjuntos A,, *Ru*, *Ac* de um domínio $\mathcal{D}$ modelo representados aqui através dos diagramas de *Caso de Uso*, *Classes* e *Estados*), diversas variações do domínio (criando diversos conjuntos de *Ag*, *Rs* e incrementos em *Ru* utilizando os *Diagrama de Objetos* apelidado na estrutura de *repositoryDiagram ou Repositório*). Conseqüentemente, para cada variação do domínio criada, é possível criar um conjunto de problemas de planejamento (que são modelados também através dos *Diagramas de Objetos* e identificados, snapshot inicial ou final, através da propriedade *sequenceReference* conforme mostra estrutura da Figura 49). Para exemplificar esta variação do domínio, se um projetista estivesse modelando o domínio

Logística, este poderia modelar uma variação do domínio, no mesmo projeto, onde houvesse 20 pacotes, 10 caminhões, 5, aviões, 5 cidades, com diversos locais e aeroportos e conseqüentemente modelar problemas de planejamento com estes agentes e recursos. O projetista pode utilizar as mesmas características modeladas nos diagramas de *Casos de Uso*, *Classes* e *Estados* para também analisar esta situação com diferentes agentes e recursos, bem como novas restrições. A Figura 50 ilustra esta flexibilidade durante a elaboração de um projeto no itSIMPLE.

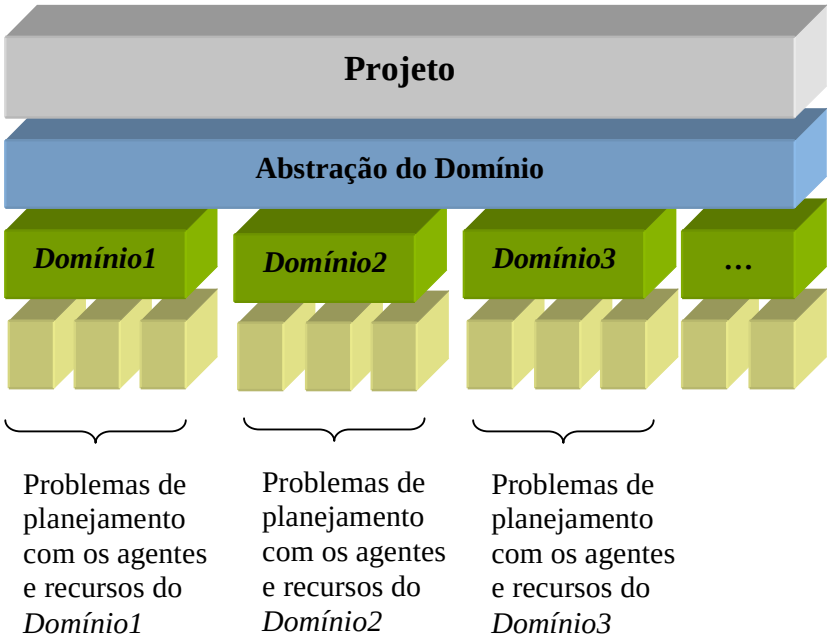


Figura 50 – Flexibilidade na construção de modelos no itSIMPLE.

O itSIMPLE provê ao projetista a interface necessária do *Ambiente* para que o mesmo possa realizar os processos modelagem e análise nas diversas linguagens presentes no trabalho. Esta interface é demonstrada durante os próximos tópicos.

4.3. O Ambiente de Modelagem e Análise do itSIMPLE

A ferramenta itSIMPLE provê uma interface gráfica (GUI) baseada principalmente nos diagramas UML. De fato, o projetista interage bastante com os diagramas da UML durante os processos de modelagem e análise. A interface da ferramenta itSIMPLE foi projetada para ser simples, agradável, com uma aparência similar às ferramentas CASES de modelagem em UML. A interface desenvolvida visa disponibilizar ao usuário mudanças de contextos sem dificuldades, ou seja, em apenas uma interface o projetista pode visualizar os diagramas UML, as redes de Petri, bem como os modelos em PDDL. Este fato faz com que o projetista não precise mudar de aplicação para realizar os principais processos de design de um domínio de planejamento, pois os processos estão interligados na ferramenta. A Figura 51 demonstra as principais características e elementos da interface do itSIMPLE.

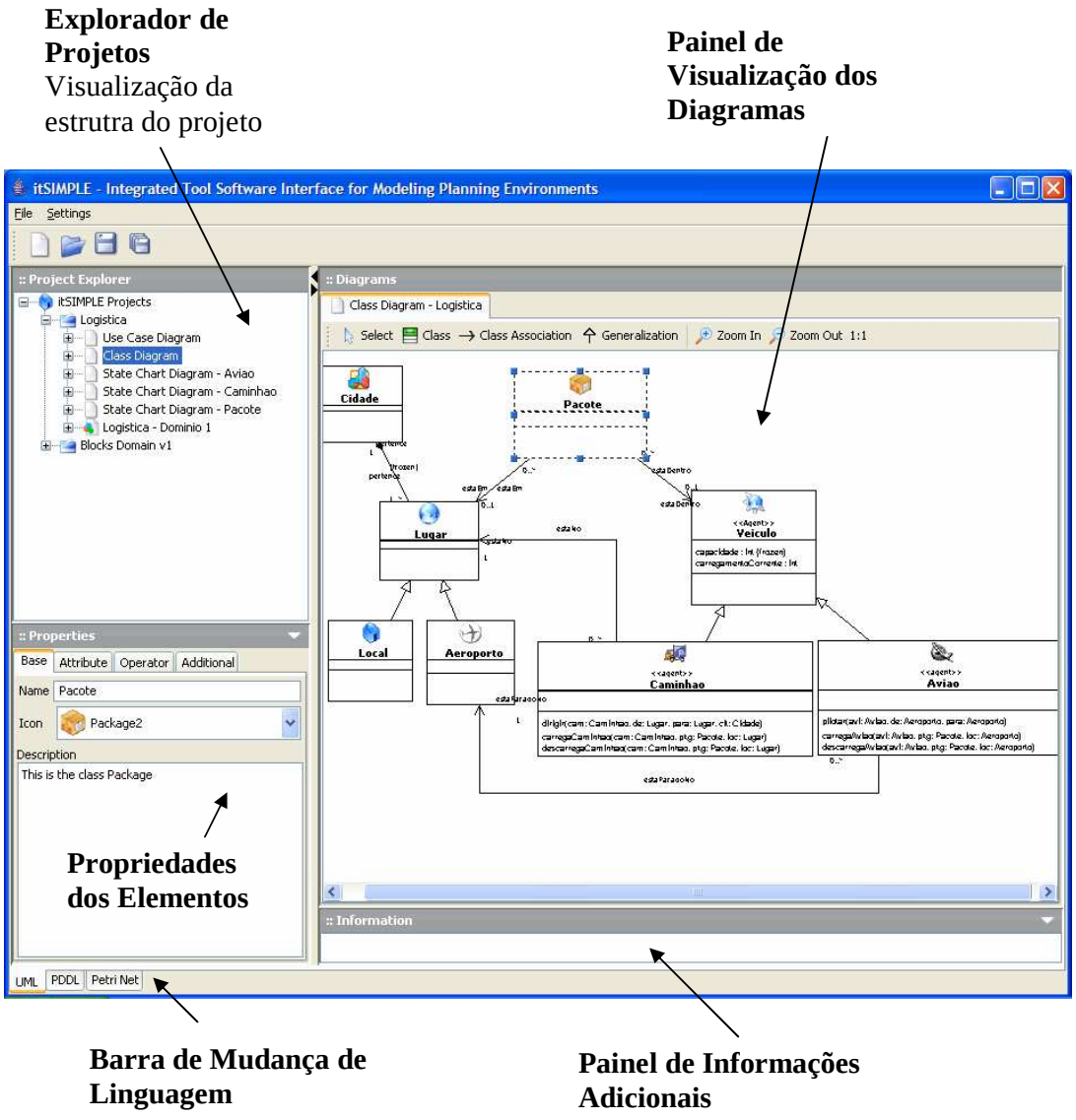


Figura 51 – Ambiente de Análise e Modelagem do itSIMPLE.

Nesta interface o projetista pode visualizar todos os projetos que deseja trabalhar simultaneamente e toda a estrutura de diagramas e seus respectivos elementos (agentes, casos de uso, classes, estados e objetos) de cada projeto através do painel "*Explorador de Projetos*". Esta área da interface fornece uma visão geral de cada projeto, e conseqüentemente de cada modelo, propiciando um acesso rápido aos elementos do mesmo. A Figura 52 demonstra

como o projetista visualiza seu projeto no painel "*Explorador de Projetos*" através do exemplo do domínio *Logística* apresentado no Capítulo 3.

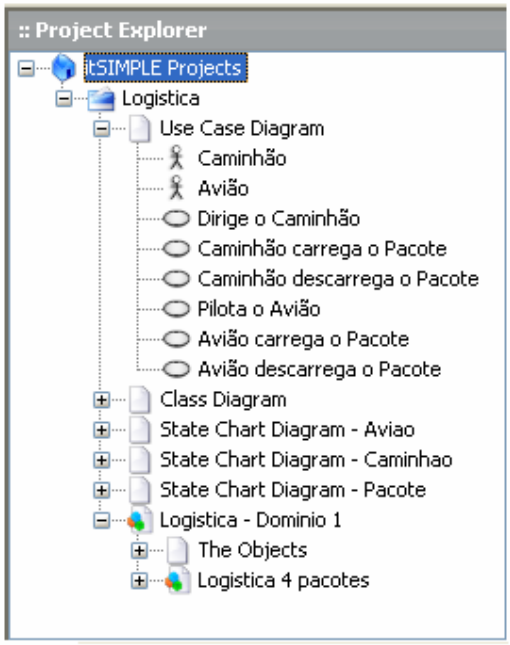


Figura 52 – O painel *Explorador de Projetos* com o projeto do domínio Logística no itSIMPLE.

No painel de "*Visualização dos Diagramas*" o projetista pode trabalhar com mais de um diagrama e nestes o usuário pode representar os elementos presentes em cada um dos diagramas (*Casos de Uso, Classes, Estados* e *Diagrama de Objetos*) da UML discutidos neste trabalho. Nesta área o projetista escolhe o elemento que deseja adicionar ao diagrama através da barra de elementos e adiciona-o na posição desejada. A área de "*Propriedade dos Elementos*" fornece ao usuário a flexibilidade de modificar as características do elemento selecionado, seja um ator, um caso de uso, uma classe, uma associação, um estado, uma ação ou um objeto. Em geral, características como nome do elemento, descrições informais, atributos e seus tipos, nome e parâmetros das ações, multiplicidades, pré e pós-condições em OCL são declaradas utilizando este painel da interface.

Informações adicionais relevantes sobre os elementos dos diagramas podem ser vistas no painel "*Informações adicionais*". Este painel mostra, por exemplo, a descrição completa de um caso de uso ou de um ator introduzida pelo projetista. Outro exemplo seria a descrição completa de uma ação do *Diagrama de Estados* quando esta for selecionada no diagrama.

Finalmente, utilizando a "*Barra de Mudança de Linguagem*" o projetista pode navegar entre as linguagens sem a necessidade de processos demorados de tradução. Nesta barra é possível mudar a visualização do modelo para a UML e PDDL no momento desejado, bem como a visualização das Redes de Petri para as eventuais análises.

Através desta interface, o projetista e toda sua equipe podem discutir os diferentes pontos de vista de um mesmo modelo através dos diagramas da UML.P e das linguagens e formalismos como PDDL e RdP tratadas na ferramenta. Os próximos itens apresentam as interfaces da ferramenta para a construção dos principais diagramas usados tanto para análise de requisitos quanto para a modelagem, bem como as interfaces para a análise do domínio e para tradução para PDDL.

4.3.1. Análise de Requisitos e Modelagem - Diagrama de Casos de Uso

Como apresentado no Capítulo 3, o processo de análise de requisitos e também o de especificação podem ser realizados utilizando o *Diagrama de Casos de Uso*, juntamente com as RdP geradas a partir de cada *Caso de Uso*, como apresentado em (SANTOS; SILVA, 2004). Como visto, este diagrama é também utilizado nos primeiros passos da modelagem de um domínio em níveis altos de abstração.

Na ferramenta, cada *Ator* do diagrama pode ser descrito textualmente, bem como cada *Caso de Uso*. Para cada *Caso de Uso* o projetista pode, de forma organizada, descrever um resumo do o caso de uso, suas pré e pós-condições, invariantes, questões importantes e principalmente o seu fluxo de eventos de forma estruturada conforme o trabalho de Santos e

Silva (2004). Esta representação do fluxo de eventos estruturada permite ao usuário visualizar uma Rede de Petri para a verificação e análise dos requisitos junto aos participantes. A Figura 53 mostra as interfaces para a construção deste diagrama e a representação em RdP.

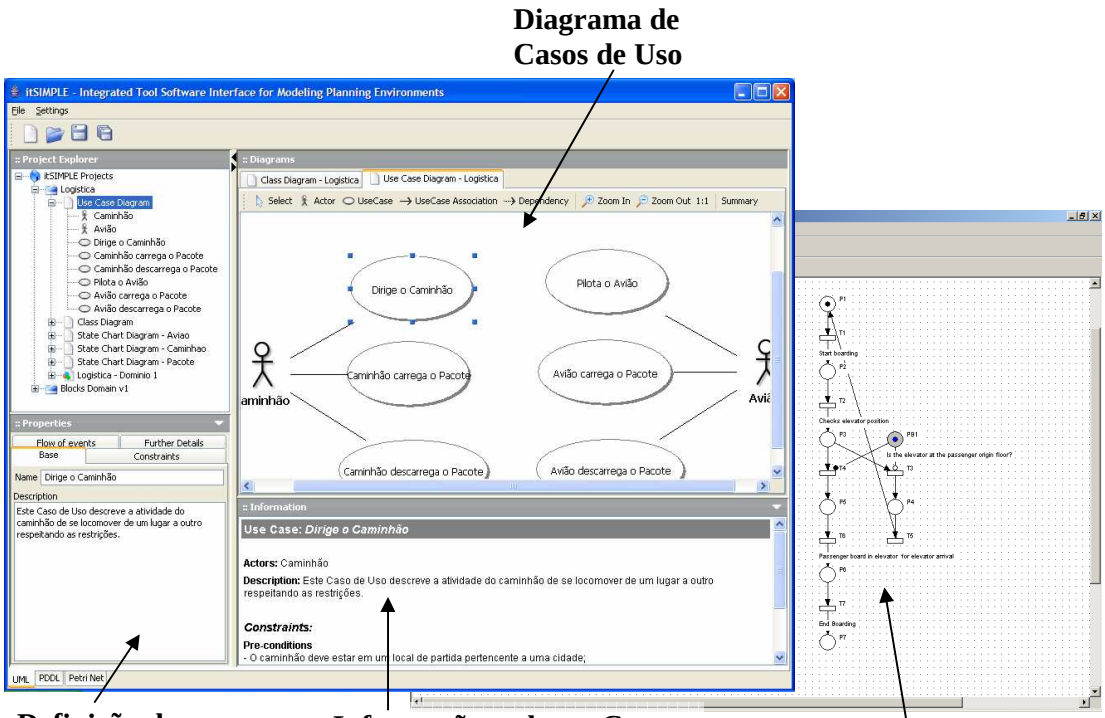


Figura 53 – Análise de Requisitos no itSIMPLE.

A Rede de Petri gerada a partir de um *Caso de Uso* segue rigorosamente as regras definidas no trabalho de Santos e Silva (2004) e esta rede é gerada em PNML. A Figura 54 demonstra como o fluxo de eventos é representado no itSIMPLE através do exemplo de *Caso de Uso* "*Dirige de um Lugar a outro*" do domínio *Logística*. A estruturação do fluxo de eventos dos Casos de Uso é realizada no painel "*Propriedades dos Elementos*".

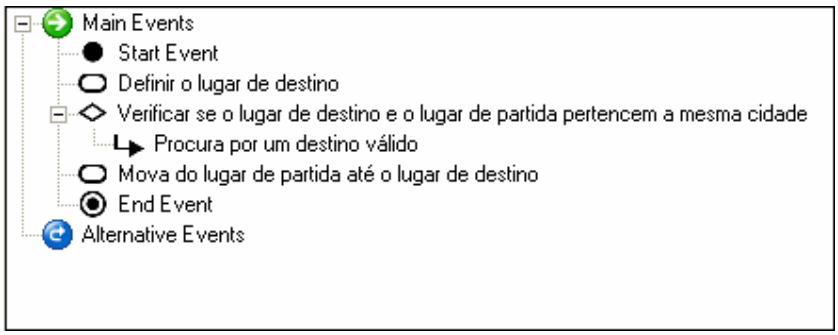


Figura 54 – Caso de Uso "*Dirige de um Lugar a outro*" do domínio *Logística* estruturado no itSIMPLE.

O itSIMPLE não fornece mecanismo de simulação das redes geradas, mas o projetista pode utilizar outras ferramentas específicas de RdP que provêem mecanismos de simulação de redes. Esta utilização de outras ferramentas só é permitida devido ao uso da PNML que contribui para este intercâmbio.

Vale citar que o projetista pode utilizar as descrições realizadas neste diagrama para compor o documento de especificação do domínio. A ferramenta fornece, no painel "*Informações Adicionais*" uma descrição textual organizada completa de cada elemento do diagrama e estas informações organizadas podem ser eventualmente utilizadas na documentação.

4.3.2. Modelagem da Estrutura Estática do Domínio - Diagrama de Classes

Na modelagem da estrutura estática do domínio através do *Diagrama de Classes* o projetista pode definir as principais classes, bem como seus atributos, associações, restrições e ações, a partir dos requisitos identificados. A interface para a construção deste é demonstrada na Figura 55.

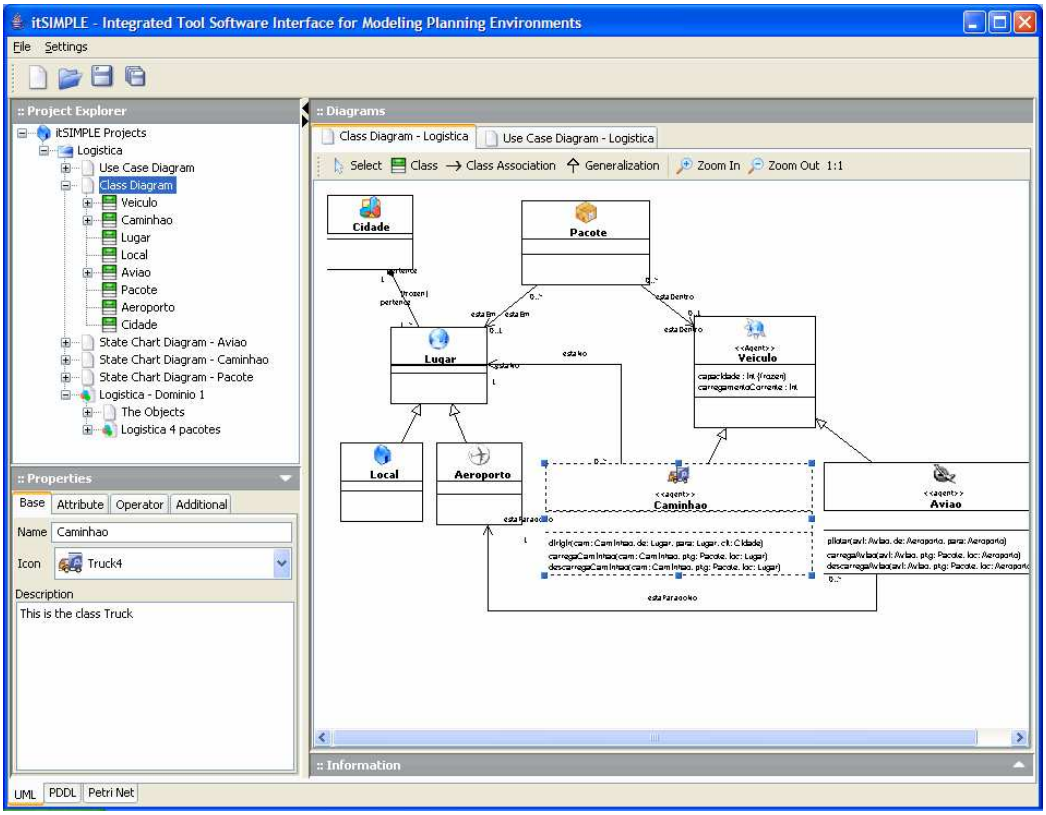


Figura 55 – Modelagem da Estrutura Estática do domínio no itSIMPLE.

Diferentemente da construção convencional dos *Diagramas de Classe*, no itSIMPLE o projetista pode utilizar recursos visuais que aproximam o modelo dos conceitos reais encontrados no domínio real. Para tal, o itSIMPLE permite ao usuário escolher uma imagem (ícone) associada a cada classe do diagrama. Esta imagem, escolhida adequadamente, traz, visualmente, o sentido da classe no mundo real favorecendo a uniformidade de conceitos do modelo perante todos os participantes.

Todas as propriedades dos elementos deste diagrama são definidas através do painel de "*Propriedades dos elementos*". Para uma classe, por exemplo, é possível definir sua imagem, seus atributos, suas ações (com seus respectivos parâmetros), bem como seu *stereotype*. Para uma associação é possível definir, por exemplo, o seu tipo (simples, agregação ou composição), os rolenames, as multiplicidades. Vale citar que no itSIMPLE o projetista pode

definir os tipos de cada atributo. Por exemplo, pode-se definir um atributo como sendo do tipo primitivo, ou seja, booleano (Boolean), inteiro (Int), float (Float), string (String) ou não-primitivo, ou seja, outra classe (por exemplo, um atributo da classe *Pacote* chamado "*fabricadoEm*" do tipo *Cidade – fabricadoEm : Cidade*) .

Seguindo o exemplo do domínio *Logística* apresentado no Capítulo 3, a Figura 56 ilustra o *Digrama de Classes*, construído no itSIMPLE, deste domínio. Uma breve comparação pode ser realizada entre os diagramas convencionais e os diagramas construídos no itSIMPLE através do diagrama na Figura 28 no Capítulo 3 e do diagrama da Figura 56.

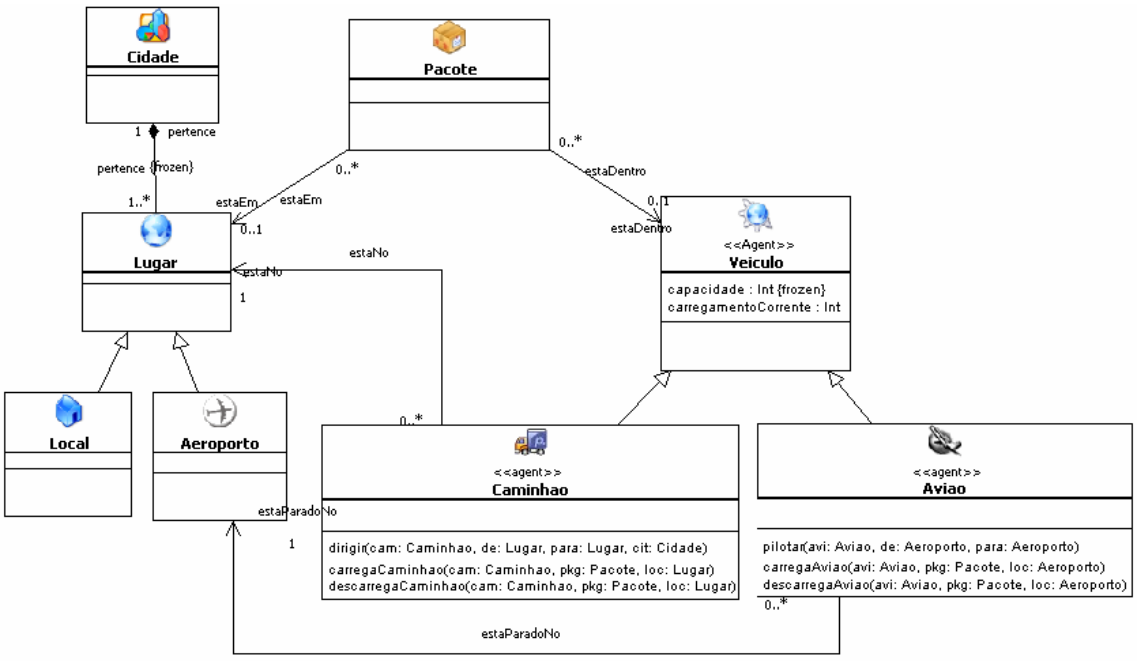


Figura 56 – Exemplo de estrutura estática do modelo do domínio *Logística* valorizado pelas imagens associadas às classes no itSIMPLE.

Para que a integração entre as linguagens presentes no *Ambiente* não seja prejudicada (principalmente a integração com a PDDL) algumas práticas são recomendadas, sendo elas:

- Utilizar nomes diferentes para rolenames, atributos e classes, ou seja, todos os elementos do diagrama devem possuir nomes distintos. Por exemplo, não utilizar o mesmo nome de um atributo em classes diferentes. Este ponto é essencial para garantir as traduções para a PDDL propostas neste trabalho;
- Definir sempre pelo menos um *rolename* para cada associação, pois o rolename é quem identifica como uma classe pode acessar a outra;
- Identificar as multiplicidades é fundamental para a modelagem dos problemas de planejamento, pois estas auxiliam na formação de estados coerentes;
- Tratar como variáveis globais do modelo os atributos das classes com *stereotype* definido como <<*utility*>>. Um exemplo seria um atributo chamado de "*consumoTotal*" de uma classe global (*stereotype* <<*utility*>>) para armazenar o consumo total de combustível do modelo.

4.3.3. Características Dinâmicas do Domínio - Diagrama de Estados

Para a modelagem das características dinâmicas do domínio, conforme apresentada e proposta no Capítulo 3, o itSIMPLE disponibiliza ao projetista a interface necessária para a construção do *Diagrama de Estados* da UML.P. No itSIMPLE, o projetista pode modelar os estados e as ações com suas respectivas pré e pós-condições. Vale lembrar que os estados são definidos de acordo com os valores atributos, bem como as pré e pós-condições, através de expressões em OCL. A Figura 57 mostra a interface para a construção dos *Diagramas de Estados* do modelo.

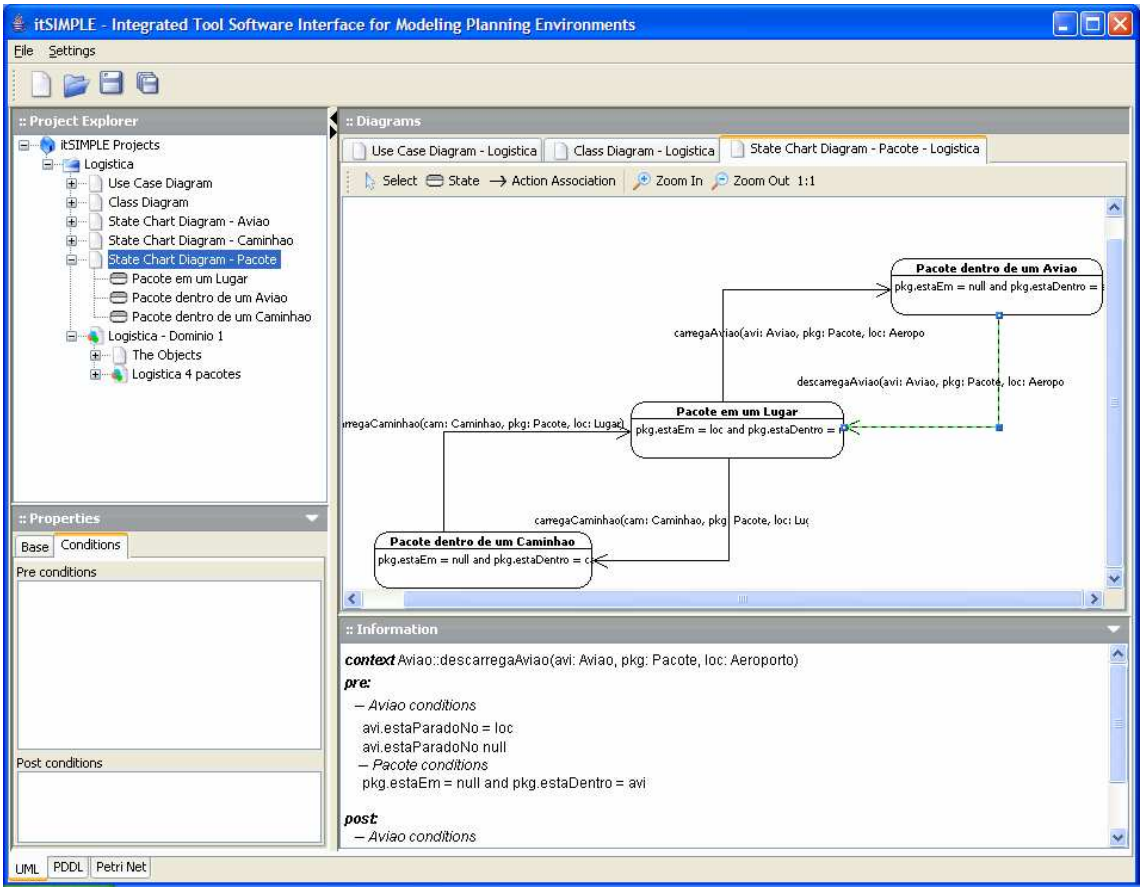


Figura 57 – Modelagem das características dinâmicas do modelo do domínio no itSIMPLE.

Para a representação tanto das pré e pós-condições das ações quanto dos estados em OCL, o itSIMPLE provê um editor de OCL para auxiliar o projetista durante a modelagem (este editor disponível não fornece verificações da sintaxe de expressões). A Figura 58 mostra como o projetista pode escrever expressões em OCL no painel "*Propriedades dos Elementos*", tanto para estados (a) quanto para ações (b). Esta figura demonstra exemplos retirados do domínio *Logística* modelado no itSIMPLE. Em (a), tem-se o estado "*Pacote em um Lugar*" e em (b) tem-se a ação *carregaCaminhao* visualizada no *Diagrama de Estados* da classe *Avião*, ou seja, sobre o ponto de vista do *Avião*.

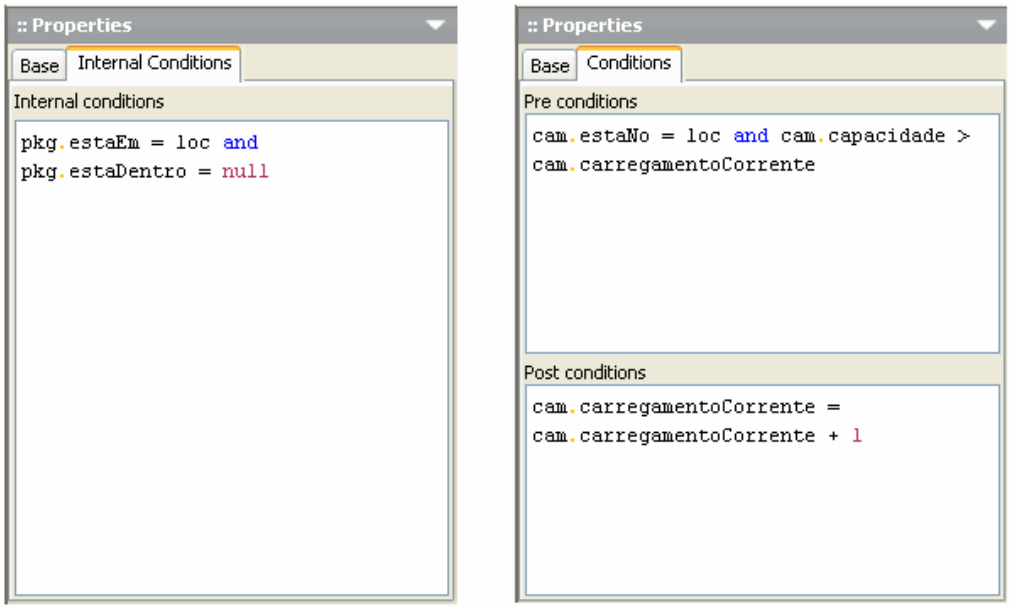


Figura 58 – Editor de auxilio a escrita de expressões OCL no itSIMPLE.

Como uma mesma ação pode estar presente em diversos *Diagramas de Estados* e a união dos diagramas fornece a visão completa de tal ação (ou seja, as pré e pós-condição de uma ação podem estar espalhadas em diversos diagramas conforme apresentado anteriormente), o itSIMPLE auxilia o projetista durante a modelagem das ações, para que este evite erros durante a construção dos diversos *Diagramas de Estados*, da seguinte maneira: durante a definição das pré e pós-condições de uma dada ação, o itSIMPLE reuni todas as pré e pós-condições desta mesma ação provenientes de outros diagramas para que o projetista saiba o que já foi declarado e o que não foi. Para exemplificar este auxílio a Figura 59 demonstra, no exemplo do domínio *Logística*, a representação da ação *carregaCaminhao* em OCL no painel "*Informações Adicionais*" quando o projetista está, por exemplo, trabalhando sobre o *Diagrama de Estados* da classe *Caminhão* e este seleciona a transição que representa tal ação.

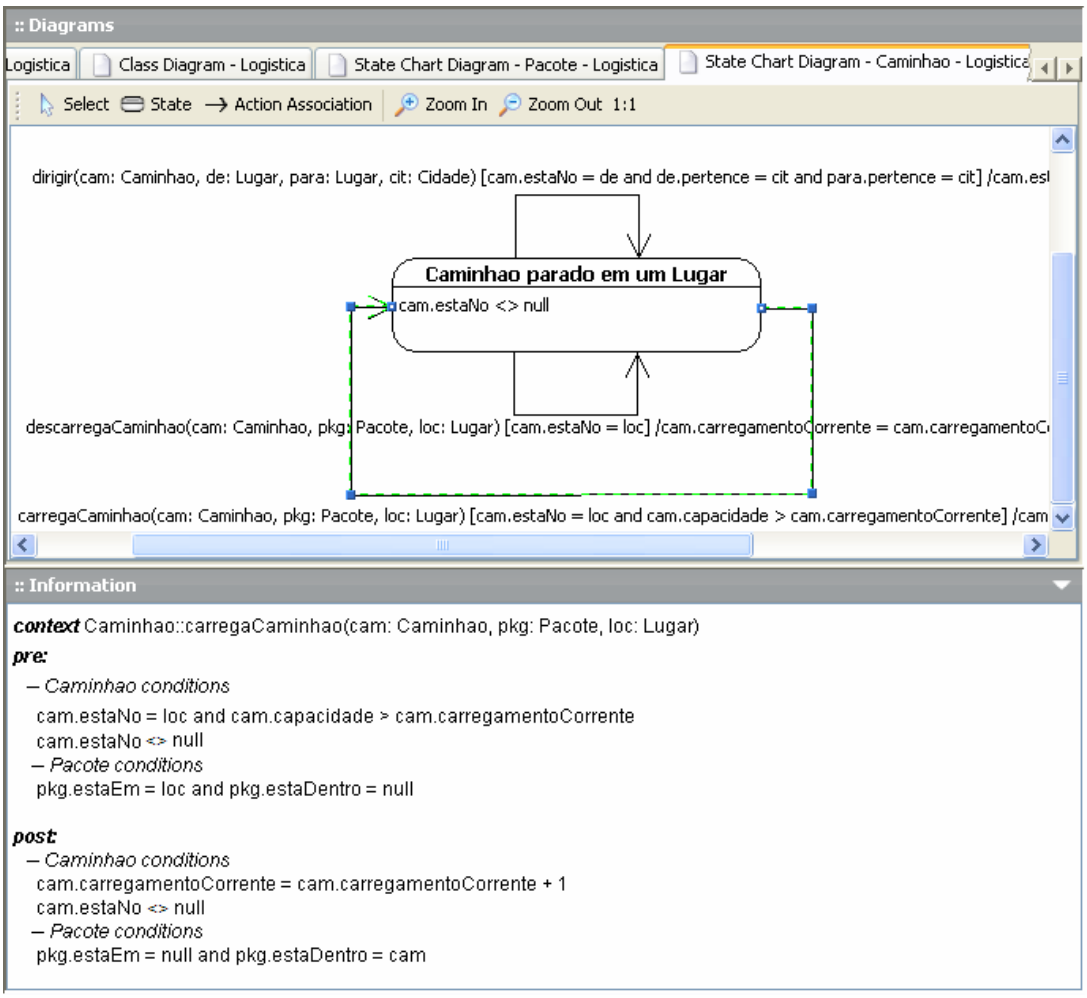


Figura 59 – Auxílio ao projetista durante a modelagem de ações em OCL.

É possível ver na Figura 59 que o itSIMPLE constrói um representação única em OCL no painel "*Informações Adicionais*" quando o projetista seleciona uma ação a ser elaborada. A ferramenta ainda separa as expressões em OCL por contexto para facilitar a visualização das procedências de tais expressões como, por exemplo, as expressões provenientes do *Diagrama de Estados* da classe *Pacote* são mostradas após o comentário "-- *Pacote conditions*" conforme mostra a Figura 59.

Inicialmente o itSIMPLE só trata expressões simples em OCL, ou seja, o uso de expressões em OCL é limitada devido aos processos de tradução, principalmente para PDDL. Essas limitações serão melhor detalhadas no tópico que discute o processo de tradução do

modelo para PDDL presente neste capítulo. Caso os processos de tradução para eventuais testes com planejadores gerais não sejam de interesse do projetista, as expressões em OCL podem ser realizadas livremente.

4.3.4. Agentes e Recursos – Diagrama de Objetos

Como visto no Capítulo 3, os Agentes e Recursos do domínio podem ser representados utilizando o *Diagrama de Objetos*. Assim cada domínio em um projeto no itSIMPLE possui um *Diagrama de Objetos*, chamado aqui de *Repositório*, onde estes elementos são declarados. A Figura 60 demonstra aonde o *Diagrama de Objetos* se encontra na estrutura de projetos do itSIMPLE no painel *"Explorador de Projetos*".

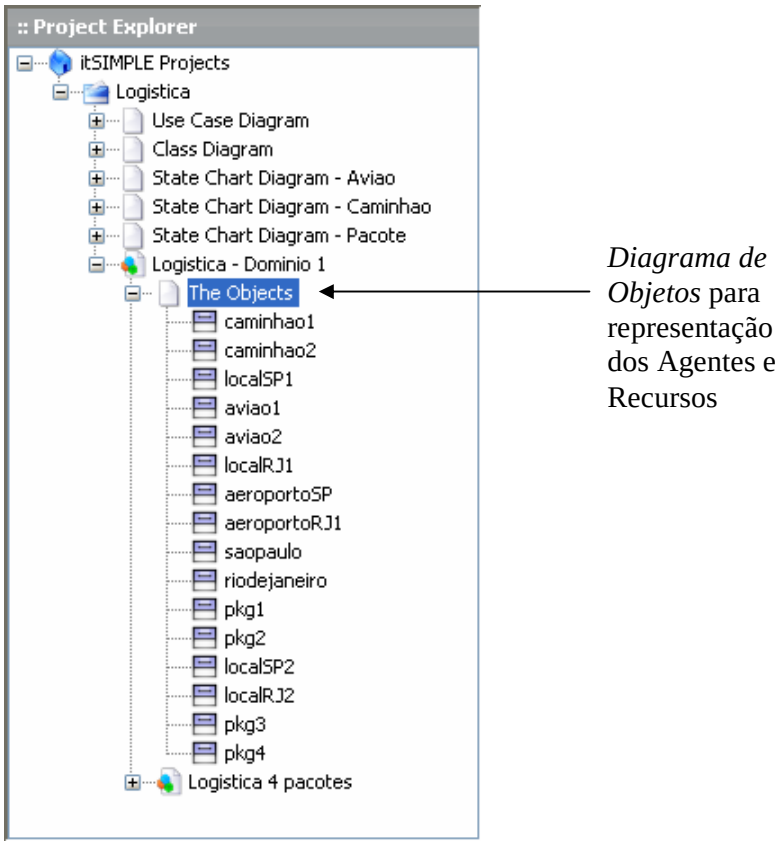


Figura 60 – Painel *Diagrama de Objetos "Explorador de Projetos"* com o *Diagrama de Objetos* para a representação de Agentes e Recursos identificado.

Além dos Agentes e Recursos é possível representar características imutáveis de um domínio. No exemplo do domínio *Logística* os locais *localSP1*, *localSP2* e *aeroportoSP* sempre pertencerão à cidade *saopaulo*, conforme visto na modelagem do Capítulo 3. A Figura 61 mostra a interface do itSIMPLE para a representação dos Agentes e Recursos de um domínio, no caso o domínio *Logística*.

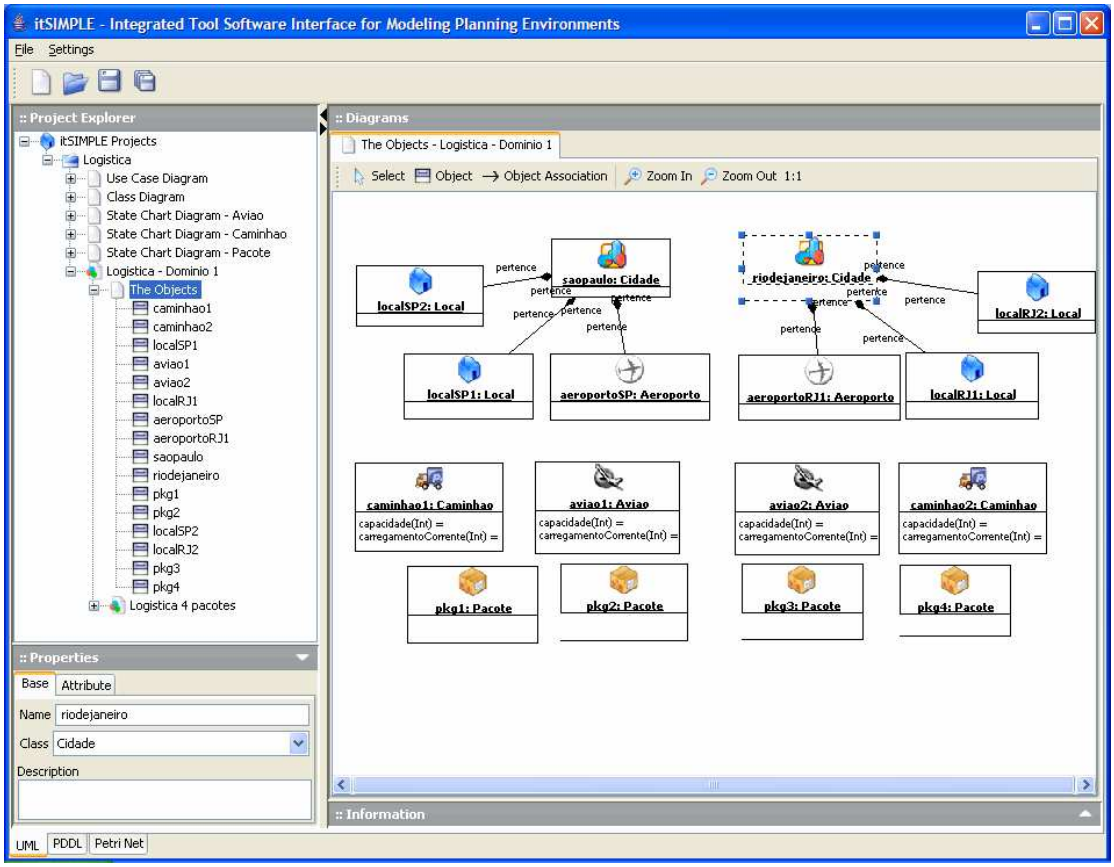


Figura 61 – *Diagrama de Objetos*, ou *Repositório*, para a representação de Agentes e Recursos no domínio *Logística*.

É possível perceber na Figura 61 que os objetos do diagrama também possuem imagens (provenientes de suas classes), o que também contribui para uma melhor visualização e modelagem do domínio de planejamento.

4.3.5. Modelagem do Problema – Snapshots

Foi visto anteriormente que a definição de um problema de planejamento é realizada através da criação de dois diagramas de objetos (*Snapshots*): o *Snapshot Inicial* e o *Objetivo*. Para tal, basta o projetista selecionar os objetos do *Repositóri*o do domínio, dar valores aos atributos e estabelecer associações de modo a criar estados que representem os estados inicial e objetivo (final). A Figura 62 e a Figura 63 demonstram a interface para a representação destes dois estados com os *Snapshots*.

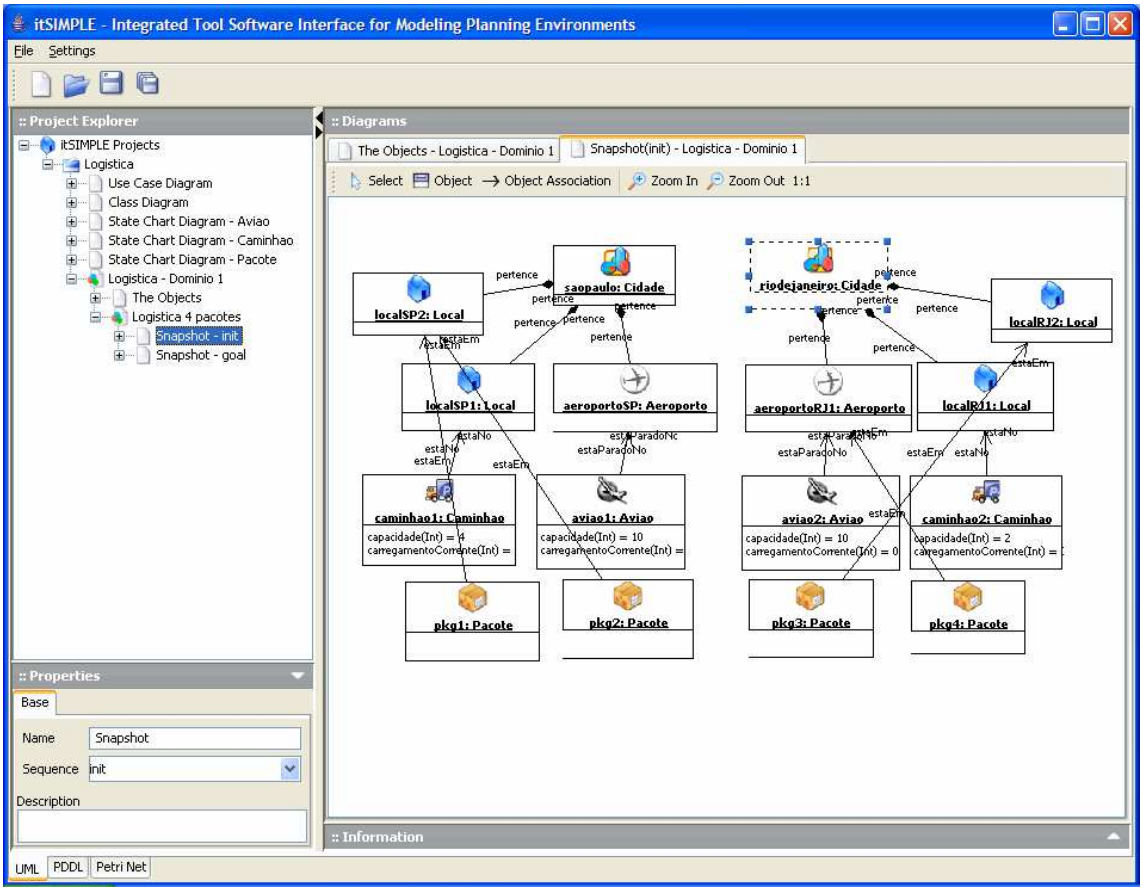


Figura 62 – Interface do itSIMPLE com *Snapshot Inicial* de um problema de planejamento no domínio *Logística*.

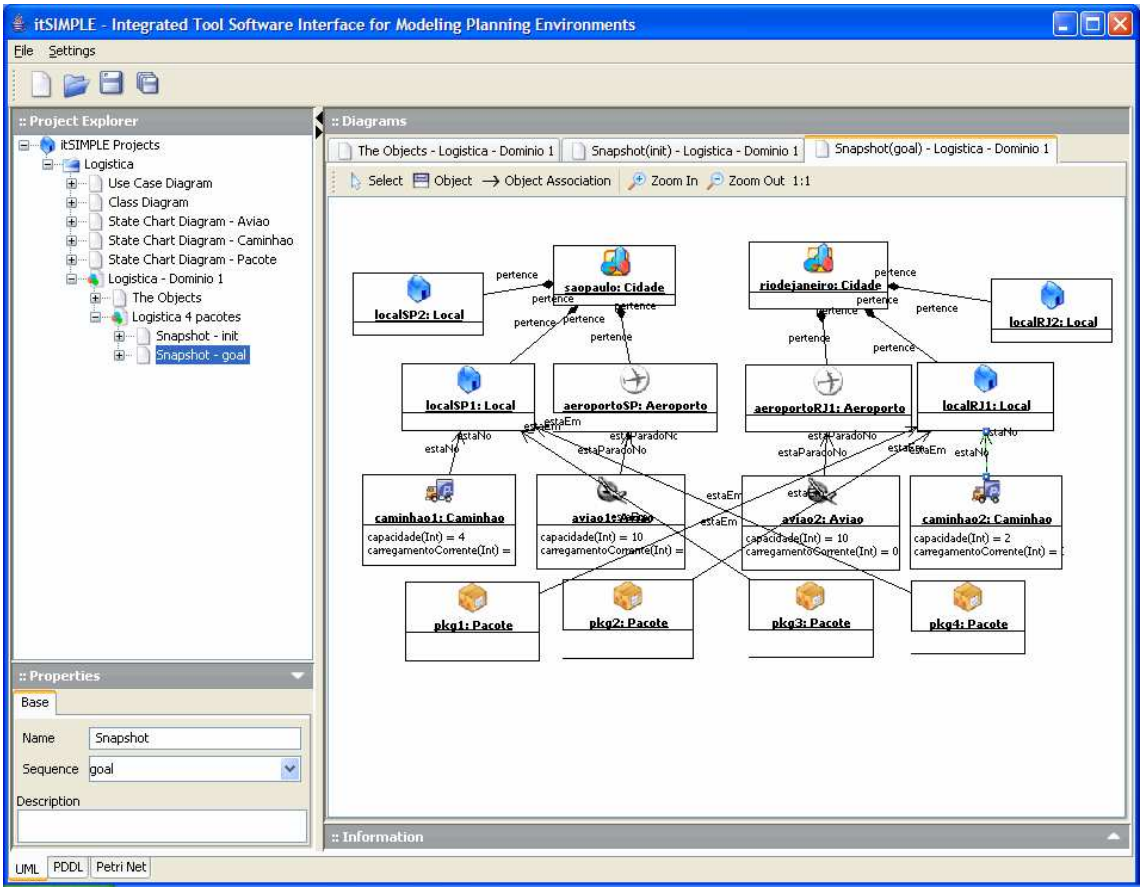


Figura 63 – Interface do itSIMPLE com *Snapshot Objetivo* de um problema de planejamento no domínio *Logística*.

A ferramenta disponibiliza um mecanismo para auxiliar o projetista a associar objetos, e conseqüentemente ajudá-lo a construir estados válidos, ou seja, *snapshots* coerentes. Quando o projetista associa dois objetos de um snapshot a ferramenta realiza duas verificações simples, sendo elas:

1. A ferramenta verifica primeiramente se as classes dos objetos sendo associados possuem alguma associação definida no *Diagrama de Classes*. Por exemplo, se o projetista tentasse associar os objetos *caminhao1* da classe *Caminhão* e *aviao1* da classe *Avião*, o itSIMPLE estaria informando o projetista que não existem associações entre estes objetos e portanto a associação desejada não é permitida. Já para o caso da

associação do *caminhao1* com o objeto *localSP1* da classe *Local* o itSIMPLE aprovaria tal associação faltando apenas uma última verificação descrita a seguir;

2. Caso exista uma associação entre as classes dos objetos no *Diagrama de Classes*, a ferramenta verifica então se as regras da multiplicidade da associação não serão violadas caso tal associação seja estabelecida. Por exemplo, se o objeto *caminhao1* já estiver associado ao *localSP1* pela associação *estaNo* e o usuário tentar associar o mesmo objeto *caminhao1* com um outro objeto *localSP2* da classe *Local*, o itSIMPLE primeiramente verifica que existe uma única associação possível entre estes objetos, que é a associação *estaNo*. Logo em seguida, o itSIMPLE verifica a que multiplicidade de tal associação foi declarada como sendo: um veículo pode estar em um e somente (1) um lugar em qualquer estado, já o no lugar podem estar diversos (ou nenhum) veículos (0..*) distintos em um determinado estado (conforme mostra a Figura 56). Seguindo esta regra o itSIMPLE verifica que o objeto *caminhao1* já esta associado por uma associação *estaNo* e então o mesmo não permite tal associação com outro objeto.

Estas verificações providas pela ferramenta inibem erros durante a associação de objetos e, principalmente, auxilia o projetista a criar estados iniciais e finais (*snapshots*) válidos perante as regras modeladas no *Diagrama de Classes*. Através da ferramenta o usuário pode criar quantos problemas de planejamento desejar, para um mesmo modelo de domínio conforme discutido anteriormente.

4.3.6. Análise do Modelo do Domínio

Para realizar as análises (Modular e de Interfaces) propostas no Capítulo 3, o itSIMPLE provê uma interface amigável onde o projetista pode visualizar, analisar e verificar a estrutura das Redes de Petri resultantes dos *Diagramas de Estados* modelados em UML. Ao selecionar a linguagem Redes de Petri (*Petri Net*) na "*Barra de Mudança de Linguagem*" a ferramenta muda a perspectiva do usuário para o contexto da *Análise Modular* e *de Interfaces* em Redes de Petri.

Como o itSIMPLE permite que o usuário trabalhe com mais de um projeto, o mesmo pode escolher qual deles deseja analisar os aspectos dinâmicos modelados nos *Diagramas de Estados*. Ao selecionar um projeto na interface de análise em RdP, o itSIMPLE disponibiliza todos os *Diagramas de Estados* disponíveis do projeto selecionado para que o projetista possa escolher quais serão analisados. Caso o projetista escolha apenas um *Diagrama de Estados* a ser analisado, o itSIMPLE representará tal diagrama na forma de um módulo em Redes de Petri da mesma maneira daquela apresentada no Capítulo 3. A visualização de apenas um módulo em RdP indica que o projetista está realizando a *Análise Modular*.

A ferramenta permite também que o usuário escolha mais de um diagrama (módulo) a ser analisado para estudar e verificar as interfaces entre os módulos desejados. O projetista pode escolher, por exemplo, dois ou todos *Diagramas de Estados* do projeto para visualizar a RdP resultante e assim realizar a *Análise de Interfaces* (também apresentada no Capítulo 3).

Caso o projetista deseje exportar as Redes de Petri geradas o itSIMPLE disponibiliza tais redes em *PNML modular* (KINDLER; WEBER, 2001) um modo transparente e rápido. O usuário pode salvar o arquivo PNML contento a rede para realizar eventuais análises com ferramentas específicas de acordo com o interesse do mesmo.

A Figura 64 demonstra a interface para a análise dos aspectos dinâmicos proposta no presente trabalho, bem como suas principais características.

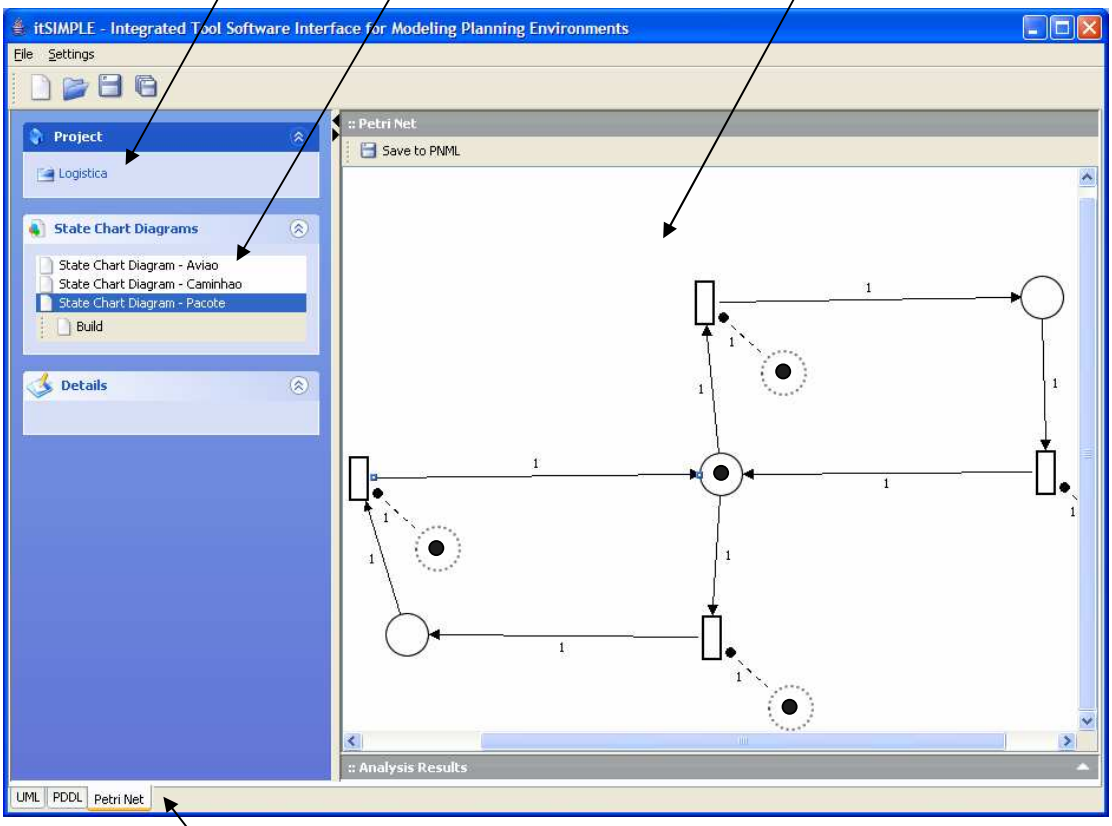


Figura 64 – Interface de análise dinâmica em Redes de Petri do modelo do domínio no itSIMPLE.

Os processos de representação dos *Diagramas de Estados* da UML (OMG, 2001) em *PNML modular* (KINDLER; WEBER, 2001), realizados no itSIMPLE para as análises *Modulares* e *de Interfaces*, são apresentados com mais detalhes nos próximos tópicos.

4.4. Traduzindo os Diagramas de Estados para PNML modular

Conforme visto no Capítulo 3, o presente trabalho propõe mecanismos de análises dos aspectos dinâmicos do modelo a partir da representação dos *Diagramas de Estados* em Redes de Petri. De fato, são propostas duas análises: *Análise Modular*, onde cada diagrama (módulo) é representado em uma rede e esta é analisada isoladamente; e a *Análise de Interfaces*, onde o projetista pode analisar a interação entre os diagramas (módulos) representados também em uma única rede. Em ambas as análises o itSIMPLE realiza um processo de tradução dos *Diagramas de Estados*, selecionados para a análise, para uma rede em *PNML modular* (KINDLER; WEBER, 2001). Com a RdP gerada em *PNML modular* o itSIMPLE provê mecanismos de visualização da rede conforme apresentado no Capítulo 3.

Para gerar as RdP das análises, o itSIMPLE trabalha sobre a estrutura da *PNML modular* demonstrada na Figura 65, onde é possível visualizar os principais elementos (as principais *tags*) desta estrutura. Esta figura mostra tanto a estrutura de um módulo em *PNML modular* quanto a estrutura de um arquivo *PNML modular* contento uma rede (KINDLER; WEBER, 2001). É possível perceber na figura a seguir que uma rede na *PNML modular* pode possuir instância (<*instance*>) dos módulos definidos no arquivo.

Estrutura de um módulo em PNML

```
<module name="">
       <interface>
              ...
              importPlace,
              importTransition
              exportPlace e
              exportTramsition
              ...
       </interface>
       ...
       places,
       transitions,
       arcs,
       referencePlace e
       referenceTransition
       ...
</module>
```

Estrutura da PNML modular

```
<pnml … >
       <net id="">
              ...
              places,
              transitions e
              arcs
              ...
              <instance id="" ref="">
                     ...
                     importPlace e
                     importTransition
                     ...
              </instance>
              ...
       </net>
       ...
       <module name="">
              ...
       </module>
       ...
</pnml>
```

Figura 65 – Elementos principais de um módulo e de uma arquivo em *PNML modular* gerados pelo itSIMPLE.

O itSIMPLE utiliza o mesmo processo de tradução tanto para gerar a rede da *Análise Modular*, quanto para gerar a rede da *Análise de Interfaces*. O processo de tradução tem como principal entrada (*imput*) uma lista dos *Diagramas de Estados* selecionados pelo projetista, que caso este forneça uma lista contendo apenas um diagrama, o mesmo estará realizando uma *Análise Modular* e caso seja fornecido uma lista com mais de um diagrama, será realizada a *Análise de Interfaces*. Para o itSIMPLE, é indiferente o processo de tradução para ambas.

Na versão do itSIMPLE apresentada neste trabalho são tratados apenas os casos onde não há repetição de ações para um mesmo *Diagrama de Estados* (ponto discutido no Capítulo 3 no tópico *Análise do Modelo do Domínio*). Assim, casos onde uma mesma ação aparece duas ou mais vezes em um mesmo diagrama não são tratados nesta versão da ferramenta durante o processo de tradução para Rede de Petri. O pseudocódigo da Figura 66 demonstra como uma lista de *Diagramas de Estados* é tratada para gerar sua respectiva rede.

```
stateChartListToModularPetriNet(stateChartList, project)
    Início
      pnml ← estrutura vazía de uma rede PNML

      //1. cria um módulo PNML para cada Diagrama de Estados contidos na lista stateChartList
      para cada elemento da lista stateChartList faça
            stateChartDiagram ← Diagrama de Estados corrente
            //transforma um Diagrama de estados em um módulo em PNML
             stateChartModule ← stateChartDiagramToPetriModule(stateChartDiagram, project)

             módulo stateChartModule é adicionado na estrutura pnml
      fimpara

      moduleList ← todos os módulos contidos em  pnml  (resultantes do passo 1)

      //2.cria as intâncias dos módulos na rede em pnml e  trata todos os elementos importantos e exportados de
      //  todos os módulos em  moduleList.
      pada cada módulo de moduleList faça
            module ← módulo corrente

            //2.1 cria uma instância (<instance>) do módulo module na rede (<net>) da pnml
            instance ←  instância de module na rede (<net>) da pnml

            //2.2 trata todos as ações importadas (importTransition) presentes na interface <interface> do
            //  módulo  corrente module
            dealwithImportTransitions(module, instance, moduleList, pnml)

            //2.3 trata todos as ações exportadas (exportTransition) presentes na interface <interface> do módulo
            //  corrente module
            dealwithExportTransitions(module, instance, moduleList, pnml, project)
      fimpara

      retorna pnml

    Fim
```

Figura 66 – Pseudocódigo da tradução dos *Diagramas de Estados* para *Redes de Petri modular* tanto para a *Análise Modular* quanto para a *Análise de Interfaces*.

Como é possível perceber no pseudocódigo da Figura 66, existem dois procedimentos principais para a tradução. O primeiro faz com que cada elemento da lista de diagramas selecionados seja traduzido para um módulo em *PNML modular*(baseado na estrutura de um módulo demonstrada na Figura 65). Este procedimento faz com que o arquivo PMNL tenha todos os módulos a serem analisados. Já o segundo procedimento instancia os módulos na rede do arquivo PNML e, ao instanciar os módulos, as *transições importadas* e *exportadas* são tratadas, bem como os *Gates* (que representam as dependências externas), para que a rede resultante seja representada conforme apresentado no Capítulo 3. Estes dois procedimentos principais são apresentados a seguir.

4.4.1. Criando os Módulos da *PNML modular*

Os elementos de um *Diagrama de Estados* são diretamente mapeados para os elementos de uma Rede de Petri lugar/transição. Como visto no Capítulo 3, algumas transições provenientes dos diagramas são diferenciadas quando representadas em RdP, como é o caso das *transições exportadas* e as *transições importadas*. Para ilustrar as regras de representação em *PNML modular* do diagramas (discutidas no Capítulo 3), a Tabela 5 apresenta como as características de um *Diagrama de Estados* qualquer *D*, definido como sendo da classe C_D, são representadas em um módulo *m* na *PNML modular* da forma proposta no presente trabalho.

Tabela 5 - Mapeamento entre elementos do *Diagrama de Estados* e os elementos na *PNML modular* proposta.

Característica UML	**PNML modular (módulo)**
(1) Um estado qualquer *s* no diagrama *D*. Obs: o elemento *estado inicial* não é representado na rede, o mesmo apenas indica a marcação inicial da rede.	<module name="m" <interface> … </interface> … **<place id="**{s}**"/>** … </module>

(2) Uma ação *t* realizada pela classe $C_{D,}$ que sai do estado *s1* e vai para o estado *s2*, e só aparece em *D*, ou seja, sua execução depende exclusivamente da classe $C_{D.}$ A ação *t* é uma *transição normal* no módulo *m*.	<module name="m"> <interface> … </interface> … <place id="{s1}"/> <place id="{s2}"/> **<transition id="{t}"/>** <arc source="{s1}" target="{t}" /> <arc source="{t}" target="{s2}" /> … </module>
(3) Uma ação *t* realizada pela classe $C_{D,}$ que sai do estado *s1* e vai para o estado *s2*, e aparece em outros diagramas além de *D*, ou seja, sua execução depende de outras classes além de C_D. A ação *t* é uma *transição exportada* no módulo *m*.	<module name="m"> <interface> … **<exportTransition id="{t}" ref="{t}" />** … </interface> … <place id="{s1}"/> <place id="{s2}"/> **<transition id="{t}"/>** <arc source="{s1}" target="{t}" /> <arc source="{t}" target="{s2}" /> … </module>
(3) Uma ação *t* realizada pela classe $C_t \neq C_D$ (ou seja, C_D não é a classe agente da ação *t*) que sai do estado *s1* e vai para o estado *s2*. A ação *t* é uma *transição importada* no módulo *m*.	<module name="m"> <interface> … **<importTransition id="{t}" />** … </interface> … <place id="{s1}"/> <place id="{s2}"/> **<referenceTransition id="{t}" ref="{t}"/>** <arc source="{s1}" target="{t}" /> <arc source="{t}" target="{s2}" /> … </module>

Observação: casos não presentes nesta tabela não são tratados. O termo {*elemento*} indica o identificador do *elemento* (um nome ou um número).

A Figura 67 a seguir demonstra o pseudocódigo do procedimento de criação de cada módulo em *PNML modular* a partir de um dado *Diagrama de Estados*, conforme os casos presentes na Tabela 5. Este procedimento é um dos primeiros passos realizados no

pseudocódigo do processo de tradução como um todo (*stateChartListToModularPetriNet* da Figura 66).

```
stateChartToModularPetriNet(stateChartDiagram, project)
    Início
      module ← estrutura vazía de uma módulo em PNML
      class ← classe do Diagrama de Estados stateChartDiagram

      //1. cria todos os lugares do módulo module a partir dos estados em stateChartDiagram
       para todos os estados em stateChartDiagram cria-se um lugar (<place>) em module

      //2. cria as transições
      pada cada ação contida em stateChartDiagram faça
            action ← ação corrente

            se  action é realizada pela classe class:
                cria uma transição normal <transition> em module.
                se action aparece em outros Diagramas de Estados em project então
                        cria-se uma transição exportada <exportTransition> na interface do módulo module
                fimse
            senão
                 cria-se um lugar-referência <referenceTransition> em module.
                 cria uma transição importada <impotTransition> na interface <interface> do módulo module

            fimse

            cria os arcos <arc> que ligam a transição aos seus respectivos lugares
      fimpara

      retorna module

    Fim
```

Figura 67 – Pseudocódigo da tradução do um *Diagrama de Estados* para um módulo em *PNML modular*.

Para exemplificar o processo de representação de um módulo em *PNML modular* a partir de uma *Diagrama de Estados* realizado pelo itSIMPLE, segue abaixo uma representação simplificada de módulo, proveniente do *Diagrama de Estados* da classe *Avião* (identificado aqui por {Aviao}), com seus principais elementos em *PNML modular:*

```
<module name="{Aviao}">
   <interface>
        <exportTransition id="{Aviao::descarregaAviao}" ref="{Aviao::descarregaAviao}" />
        <exportTransition id="{Aviao::carregaAviao}" ref="{Aviao::carregaAviao}" />
   </interface>
   <place id="{Aviao parado em um Aeroporto}" />
   <transition id="{Aviao::pilotar}" />
   <transition id="{Aviao::descarregaAviao}" />
   <transition id="{Aviao::carregaAviao}" />
   <arc source="{Aviao parado em um Aeroporto}" target="{Aviao::pilotar}" />
   <arc source="{Aviao::dirigir}" target="{Aviao parado em um Aeroporto}" />
   <arc source="{Aviao parado em um Aeroporto}" target="{Aviao::descarregaAviao}" />
   <arc source="{Aviao::descarregaAviao}" target="{Aviao parado em um Aeroporto}" />
   <arc source="{Aviao parado em um Aeroporto}" target="{Aviao::carregaAviao}" />
   <arc source="{Aviao::carregaAviao}" target="{Aviao parado em um Aeroporto}" />
</module>
```

Depois de representado os módulos a serem analisados no arquivo PNML, as instâncias destes são criadas na rede (<net>), observadas as restrições quanto às *transições exportadas* e *importadas*, bem como os *Gates*, conforme apresentado a seguir.

4.4.2. Criando as Instâncias dos Módulos na Rede da *PNML modular*

Conforme visto no Capítulo 2, para se instanciar um módulo em uma rede é necessário fornecer os parâmetros (lugares ou transições importadas) do módulo para sua inicialização. Como os elementos (lugares e/ou transições) importados serão fundidos com os elementos exportados dos módulos sendo analisados, é necessário tratar também todos os elementos exportados dos módulos.

Conforme demonstrado no pseudocódigo do processo *stateChartListToModularPetriNet*, para todo módulo em PNML (proveniente dos diagramas) é criada uma instância na rede e em seguida são tratadas, primeiramente, as *transições importadas* e em seguida as *transições exportadas*.

Para tratar as *transições importadas* de cada módulo, o itSIMPLE realiza o procedimento chamado *dealwithImportTransition* (requisitado no processo *stateChartListTo-*

ModularPetriNet) que trata como essas transições serão referenciadas na instância do módulo na rede. A Figura 68 demonstra o pseudocódigo do procedimento *dealwithImportTransition.*

```
dealwithImportTransitions(module, instance, moduleList, pnml)
  Início
     // trata todos as ações importadas (importTransition) presentes na interface <interface> do módulo
     //  corrente module
     para cada importTransition contido na interface do módulo module faça
          importTransition ← transição importada corrente
          se algum dos módulo em moduleList exportar a transição que importTransition importa então
                A instância instance do módulo module na rede recebe a referência da transição exportada e do
                     seu respectivo módulo.
          senão
             // se a transição original não foi encontrada em nenhum módulo de moduleList, cria-se uma transição
             // original <transitions>  na  rede <net>
              se ainda não foi criada tal transição original na rede então
                     transition ← cria-se uma transição (<transition>) na rede <net> da pnml com a mesma
                        identificação que importTransition
                     gate ← cria-se uma pseudo lugar (<place>) na rede <net> da pnml que representa a
                        dependência de outro módulo que não esta contido em moduleList
                     arcRead ← cria-se um arco habilitador em module que sai de gate e vai para transition
             fimse
          fimse
     fimpara
  Fim
```

Figura 68 – Pseudocódigo para tratar as *ações importadas* dos módulos na *PNML modular*.

Para melhor visualizar os casos tratados no pseudocódigo acima, a Tabela 6 provê uma forma alternativa de entender o procedimento de tratamento das *transições importadas*. Nesta tabela são utilizados os termos: *module* identificando o módulo em que se encontra a *transição importada*; *instance* identificando a instância de *module* na rede; *moduleList* representando a lista de todos os módulos em PNML sendo analisados; e *pnml* a estrutura do arquivo PNML sendo construído.

Tabela 6 - Tratamento das *transições importadas* dos módulos em PNML para estabelecer as instâncias dos mesmos na rede.

Característica UML	PNML modular (rede)
(1) Se uma transição importada *ti* em *module* representa uma ação *a* e um outro módulo *module1*, presente na lista *moduleList*, exporta uma transição *te* que também representa *a*, então a instância *instance* de *module* recebe *te* como parâmetro para a transição *ti*.	<pnml> <net> ... <instance id="" ref="{module}" > **<importTransition parameter="{ti}" instance="{module1}" ref="{te}"/>** </instance>... <instance id="" ref="{module1}"> </instance> ... </net> <module name="{module}" <interface> ... <importTransition id="**{ti}**" />... </interface> ... </module>... <module name="{module1}" <interface> ... <exportTransition id="**{te}**" />... </interface> ... </module>... </pnml>
(2) Se uma transição importada *ti* em *module* representa uma ação *a* e nenhum outro módulo, presente na lista *moduleList*, exporta tal transição que também representa *a*, então cria-se uma transição *t* na rede para representar tal transição. É criado também um *Gate* na rede para representar a dependência de um módulo que não estará na análise (ou seja, não está em *moduleList*). Novamente, a instância *instance* de *module* recebe *t* como parâmetro para a transição *ti*	<pnml> <net> ... **<transition id="{t}" />** **<place id="{gate}"/>** **<arc source="{gate}" target="{t}" />** *//tipo habilitador* <instance id="" ref="{module}" > **<importTransition parameter="{ti}" ref="{t}"/>** </instance> ... </net> <module name="{module}" <interface> ... <importTransition id="**{ti}**" />... </interface> ... </module>... </pnml>

Observação: casos não presentes nesta tabela não são tratados. O termo {*elemento*} indica o identificador do *elemento* (um nome ou um número).

Depois de tratar as *transições importadas*, o itSIMPLE trata as *transições exportadas* de cada módulo. Para isso, é realizado o procedimento chamado *dealwithExportTransition* (também requisitado no processo *stateChartListToModularPetriNet*) que trata principalmente

os casos onde existem dependências externas aos módulos presentes na lista de módulos a serem analisados. A Figura 69 demonstra o pseudocódigo do procedimento *dealwithExportTransition.*

```
dealwithExportTransitions(module, instance, moduleList, pnml, project)
    Início
      // trata todos as ações exportadas (exportTransition) presentes na interface <interface> do módulo
      //  corrente module
      para cada exportTransition contido na interface do módulo module faça
            exportTransition ← transição exportada corrente
            se outros módulos em project não presentes em moduleList importam a transição exportada por
                 exportTransition em module então
                 //cria-se um gate na rede <net> para representar a dependência de um módulo que não está
                 //     presente em moduleList. Este gate é devidamente referenciado dentro de module
                 gate ← cria-se uma lugar gate (<place>) na rede <net> da pnml que representa a
                           depencência de outro módulo que não esta contido em moduleList
                //cria-se um lugar de referencia dentro do módulo representando o gate
                 importPlace ← cria-se uma lugar importado dentro da interface (<interface>) do módulo module
                           que faz referencia ao gate
                refencePlace← cria-se uma lugar referência que indica que o lugar é importado.
                arcRead ← cria-se um arco habilitador que sai de gate e vai para transição (<transition>) que
                           é exportada
                estabelece na instância instance do módulo module o Gate como parâmetro do lugar importado
                      importPlave.
            fimse
      fimpara
    Fim
```

Figura 69 – Pseudocódigo para tratar as *ações exportadas* dos módulos na *PNML modular*.

Novamente, para melhor visualizar os casos tratados no pseudocódigo da Figura 69, a Tabela 7 provê uma forma alternativa de entender o procedimento de tratamento das *transições exportadas*. Nesta tabela são utilizados os termos: *module* identificando o módulo em que se encontra a *transição importada*; *instance* identificando a instância de *module* na rede; *moduleList* representando a lista de todos os módulos em PNML sendo analisados; *pnml* a estrutura do arquivo PNML sendo construído; e *project* representando o projeto sendo elaborado no itSIMPLE.

Tabela 7 - Tratamento das *transições exportadas* dos módulos em PNML para estabelecer eventuais dependências com *Gates* na rede.

Característica UML	**PNML modular (rede)**
(1) Se uma transição exporta *ie* em *module* representa uma ação *a* e outros módulos não presentes na lista *moduleList*, importam transições que representam a mesma a ação *a*, então existem dependências externas aos módulos em *moduleList*. Esta dependência é tratada através da criação de um lugar *Gate* na rede e sua referencia é inserida no modulo *module*, juntamente com um arco para a transição *te*. A instância *instance* de *module* recebe *Gate* como parâmetro para o lugar de referência criado no módulo.	<pnml> <net> ... **<place id="{gate}"/>** <instance id="" ref="{module}" > **<importPlace parameter="{gate}" ref="{gate}"/>** </instance> ... </net> <module name="{module}" <interface> ... **<importPlace id="{gate}" />**... </interface> ... <transition id="**{te}**" /> **<referencePlace id="{gate}" ref="{gate}" />** **<arc source="{gate}" target="{te}" />** *//tipo habilitador* ... </module>... ... </pnml>

Observação: O termo {*elemento*} indica o identificador do *elemento* (um nome ou um número).

Para exemplificar os procedimentos de instância e de tratamentos das transições, a estrutura XML a seguir demonstra o exemplo simplificado de um arquivo PNML com uma rede gerada para a *Análise Modular* do comportamento da classe *Avião* (ilustrada na Figura 42 no Capítulo 3).

```
<pnml>
   <net id="n1">
        <place id="{condição externa de descarregaAviao}"/>
        <place id="{condição externa de carregaAviao}"/>
        <instance id="{Aviao}" ref="{Aviao}" />
               <importPlace parameter=”{condição externa de descarregaAviao}”
                      ref=”{condição externa de descarregaAviao}” />
               <importPlace parameter=”{condição externa de carregaAviao}”
                      ref=”{condição externa de carregaAviao}” />
        </instance>
   </net>
   <module name="{Aviao}">
        <interface>
           <exportTransition id="{Aviao::descarregaAviao}" ref="{Aviao::descarregaAviao}" />
           <exportTransition id="{Aviao::carregaAviao}" ref="{Aviao::carregaAviao}" />
           <importPlace id="{condição externa de descarregaAviao}" />
           <importPlace id="{condição externa de carregaAviao}" />
        </interface>
        <place id="{Aviao parado em um Aeroporto}" />
        <transition id="{Aviao::pilotar}" />
        <transition id="{Aviao::descarregaAviao}" />
        <transition id="{Aviao::carregaAviao}" />
        <arc source="{Aviao parado em um Aeroporto}" target="{Aviao::pilotar}" />
        <arc source="{Aviao::dirigir}" target="{Aviao parado em um Aeroporto}" />
        <arc source="{Aviao parado em um Aeroporto}" target="{Aviao::descarregaAviao}" />
        <arc source="{Aviao::descarregaAviao}" target="{Aviao parado em um Aeroporto}" />
        <arc source="{Aviao parado em um Aeroporto}" target="{Aviao::carregaAviao}" />
        <arc source="{Aviao::carregaAviao}" target="{Aviao parado em um Aeroporto}" />

        <referencePlace id="{condição externa de descarregaAviao}"/>
        <referencePlace id="{condição externa de carregaAviao}" />
        <arc source="{condição externa de carregaAviao}" target="{Aviao::descarregaAviao}" />
        <arc source="{condição externa de carregaAviao}" target="{Aviao::carregaAviao}" />
   </module>
</pnml>
```

Com a criação das instâncias dos módulos e os tratamentos das transições apresentados neste tópico, o arquivo PNML da Rede de Petri está pronto para ser visualizado no itSIMPLE (e eventualmente exportado) pelo projetista conforme as visualizações propostas no Capítulo 3.

4.5. Traduzindo o Modelo para PDDL

O ambiente do itSIMPLE permite que o projetista exporte os modelos dos domínios, e seus respectivos problemas de planejamento, modelados em UML.P para a linguagem PDDL. Todo o modelo, como visto anteriormente, é representado em um arquivo XML (a *Representação Core*) de onde a ferramenta extrai as informações para, primeiramente, representar (traduzir) o modelo em XPDDL (GOUGH, 2004) e em seguida em PDDL

(MCDERMOTT, 1998). De fato, o projetista pode escolher em qual dessas duas linguagens o mesmo deseja trabalhar, XPDDL ou PDDL.

O itSIMPLE disponibiliza uma interface simples na qual o usuário pode visualizar e trabalhar com os modelos em PDDL. Ao selecionar a linguagem PDDL na "*Barra de Mudança de Linguagem*" a ferramenta muda a perspectiva do usuário para o contexto PDDL. Com isso o usuário pode escolher o domínio, e seus respectivos problemas de planejamento, e o itSIMPLE traduz automaticamente os modelos para PDDL de um modo transparente e rápido. O usuário pode selecionar e salvar o modelo do domínio em um arquivo PDDL, bem como selecionar o problema desejado e salvá-lo também em um arquivo PDDL, para realizar os eventuais testes com planejadores.

Caso o projetista sinta a necessidade de refinar o modelo em PDDL já traduzido pelo itSIMPLE, a ferramenta disponibiliza um editor de PDDL (novamente sem verificações de sintaxe) para que o trabalho com esta linguagem se torne mais fácil. A Figura 70 mostra a interface do itSIMPLE onde o usuário pode exportar os modelos para uma representação em PDDL, ou eventualmente refiná-lo.

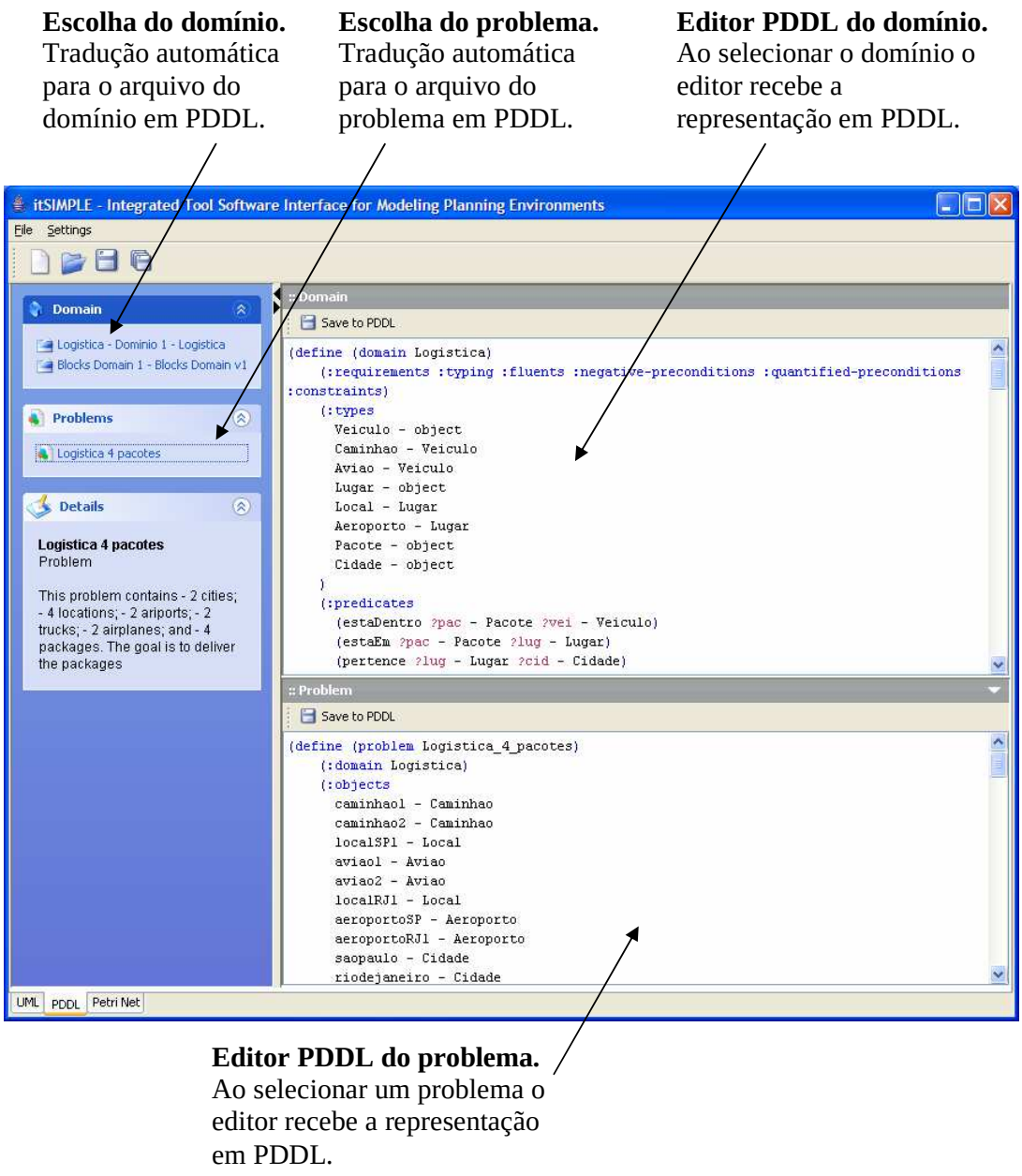


Figura 70 – Visualização da representação em PDDL do modelo no itSIMPLE.

Como visto no Capítulo 2, os planejadores gerais precisam de uma descrição do domínio e uma descrição do problema em PDDL para que estes possam iniciar o processo de resolução do problema de planejamento. Para que estas duas representações sejam formadas a partir da *Representação Core* (de fato, dos diagramas UML), o itSIMPLE realiza dois processos de

tradução para a PDDL: a *tradução do domínio* e a *tradução do problema* de planejamento. Cada uma dessas traduções é fornecida ao usuário nos *Editores de PDDL* (conforme mostra a Figura 70). Os dois processos de tradução são apresentados a seguir.

4.5.1. Tradução do Domínio de Planejamento

A versão da ferramenta desenvolvida para este trabalho pode fornecer um modelo do domínio em PDDL com características provenientes das versões PDDL1.2 (MCDERMOTT, 1998), PDDL2.1 (FOX; LONG, 2003) e PDDL3.0 (GEREVINI; LONG, 2005). Conforme visto no Capítulo 2, um domínio em PDDL (ou em XPDDL), considerando as versões acima citadas, possui uma estrutura geral bem definida. Os principais elementos desta estrutura tratados no itSIMPLE são demonstrados na Figura 71.

PDDL

```
(define (domain {nome})

       (:requirements … )

       (:types … )

       (:predicates … )

       (:functions … )

       (:constraints … )

       (:action {nome_ac}
              :parameters ( … )
              :precondition ( … )
              :effect ( … )
       )
       …
)
```

XPDDL

```
<xpddl:domain … >
       <name>{nome}</name>
       <requirements> …
       </requirements>
       <types> …
       </types>
       <predicates> …
       </predicates>
       <functions> …
       </functions>
       <constraints> …
       </constraints>
       <actions>
          <action name=″{nome_ac}″>
              <parameter …/> …
              <precondition> …
              </precondition>
              <effect> …
              </effect>
          </action>
          …
       </actions>
</xpddl:domain>
```

Figura 71 – Elementos principais de um domínio em PDDL tratados no itSIMPLE.

Baseado nesta estrutura, o itSIMPLE realiza um procedimento que passo a passo extrai as informações provenientes da *Representação Core* e preenche cada elemento principal da XPDDL (`<requirements>`, `<types>`, `<predicates>`, entre outros). Logo depois o itSIMPLE traduz o modelo para PDDL a partir da XPDDL. Para ilustrar o procedimento de tradução para XPDDL, a Figura 72 representa o pseudocódigo do mesmo. Este pseudocódigo demonstra que dado um domínio de um determinado projeto no itSIMPLE, a ferramenta retorna a representação completa em XPDDL do domínio fornecido.

```
XMLToXPDDLDomain(dominio, projeto)
  Início
    xppdldomain ← estrutura vazía de um domínio em XPDDL

    //1. nomeia o domínio em XPDDL
    nomeia em xpddldomain nó referente ao nome do domínio

    //2. preenche os tipos em <types>
    adiciona no nó <types> do xpddldommain todas as classes contidas em projeto

    //3. preenche os predicados em <predicates>
    para cada associação de classes contida em projeto faça
        assoc ← associação corrente
        adiciona um predicado para cada rolename da associação assoc
        se nenhuma extremidade da associação possuir rolename então
          cria um predicado de acordo com a navegação da associação assoc
          adiciona predicado no nó <predicates> do xpddldomain
        fimse
    fimpara
    para cada atributo de todas as classes contidas em projeto faça
       attr ← atributo corrente
        se o attr seja do tipo booleano ou não-primitivo então
          cria um predicado no nó <predicates> do xpddldomain
        fimse
    fimpara

    //4. preenche as funções em <functions>
    para cada atributo de todas as classed contidas em projeto faça
       attr ← atributo corrente
        se o attr seja do tipo númerico (tipos "int" ou "float") então
          cria um predicado no nó <functions> do xpddldomain
        fimse
    fimpara

    //5. preenche as restrições em <constraints>
    para cada extremidade de associação com multiplicidade 1 ou 0..1 contida em projeto faça
      cria uma restrição e adiciona no nó <constraints> do xpddldomain
    fimpara

    //6. preenche as ações em <actions>
    para cada ação encontrada nas classes do Diagrma de Classes em projeto faça
      cria uma ação <action> com os parâmetros e adiciona em <actions> do xpddldomain
      reúne todos as pré-condições de todos os Diagramas de Estados onde a ação aparece
      constrói as pré-condições em <precondition>
      reúne todos as pós-condições de todos os Diagramas de Estados onde a ação aparece
      constrói as pós-condições em <effect>
    fimpara

    //7. refina as ações com predicados negados nas pós-condições
    refina ações

    //8. adiciona os requirements em <requirements>
    adiciona os requerimentos em <requirements> de acordo com o que já foi preenchido em xppdldomain

    retorna xpddldomain

  Fim
```

Figura 72 – Pseudocódigo da tradução do domínio para XPDDL.

Conforme visto na Figura 72, o pseudocódigo de tradução para XPDDL possui, de um modo geral, 8 (oito) passos importantes. Esses oito passos focalizam no preenchimento dos elementos principais, tais como *tipos, predicados, funções, restrições, ações* e os *requisitos* (*requirements*), da PDDL. Depois de preenchido e formado o modelo XPDDL, o itSIMPLE traduz tal modelo para PDDL, que é devidamente fornecido ao usuário. Este processo de tradução de XPPDL para PDDL, proveniente do trabalho (GOUGH, 2004), foi implementado nesta proposta para que o projetista possa trabalhar com ambas as linguagens, principalmente com a PDDL.

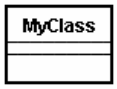
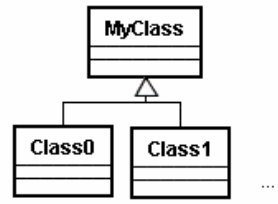
A seguir são discutidos os principais pontos do procedimento *XMLtoXPDDLDomain* através dos elementos principais da PDDL. Durante as explicações a seguir, são apresentados também os mapeamentos de cada característica dos diagramas da UML em XPDDL e também em PDDL, representando como o itSIMPLE extrai as informações necessárias.

4.5.1.1. Tipos - (:types)

Os Tipos são facilmente extraídos das classes representadas no *Diagrama de Classes* de um projeto, onde cada classe equivale a um tipo em PDDL. São consideradas também as hierárquicas de classes, ou seja, as generalizações de classes. A extração dos tipos representa o passo 2 do pseudocódigo da Figura 72.

A tabela a seguir demonstra o mapeamento entre as características do *Diagrama de Classes* e os elementos Tipos na XPDDL e na PDDL.

Tabela 8 - Mapeamento entre classes do Diagrama de Classes e os Tipos na XPDDL e na PDDL.

Característica UML	XPDDL	PDDL
(1) Uma classe *MyClass* qualquer. Por exemplo, a classe *Pacote*. 	<types> <type name="MyClass" parent="object"/> ... </types>	(:types MyClass - object …)
(2) Uma classe *MyClass* qualquer que generalize outra ou outras classes *Class0*, *Class1*, ..., *ClassN*. Por exemplo, a classe Veículo generaliza as classes *Caminhão* e *Avião*. 	<types> <type name="MyClass" parent="object"/> <type name="Class0" parent="MyClass"/> <type name="Class1" parent="MyClass"/> ... <type name="ClassN" parent="MyClass"/> ... </types>	(:types Class0 Class ... ClassN - MyClass …)

Observação: neste trabalho considera-se que toda classe é derivada de uma superclasse chamada "object" como o que ocorre em Java™, por exemplo.

É importante citar que classes com *stereotype* defino como <<*utility*>> não são representadas com tipos, ou seja, elas não aparecem em `:types` (<types>).

No exemplo do domínio *Logística* modelado no Capítulo 3, e elemento `:types` na PDDL seria preenchido da seguinte forma:

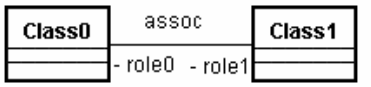
```
(:types
   Veiculo - object
   Caminhao - Veiculo
   Aviao - Veiculo
   Lugar - object
   Local - Lugar
   Aeroporto - Lugar
   Pacote - object
   Cidade - object
 )
```

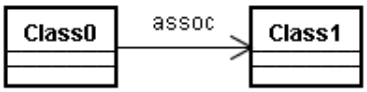
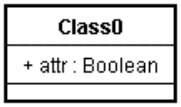
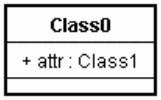
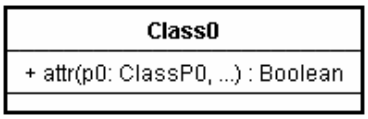
Vale lembrar que é necessário o requisito `:typing` em `:requirements` para que o elemento `:types` da PDDL seja utilizado.

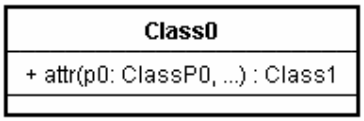
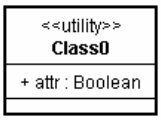
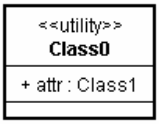
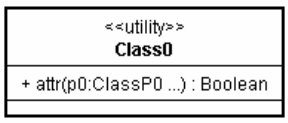
4.5.1.2. Predicados - (:predicates)

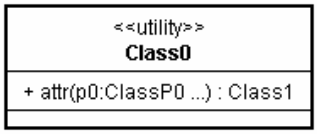
As associações e os atributos (parametrizados ou não (D'SOUZA; WILLS, 1999)) booleanos (atributos primitivos do tipo Boolean) e os não-primitivos, presentes no *Diagrama de Classes*, são representados na forma de Predicados na PDDL. Nas associações, são levados em consideração principalmente os *rolenames* e, se necessário, também as navegações. A extração dos predicados representa o passo 3 do pseudocódigo da Figura 72. A Tabela 9 demonstra como as associações e atributos do *Diagrama de Classes* da UML são representados tanto em XPDDL quanto em PDDL.

Tabela 9 - Mapeamento entre associações e atributos do *Diagrama de Classes* e os predicados na XPDDL e na PDDL.

Característica UML	XPDDL	PDDL
(1) Uma associação *assoc* entre as classes quaisquer *Class0* e *Class1*, com apenas um rolename *role* defino. Por exemplo, a associação *estaEm* entre as classes *Pacote* e Lugar com rolename *estaEm*. 	<predicates> <predicate name="role"> <parameter name="cla0" type="Class0"/> <parameter name="cla1" type="Class1"/> </predicate> ... </predicates>	(:predicates (role cla0 – Class0 cla1 – Class1) …)
(2) Uma associação *assoc* entre as classes quaisquer *Class0* e *Class1*, com apenas dois rolename *role0* e *role1* definidos. 	<predicates> <predicate name="role1"> <parameter name="cla0" type="Class0"/> <parameter name="cla1" type="Class1"/> </predicate> <predicate name="role0"> <parameter name="cla1" type="Class1"/> <parameter name="cla0" type="Class0"/> </predicate> ... </predicates>	(:predicates (role1 cla0 – Class0 cla1 – Class1) (role0 cla1 – Class1 cla0 – Class0) …)

(3) Uma associação *assoc* entre as classes quaisquer *Class0* e *Class1*, sem definição de rolenames e com restrições na navegação. 	<predicates> <predicate name="assoc"> <parameter name="cla0" type="Class0"/> <parameter name="cla1" type="Class1"/> </predicate> ... </predicates>	(:predicates (assoc cla0 – Class0 cla1 – Class1) …)
(4) Um atributo qualquer *attr* booleano (tipo primitivo Boolean) da classe *Class0*. Por exemplo, um atributo *lacrado* da classe *Pacote*. 	<predicates> <predicate name="attr"> <parameter name="cla" type="Class0"/> </predicate> ... </predicates>	(:predicates (attr cla – Class0) …)
(5) Um atributo qualquer *attr* não-primitivo da classe *Class0* do tipo *Class1*. Por exemplo, um atributo *fabricadoEm* da classe *Pacote* do tipo *Cidade*. 	<predicates> <predicate name="attr"> <parameter name="cla0" type="Class0"/> <parameter name="cla1" type="Class1"/> </predicate> ... </predicates>	(:predicates (attr cla0 – Class0 cla1 – Class1) …)
(6) Um atributo parametrizado qualquer *attr* booleano (tipo primitivo Boolean) da classe *Class0*, com parâmetros quaisquer (*p0: ClassP0, p1: ClassP1, ... pn: ClassPn*). O atributo *attr* deve ter pelo menos um parâmetro para ser considerado um atributo parametrizado. Por exemplo, um atributo *conectada(c:Cidade):Boolean* da classe *Cidade*. 	<predicates> <predicate name="attr"> <parameter name="cla0" type="Class0"/> <parameter name="p0" type="ClassP0"/> <parameter name="p1" type="ClassP1"/> ... <parameter name="pn" type="ClassPn"/> </predicate> ... </predicates>	(:predicates (attr cla0 – Class0 p0 – ClassP0 p1 – ClassP1 ... pn - ClassPn) …)

<table>
<tr>
<td>(7) Um atributo parametrizado qualquer attr não-primitivo da classe Class0 do tipo Class1, com parâmetros quaisquer (p0: ClassP0, p1: ClassP1, ... pn: ClassPn). Por exemplo, um atributo conectadaCom(c:Cidade):Estrada da classe Cidade.
 
</td>
<td><predicates>
<predicate name="attr">
<parameter name=″cla0″
type="Class0"/>
<parameter name=″p0″
type="ClassP0"/>
<parameter name=″p1″
type="ClassP1"/>
...
<parameter name=″pn″
type="ClassPn"/>
<parameter name=″cla1″
type="Class1"/>
</predicate>
...
</predicates></td>
<td>(:predicates
(attr cla0 – Class0
p0 – ClassP0
p1 – ClassP1
...
pn – ClassPn
cla1 – Class1)
…
)</td>
</tr>
<tr>
<td>(8) Um atributo qualquer <i>attr</i> booleano (tipo primitivo Boolean) da classe <i>Class0</i> com <i>stereotype</i> definido como <<<i>utility</i>>>.
 
</td>
<td><predicates>
<predicate name="attr">
</predicate>
...
</predicates></td>
<td>(:predicates
(attr)
…
)</td>
</tr>
<tr>
<td>(9) Um atributo qualquer <i>attr</i> não-primitivo, do tipo <i>Class1</i>, da classe <i>Class0</i> com <i>stereotype</i> definido como <<<i>utility</i>>>.
 
</td>
<td><predicates>
<predicate name="attr">
<parameter name=″cla1″
type="Class1"/>
</predicate>
...
</predicates></td>
<td>(:predicates
(attr cla1 – Class1)
…
)</td>
</tr>
<tr>
<td>(10) Um atributo parametrizado qualquer <i>attr</i> booleano (tipo primitivo Boolean) da classe <i>Class0</i> (com <i>stereotype</i> definido como <<<i>utility</i>>>), com parâmetros quaisquer (<i>p0: ClassP0, p1: ClassP1, ... pn: ClassPn</i>). O atributo <i>attr</i> deve ter pelo menos um parâmetro para ser considerado um atributo parametrizado. Por exemplo, um atributo <i>conectada(c1:Cidade, c2:Cidade):Boolean</i> de uma classe com stereotype <<<i>utility</i>>>.
 
</td>
<td><predicates>
<predicate name="attr">
<parameter name=″p0″
type="ClassP0"/>
<parameter name=″p1″
type="ClassP1"/>
...
<parameter name=″pn″
type="ClassPn"/>
</predicate>
...
</predicates></td>
<td>(:predicates
(attr p0 – ClassP0
p1 – ClassP1
...
pn - ClassPn)
…
)</td>
</tr>
</table>

(11) Um atributo parametrizado qualquer *attr* não-primitivo da classe *Class0* do tipo *Class1*, com parâmetros quaisquer (*p0: ClassP0, p1: ClassP1, ... pn: ClassPn*). Por exemplo, um atributo *conectadaCom(c1:Cidade, c2:Cidade):Estrada* de uma classe com stereotype <<*utility*>>.  	<predicates> <predicate name="attr"> <parameter name="p0" type="ClassP0"/> <parameter name="p1" type="ClassP1"/> ... <parameter name="pn" type="ClassPn"/> <parameter name="cla1" type="Class1"/> </predicate> ... </predicates>	(:predicates (attr p0 – ClassP0 p1 – ClassP1 ... pn – ClassPn cla1 – Class1) …)

Observação: Casos com auto-associação são tratados da mesma maneira que as associações apresentadas nesta tabela. Outros casos não contidos nesta tabela não são tratados.

No exemplo do domínio *Logística* modelado no Capítulo 3, e elemento `:predicates` na PDDL seria preenchido da seguinte forma:

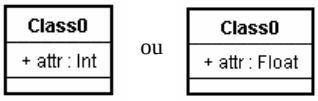
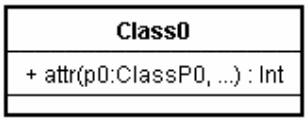
```
(:predicates
   (estaDentro ?pac - Pacote ?vei - Veiculo)
   (estaEm ?pac - Pacote ?lug - Lugar)
   (pertence ?lug - Lugar ?cid - Cidade)
   (estaNo ?cam - Caminhao ?lug - Lugar)
   (estaParadoNo ?avi - Aviao ?aer - Aeroporto)
 )
```

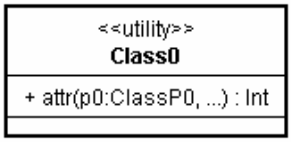
Vale citar que predicados que não possuam parâmetros (por exemplo, `(attr)`) exigem o requisito `:strips` em `:requirements` na PDDL.

4.5.1.3. Funções - (:functions)

Atributos primitivos (parametrizados ou não) do tipo Inteiro e Float (Int e Float) das classes no *Diagrama de Classes* do projeto são representados na forma de funções (*functions*) na PDDL (presentes a partir da versão PDDL2.1). A extração dos predicados representa o passo 4 do pseudocódigo da Figura 72. A Tabela 10 demonstra como os atributos (do tipo Int ou Float) do *Diagrama de Classes* da UML são representados tanto em XPDDL quanto em PDDL.

Tabela 10 - Mapeamento entre os atributos (do tipo Int ou Float) do *Diagrama de Classes* e funções (*functions*) na XPDDL e na PDDL.

Característica UML	XPDDL	PDDL
(1) Um atributo qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float) da classe *Class0*. Por exemplo, o atributo *capacidade* da classe *Veículo*. 	<functions> <function name="attr"> <parameter name=”cla” type="Class0"/> </ function> … </functions>	(:functions (attr cla – Class0) …)
(2) Um atributo parametrizado qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float) da classe *Class0,* com parâmetros quaisquer (*p0: ClassP0, p1: ClassP1, ... pn: ClassPn*). O atributo *attr* deve ter pelo menos um parâmetro para ser considerado um atributo parametrizado. Por exemplo, um atributo *distancia(c:Cidade):Float* da classe *Cidade.* 	<functions > <function name="attr"> <parameter name=”cla0” type="Class0"/> <parameter name=”p0” type="ClassP0"/> <parameter name=”p1” type="ClassP1"/> … <parameter name=”pn” type="ClassPn"/> </function> … </functions>	(:functions (attr cla0 – Class0 p0 – ClassP0 p1 – ClassP1 ... pn - ClassPn) …)
(3) Um atributo qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float) da classe *Class0* com *stereotype* definido como <<*utility*>>. Por exemplo, um atributo *combustivelTotalGasto: Int* de uma classe com *stereotype* <<*utility*>>. 	<functions> <function name="attr" /> … </functions>	(:functions (attr) …)

(4) Um atributo parametrizado qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float) da classe *Class0* (com *stereotype* definido como <<*utility*>>), com parâmetros quaisquer (*p0: ClassP0, p1: ClassP1, ... pn: ClassPn*). Por exemplo, um atributo *distancia(c1:Cidade, c2:Cidade):Int* de uma classe com stereotype <<*utility*>>.  	<functions> <function name="attr"> <parameter name="p0" type="ClassP0"/> <parameter name="p1" type="ClassP1"/> ... <parameter name="pn" type="ClassPn"/> </function > ... </functions>	(:functions (attr p0 – ClassP0 p1 – ClassP1 ... pn - ClassPn) ...)

Observação: Outros casos não contidos nesta tabela não são tratados.

No exemplo do domínio *Logística* modelado no Capítulo 3, e elemento `:functions` na PDDL seria preenchido da seguinte forma:

```
(:functions
  (capacidade ?vei - Veiculo)
  (carregamentoCorrente ?vei - Veiculo)
)
```

É necessário o requisito `:fluents` em `:requirements` para que as funções (`:functions`) sejam utilizadas no modelo em PDDL. Esta característica só é encontrada em versões superiores ou igual a PDDL2.1 (FOX; LONG, 2003).

4.5.1.4. Restrições - (:constraints)

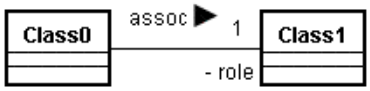
Caso o projetista deseje trabalhar com planejadores capazes de lidar com a PDDL3.0 (GEREVINI; LONG, 2005), a versão do itSIMPLE, apresentada no presente trabalho, trata somente as restrições provenientes das multiplicidades das associações presentes nos *Diagramas de Classes.* São tratadas apenas as associações com multiplicidades definidas com "1" ou "0..1", pois estas possuem um mapeamento claro na PDDL. A extração das restrições representa o passo 5 do pseudocódigo da Figura 72. A Tabela 11 demonstra uma forma de

como as multiplicidades, apenas nos casos “1” e “0..1”, das associações do *Diagrama de Classes* da UML podem ser representadas tanto em XPDDL quanto em PDDL.

Tabela 11 - Mapeamento entre multiplicidades “1” e “0..1” de associações no *Diagrama de Classes* e restrições (*constraints*) na XPDDL e na PDDL.

Característica UML

Uma associação *assoc* entre as classes quaisquer *Class0* e *Class1*, com apenas um rolename *role* defino onde a multiplicidade é “1” ou “0..1”. Por exemplo, a associação *estaEm* entre as classes *Pacote* e Lugar com rolename *estaEm*.



XPDDL

```
<constraints>
  ...
  <forall>
        <parameter name="cla11" type="Class1" />
        <parameter name="cla12" type="Class1" />
        <parameter name="cla0" type="Class0" />
        <always>
                <imply>
                        <and>
                                <predicate id="role">
                                        <parameter id="cla0" />
                                        <parameter id="cla11" />
                                </predicate>
                                <predicate id="role">
                                        <parameter id="cla0" />
                                        <parameter id="cla12" />
                                </predicate>
                        </and>
                        <implies>
                                <equals>
                                        <parameter id="cla11" />
                                        <parameter id="cla12" />
                                </equals>
                        </implies>
                </imply>
        </always>
  </forall>
  ...
</constraints>
```

PDDL

```
(:constraints
    ...
    (forall (?cla11 – Class1 ?cla12 – Class1 ?cla0 – Class0)
        (always
            (imply (and (role ?cla0 ?cla11) (role ?cla0 ?cla12))(= ?cla11 ?cla12))
         )
     )
    ...
)
```

Um processo mais elaborado de tradução de restrições é deixado para trabalhos futuros já que o uso das restrições (`:constraints`) não está no escopo principal deste trabalho. Mesmo não sendo o foco deste trabalho, no exemplo do domínio *Logística* modelado no Capítulo 3, e elemento `:constraints` na PDDL seria preenchido da seguinte forma:

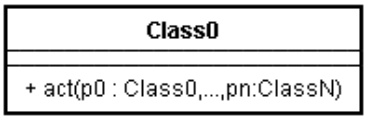
```
(:constraints
  (and
    (forall (?vei1 - Veiculo ?vei2 - Veiculo ?pac - Pacote)
      (always
        (imply (and (estaDentro ?pac ?vei1) (estaDentro ?pac ?vei2))(= ?vei1 ?vei2))
      )
    )
    (forall (?lug1 - Lugar ?lug2 - Lugar ?pac - Pacote)
      (always
        (imply (and (estaEm ?pac ?lug1) (estaEm ?pac ?lug2))(= ?lug1 ?lug2))
      )
    )
    (forall (?cid1 - Cidade ?cid2 - Cidade ?lug - Lugar)
      (always
        (imply (and (pertence ?lug ?cid1) (pertence ?lug ?cid2))(= ?cid1 ?cid2))
      )
    )
    (forall (?lug1 - Lugar ?lug2 - Lugar ?cam - Caminhao)
      (always
        (imply (and (estaNo ?cam ?lug1) (estaNo ?cam ?lug2))(= ?lug1 ?lug2))
      )
    )
    (forall (?aer1 - Aeroporto ?aer2 - Aeroporto ?avi - Aviao)
      (always
        (imply (and (estaParadoNo ?avi ?aer1)(estaParadoNo ?avi ?aer2))(= ?aer1 ?aer2))
      )
    )
  )
)
```

Vale citar que o uso de restrições no modelo em PDDL exige o requisito `:constraints` em `:requirements`. Devido ao fato das restrições tratadas neste trabalho exigirem o uso do termo `forall`, é necessário também o requisito `:quantified-preconditions` em `:requirements`.

4.5.1.5. Ações - (:action)

Conforme visto na Figura 71, uma ação em PDDL é composta basicamente por quatro elementos principais, sendo eles: o *nome* da ação; os *parâmetros*; as *pré-condições*; e as *pós-condições* da ação. Para a formação das ações no modelo XPDDL, e conseqüentemente em PDDL, o itSIMPLE extrai informações provenientes de dois diagramas, o *Diagrama de Classes* e o *Diagrama de Estados*. A ferramenta identifica os nomes das ações e seus parâmetros através do *Diagrama de Classes* onde estas características são declaradas nas classes. Já as pré e pós-condições são extraídas das expressões em OCL declaradas nos *Diagramas de Estados*, conforme apresentado no Capítulo 3. Vale lembrar que as pré-condição de uma ação é a união das expressões em OCL definidas nos estados antecessores e das expressões definas nas transições que representam a ação nos diagramas, o que acontesse similarmente para a pós-condição. A representação inicial das ações em XPPDL representa o passo 6 no processo de tradução do modelo do domínio ilustrado no pseudocódigo da Figura 72. Assim, primeiramente, para cada ação encontrada no *Diagrama de Classes*, o itSIMPLE cria a estrutura básica de um ação em XPDDL, contendo apenas seu nome e seus parâmetros. A Tabela 12 ilustra a representação desta estrutura inicial de uma ação, criada a partir das ações encontradas no *Diagrama de Classes*, em XPDDL, bem como em PDDL.

Tabela 12 - Mapeamento entre as ações encontradas do *Diagrama de Classes* e ações (*actions*) na XPDDL e na PDDL (apenas os nomes e os parâmetros).

Característica UML	**XPDDL**	**PDDL**
Uma ação qualquer *act*, realizada pela classe *Class0*, com parâmetros *p0: Class0, p1: Class1, ... , pn: ClassN*. Por exemplo, a ação *pilotar(avi:Avião, de:Aeroporto, para: Aeroporto)* realizada pela classe *Avião*. 	<actions> <action name="act"> <parameter name="p0" type="Class0"/> <parameter name="p1" type="Class1"/> ... <parameter name="pn" type="ClassN"/> <precondition /> <effect /> </action> ... </actions>	(:action act (:parameters p0 – Class0 p1 – Class1 ... pn - ClassN) (:precondition) (:effect))

Depois de representado o nome da ação e seus respectivos parâmetros na ação em XPDDL, é necessária completar a representação desta com suas respectivas pré-condições (<precondition>, `:precondition`) e pós-condição (<effect>, `:effect`). Para tal, o itSIMPLE extrai primeiramente todas as expressões OCL dos *Diagramas de Estados* que representam as pré-condições de tal ação, conforme discutido no Capítulo 3. Esta extração das pré-condições forma uma representação única que será devidamente traduzida para XPDDL e PDDL conforme será visto a seguir. O mesmo é feito para as pós-condições, o que resulta em uma representação única para toda a pós-condição da ação em questão.

Tendo as duas representações em OCL, tanto da pré quanto da pós-condição, o itSIMPLE realiza o processo de tradução das expressões em OCL para XPDDL. A versão da ferramenta apresentada no presente trabalho possui algumas limitações quanto a traduções de expressões em OCL para PDDL, ou seja, apenas algumas expressões são interpretadas e traduzidas. Estas expressões são apresentadas na Tabela 13 com suas respectivas traduções em XPDDL e PDDL. Nesta tabela são utilizados os seguintes termos: *precond* é o conjunto de expressões em OCL que representa as pré-condições de uma ação qualquer *act*; e *poscond* é o conjunto de expressões em OCL que representa as pós-condições da ação *act.*

Tabela 13 - Mapeamento entre expressões OCL (pré e pós-condições) e expressões XPDDL e PDDL (*precondition* e *effect*).

Expressões OCL	**XPDDL**	**PDDL**
(1) Expressões em *precond* ou em *poscond* com operadores "*and*" da seguinte forma: `[expression1] and [expression2] and ... and [expressionN]`	<and> [expression1] [expression2] ... [expressionN] </and>	(and [expression1] [expression2] ... [expressionN])
(2) Expressões em *precond* ou em *poscond* com operadores "*or*"da seguinte forma: `[expression1] or [expression2] or ... or [expressionN]`	<or> [expression1] [expression2] ... [expressionN] </or>	(or [expression1] [expression2] ... [expressionN])

(3) Expressões em *precond* ou em *poscond* na forma: **not**([expression])	<not> [expression1] </not>	(not([expression]))
(4) Uma expressão [expression] em *precond* na forma: **p0 = p1** onde: p0 e p1 são parâmetros da ação *act*. O caso **p0 <> p1** é idêntico a not(p0 = p1).	<equals> <parameter id="p0"> <parameter id="p0"> </equals>	(= p0 p1)
(5) Uma expressão [expression] em *precond* ou em *poscond* na forma: **p0.attr = true** onde: p0 é um parâmetro da ação *act*; e attr é um atributo não parametrizado de p0 de tipo booleano.	<predicate id="attr"> <parameter id="p0"> </predicate>	(attr p0)
(6) Uma expressão [expression] em *precond* ou em *poscond* na forma: **p0.attr = false** onde: p0 é um parâmetro da ação *act*; e attr é um atributo não parametrizado de p0 de tipo booleano.	<not> <predicate id="attr"> <parameter id="p0"> <parameter id="p1"> </predicate> </not>	(not (attr p0))
(7) Uma expressão [expression] em *precond* ou em *poscond* na forma: **p0.attr(o1,...,on) = true** onde: p0 é um parâmetro da ação *act*; e attr é um atributo parametrizado de p0 de tipo booleano. Caso o valor seja **false**, a tradução é idêntica a not([expression]).	<predicate id="attr"> <parameter id="p0"> <parameter id="o1"> ... <parameter id="on"> </predicate>	(attr p0 o1 ... on)
(8) Uma expressão [expression] em *precond* ou em *poscond* na forma: **p0.attr = p1** onde: p0 e p1 são parâmetros da ação *act*; e attr é um atributo não parametrizado de p0 de tipo não-primitivo. Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a not([expression]).	<predicate id="attr"> <parameter id="p0"> <parameter id="p1"> </predicate>	(attr p0 p1)
(9) Uma expressão [expression] em *precond* ou em *poscond* na forma: **p0.attr(o1,...,on) = p1** onde: p0 e p1 são parâmetros da ação *act*; e attr é um atributo parametrizado de p0 de tipo não-primitivo. Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a not([expression]).	<predicate id="attr"> <parameter id="p0"> <parameter id="o1"> ... <parameter id="on"> <parameter id="p1"> </predicate>	(attr p0 o1 ... on p1)

(10) Uma expressão `[expression]` em *precond* ou em *poscond* na forma: **`p0.role = p1`** onde: `p0` e `p1` são parâmetros da ação *act*; e `role` é um *rolename* ,com multiplicidade "1" ou "0..1", de uma associação qualquer entre as classes de `p0` e `p1`. Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a `not([expression])`.	<predicate id="role"> <parameter id="p0"> <parameter id="p1"> </predicate>	`(attr p0 p1)`
(11) Uma expressão `[expression]` em *precond* na forma: **`p0.role->exists(p|p = p1)`** onde: `p0` e `p1` são parâmetros da ação *act*; e `role` é um *rolename*, com multiplicidade maior que 2 ou "*", de uma associação qualquer entre as classes de `p0` e `p1`. Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a `not([expression])`.	<predicate id="role"> <parameter id="p0"> <parameter id="p1"> </predicate>	`(attr p0 p1)`
(12) Uma expressão `[expression]` em *precond* ou em *poscond* na forma: **`attr = true`** onde: `attr` é um atributo não parametrizado booleano *global*. Caso o valor seja **`false`**, a tradução é idêntica a `not([expression])`.	<predicate id="attr" />	`(attr)`
(13) Uma expressão `[expression]` em *precond* ou em *poscond* na forma: **`attr(o1,...,on) = true`** onde: `attr` é um atributo parametrizado booleano *global*. Caso o valor seja **`false`**, a tradução é idêntica a `not([expression])`.	<predicate id="attr"> <parameter id="o1"> ... <parameter id="on"> </predicate>	`(attr o1 ... on)`
(14) Uma expressão `[expression]` em *precond* ou em *poscond* na forma: **`attr = p1`** onde: `attr` é um atributo não parametrizado *global* de tipo não-primitivo; e `p1` é um parâmetro da ação *act*. Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a `not([expression])`.	<predicate id="attr"> <parameter id="p1"> </predicate>	`(attr p1)`

(15) Uma expressão `[expression]` em *precond* ou em *poscond* na forma: **`attr(o1,...,on) = p1`** onde: `attr` é um atributo parametrizado *global* de tipo não-primitivo; e `p1` é um parâmetro da ação *act*; Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a `not([expression])`.	<predicate id="attr"> <parameter id="o1"> ... <parameter id="on"> <parameter id="p1"> </predicate>	`(attr o1 ... on p1)`
(16) Uma expressão `[expression]` em *precond* ou em *poscond* na forma: **`p0.role = p1`** onde: `p0` e `p1` são um parâmetros da ação *act*; e `role` é um rolename de uma associação qualquer entre as classes de `p0` e `p1`. Caso o operador não seja "=" e sim "**<>**", a tradução é idêntica a `not([expression])`.	<predicate id="role"> <parameter id="p0"> <parameter id="p1"> </predicate>	`(attr p0 p1)`
(17) Expressões igualadas a **`null`** tanto na *precond* quanto na *poscond* são ignoradas nesta versão do itSIMPLE. Por exemplo **`p0.role = null`**	Caso ignorado	`Caso ignorado`

(18) Expressões numéricas em *precond* devem ser traduzidas na forma prefixa. Um exemplo seria: **`p0.attr > p1.attr1 + num`** onde: `p0` e `p1` são parâmetros da ação *act*; `attr` e `attr1` são atributos numéricos e `num` é um número inteiro ou float.
Obs: nestas expressões podem ser utilizados operadores comparadores como: "=", ">=", "<=" e "<>" (ou seja, not("=")). É também possível utilizar operadores como "+", "-", "/" e "*" nas expressões.
	Exemplo:
	<gt>
	<function id="attr">
	<parameter id="p0">
	</function>
	<add>
	<function id="attr1">
	<parameter id="p1">
	</function>
	<value number="num"/>
	</add>
	</gt>
		`Exemplo:`
		`(> (attr p0)`
		`(+ (attr1 p1) num)`
		`)`

(19) Expressões numéricas em *poscond* devem ser traduzidas na forma prefixa onde o operador "=" indica atribuição. Um exemplo seria: **`p0.attr = p1.attr1 + num`** onde: `p0` e `p1` são parâmetros da ação *act*; `attr` e `attr1` são atributos numéricos e `num` é um número inteiro ou float.	Exemplo:
	<assign>
	<function id="attr">
	<parameter id="p0">
	</function>
	<add>
	<function id="attr1">
	<parameter id="p1">
	</function>
	<value number="num"/>
	</add>
	</assign >
		`Exemplo:`
		`(= (attr p0)`
		`(+ (attr1 p1) num)`
		`)`

(20) Uma expressão numérica em *poscond* na forma: **`p0.attr = p0.attr + num`** onde: `p0` e `p1` são parâmetros da ação *act*; `attr` é um atributo numérico e `num` é um número inteiro ou float.	Exemplo:
	<increase>
	<function id="attr">
	<parameter id="p0">
	</function>
	<value number="num" />
	</increase >
		`Exemplo:`
		`(increase (attr p0)`
		`num)`

(21) Uma expressão numérica em *poscond* na forma: **`p0.attr = p0.attr - num`** onde: `p0` é um parâmetro da ação *act*; `attr` é um atributo numérico e `num` é um número inteiro ou float.	Exemplo:
	<decrease>
	<function id="attr">
	<parameter id="p0">
	</function>
	<value number="num" />
	</decrease >
		`Exemplo:`
		`(decrease (attr p0)`
		`num)`

Observação: Outros casos não contidos nesta tabela não são tratados.

Mesmo a utilização da OCL sendo restrita conforme mostra a tabela anterior, é possível modelar muitos domínios de planejamento clássicos e reais conforme será visto no próximo capítulo.

Mesmo depois de traduzidas as expressões, tanto das pré-condições quanto das pós-condições, são necessários ainda alguns refinamentos na pós-condição. Estes refinamentos representam o passo 7 no processo de tradução ilustrado no pseudocódigo da Figura 72. Os refinamentos visam adicionar predicados negados na pós-condição da ação (`:effect`) para os casos onde um predicado aparece na pré-condição e este é alterado na pós-condição. Para melhor entender a necessidade destes refinamentos, o exemplo mostrado na Tabela 14 a seguir ilustra um caso de refinamento. Neste exemplo uma ação *act* em OCL é traduzida para PDDL conforme a Tabela 13.

Tabela 14 - Exemplo de tradução sem refinamento.

Representação OCL	**Representação PDDL**

`act(…)`
`pre:`
`p0.attr = p1`
`post:`
`p0.attr = p2`
	`(action act`
	`:parameters …`
	`:precondition (attr p0 p1)`
	`:effect (attr p0 p2)`
	`)`

É possível perceber na Tabela 14 que é necessário adicionar o termo (`not(attr p0 p2)`) na pós-condição (`:effect`) da ação *act* para que a tradução seja correta. Assim, com o refinamento do exemplo da Tabela 14 o resultado da tradução para PDDL da ação *act* seria o seguinte:

```
(action act
  :parameters …
  :precondition
              (attr p0 p1)
  :effect
              (not (attr p0 p1))(attr p0 p2)
)
```

Estes casos onde há a necessidade de refinamentos ocorrem exclusivamente em expressões com atributos não-primitivos (parametrizados ou não) ou com *rolenames* de associações com multiplicidade “1” ou “0..1”. Quando esses atributos e associações fazem parte da pré-condição de uma ação e são modificados (atribui-se novos valores, ou até mesmo valores nulos - null) na pós-condição são necessárias as adições das negações dos mesmos, conforme exemplificado anteriormente. Para exemplificar o processo completo de tradução das ações em PDDL realizado pelo itSIMPLE, segue abaixo a ação *carregarCaminhão* do modelo do domínio *Logística* representada em OCL e também em PDDL (resultante da tradução).

```
context Caminhão:: carregaCaminhao(cam:Caminhão, pkg:Pacote, loc:Lugar)
 pre:
        -- contexto do Avião
        -- O Caminhão deve estar em um lugar (loc) e sua capacidade deve ser
        --  maior que seu carregamento corrente para que um pacote seja
        --  carregado
        cam.estaNo <> null and cam.estaNo = loc and
        cam.capacidade > cam.carregamentoCorrente and

        -- contexto do Pacote
        -- o Pacote deve também estar no mesmo lugar
        pkg.estaEm = loc and pkg.estaDentro = null

 post:
        -- contexto do Caminhão
        cam.carregamentoCorrente = cam.carregamentoCorrente + 1 and

        -- contexto do Pacote
        pkg.estaEm = null and pkg.estaDentro = avi
```

```
(:action carregaCaminhao
   :parameters (?cam - Caminhao ?pkg - Pacote ?loc - Lugar)
   :precondition
     (and
       (estaNo ?cam ?loc)
       (> (capacidade ?cam) (carregamentoCorrente ?cam))
       (estaEm ?pkg ?loc)
     )
   :effect
     (and
       (increase (carregamentoCorrente ?cam) 1)
       (not (estaEm ?pkg ?loc))
       (estaDentro ?pkg ?cam)
     )
)
```

É possível visualizar no exemplo acima algumas das traduções realizadas pelo itSIMPLE, seguindo as regras da Tabela 13, bem como alguns refinamentos. Um exemplo de refinamento neste caso seria a adição do termo `(not (estaEm ?pkg ?loc))` na pós-condição da ação *carregaCaminhao*.

4.5.1.6. Requisitos - (:requirements)

Baseado nos características citadas acima do modelo gerado em XPDDL o itSIMPLE preenche os requisitos no elemento :requirements conforme ilustrado no passo 8 (último passo) do pseudocódigo na Figura 72. Como foi visto nos itens anteriores algumas características exigem certos requisitos, e assim o itSIMPLE consegue mapear no modelo (baseado nas definições da PDDL (MCDERMOTT, 1998) (FOX; LONG, 2003) (GEREVINI; LONG, 2005)) quais delas estão presentes no mesmo.

No exemplo do domínio *Logística* modelado no Capítulo 3, e elemento `:requirements` na PDDL seria preenchido da seguinte forma:

```
(:requirements :typing :fluents :negative-preconditions
               :quantified-preconditions :constraints)
```

4.5.2. Tradução do Problema de Planejamento

Conforme visto no Capítulo 2, um problema em PDDL (MCDERMOTT, 1998) (ou em XPDDL(GOUGH, 2004)) possui também uma estrutura bem definida. Os principais elementos desta estrutura tratados no itSIMPLE são demonstrados na Figura 73.

PDDL

```
(define (problem {nome_prob})
      (domain {nome-dom})

      (:objects … )

      (:init … )

      (:goal … )
)
```

XPDDL

```
<xpddl:problem … >
      <name>{nome_prob}</name>
      <domain>{nome-dom}</domain>
      <objects> …
      </objects>
      <init> …
      </init>
      <goal> …
      </goal>
</xpddl:problem>
```

Figura 73 – Elementos principais de um problema em PDDL tratados no itSIMPLE.

Tomando como referência a estrutura de um problema em PDDL, o itSIMPLE realiza um procedimento que passo a passo colhe as informações provenientes na *Representação Core* (principalmente dos *Diagramas de Objetos*) e preenche cada elemento principal da XPDDL do problema (`:objects`, `:init` e `:goal`). Depois de representado o modelo em XPDDL o itSIMPLE fornece ao usuário o modelo em PDDL através de um procedimento simples que pode ser entendido em (GOUGH, 2004). Para ilustrar o procedimento de tradução para XPDDL, a Figura 74 representa o pseudocódigo do processo de tradução do problema para XPDDL. Este pseudocódigo demonstra que dado um problema de um determinado domínio (e

seu respectivo projeto no itSIMPLE), o mesmo retorna a representação completa em XPDDL do problema fornecido.

```
XMLToXPDDLProblem(problema, domínio, projeto)
  Início
     xpddlproblem ← estrutura vazia de um problema em XPDDL

     //1. nomea o problema
     adiciona o nome de problema em xpddlproblem
     //2. nomea o domínio
     adiciona o nome do domínio  em xpddlproblem
     //3.cria os objetos em <objects>
     para cada objeto existente no repositório de objetos do domínio faça
       adiciona um objeto no nó de objetos <objects> de xpddlproblem
     fimpara
     //4. cria o estado inicial <init> e o estado objetivo <goal> do problema
     para cada diagrama de objetos em problema faça
          diagrama ← o diagrama corrente
          avalia (diagrama)
            //4.1 cria estado inicial
            caso diagrama seja da cena inicial (sequenceReference = init):
                cria a cena inicial em <initi> da xpddlproblem
            fimcaso

            //4.2 cria estado objetivo
            caso diagrama seja da cena final (sequenceReference = goal):
                cria a cena final em <goal> da xpddlproblem
            fimcaso
          fimavalia
     fimpara

     retorna xpddlproblem

  Fim
```

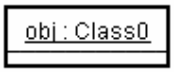
Figura 74 – Pseudocódigo da tradução do problema para XPDDL.

Conforme visto na Figura 74, o pseudocódigo de tradução para XPDDL possui quatro passos, sendo o 3 (criação dos objetos) e o 4 (criação dos estados inicial e objetivo) os mais relevantes. A seguir são discutidos os pontos mais relevantes do procedimento *XMLtoXPDDLProblem*. Durante as explicações a seguir são apresentados novamente os mapeamentos de cada característica dos diagramas em XPDDL e em PDDL, representando como o itSIMPLE extrai as informações necessárias.

4.5.2.1. Objetos – (:objects)

Os objetos do problema são os Agentes e Recursos do domínio representados no *Repositório* (*Diagrama de Objetos* do domínio). O itSIMPLE considera a classe de cada objeto (Agentes Recursos) para representar corretamente os objetos do problema em PDDL. O passo 3 do procedimento *XMLtoXPDDLProblem* (Figura 74) representa a criação dos objetos no problema em XPDDL. A Tabela 15 demonstra o mapeamento entre os objetos do diagrama *Repositório* e os objetos na XPDDL e na PDDL.

Tabela 15 - Mapeamento entre os objetos (do tipo Int ou Float) do *Diagrama de Objetos (Repositório)* e os objetos (*objects*) na XPDDL e na PDDL.

Característica UML	**XPDDL**	**PDDL**
Um objeto qualquer *obj* da classe *Class0*. Por exemplo, o objeto *caminhao1* da classe *Caminhão*.  	<objects> <object name="obj" type="Class0" /> ... </objects>	(:objects (obj – Class0) …)

No exemplo do domínio *Logística* e do problema de planejamento apresentados no Capítulo 3, os objetos seriam representados em PDDL da seguinte forma:

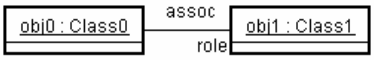
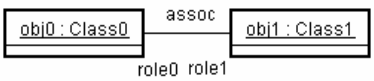
```
(:objects
  caminhao1 - Caminhao
  caminhao2 - Caminhao
  localSP1 - Local
  aviao1 - Aviao
  aviao2 - Aviao
  localRJ1 - Local
  aeroportoSP - Aeroporto
  aeroportoRJ1 - Aeroporto
  saopaulo - Cidade
  riodejaneiro - Cidade
  pkg1 - Pacote
  pkg2 - Pacote
  localSP2 - Local
  localRJ2 - Local
  pkg3 - Pacote
  pkg4 - Pacote
)
```

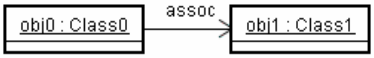
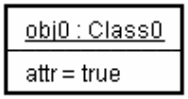
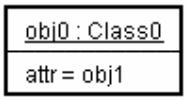
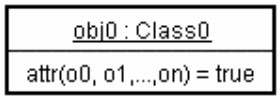
4.5.2.2. Estado Inicial – (:init)

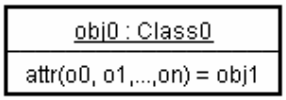
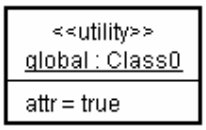
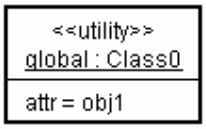
Todos os atributos, com seu respectivos valores, e associações entre objetos (Agentes e Recursos) encontrados no *Snapshot Inicial* do problema são declarados e representados no elemento `:init` da PDDL. Estes valores dos atributos e as associações definem o estado inicial do problema de planejamento.

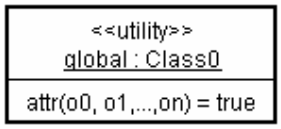
Conforme visto no procedimento *XMLtoXPDDLProblem*, o passo 4.1 representa o processo que o itSIMPLE realiza para formar o estado inicial do problema em XPDDL. A Tabela 16 demonstra como as associações e atributos dos agentes e recursos instanciados no *Snapshot* inicial são representados tanto em XPDDL quanto em PDDL.

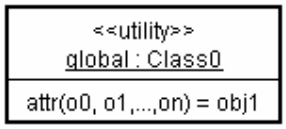
Tabela 16 - Mapeamento entre associações e atributos dos objetos do *Snapshot* e suas respectivas representações na XPDDL e na PDDL.

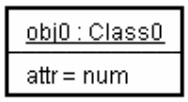
Característica UML	XPDDL	PDDL
(1) Uma associação *assoc* entre dois objetos quaisquer *obj0* e *obj1* das classes *Class0* e *Class1*, respectivamente. A associação *assoc* possui apenas um rolename *role* defino. Por exemplo, a associação *estaEm* entre os objetos *pkg1* e *localSP2* com rolename *estaEm*. 	<init> <predicate id="role"> <parameter object="obj0" /> <parameter object="obj1" /> </predicate> … </init>	(:init (role obj0 obj1) …)
(2) Uma associação *assoc* entre dois objetos quaisquer *obj0* e *obj1* das classes *Class0* e *Class1*, respectivamente. A associação *assoc* possui dois rolename *role0* e *role0* definidos. 	<init> <predicate id="role1"> <parameter object="obj0" /> <parameter object="obj1" /> </predicate> <predicate id="role0"> <parameter object="obj1" /> <parameter object="obj0" /> </predicate> … </init>	(:init (role1 obj0 obj1) (role0 obj1 obj0) …)

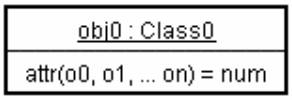
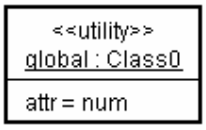
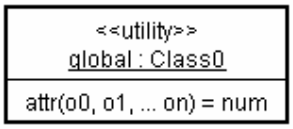
(3) Uma associação *assoc* entre dois objetos quaisquer *obj0* e *obj1* das classes *Class0* e *Class1*, respectivamente. A associação *assoc* não possui definição de rolenames, mas possui restrições na navegação. 	<init> <predicate id="assoc"> <parameter object=”obj0” /> <parameter object=”obj1” /> </predicate> … </init>	(:init (assoc obj0 obj1) …)
(4) Um atributo qualquer *attr* booleano (tipo primitivo Boolean) de um objeto *obj0*, da classe *Class0*, com valor *true*. Por exemplo, um atributo *lacrado = true* de um objeto *pkg1*. Obs: atributos deste tipo com valor *false* são ignorados na PDDL. 	<init> <predicate id="attr"> <parameter object=”obj0” /> </predicate> … </init>	(:init (attr obj0) …)
(5) Um atributo qualquer *attr* não-primitivo do tipo *Class1* de um objeto *obj0*, da classe *Class0*, com valor *obj1*. Por exemplo, um atributo *fabricadoEm = saopaulo* de um objeto *pkg1* da classe *Pacote*. 	<init> <predicate id="attr"> <parameter object=”obj0” /> <parameter object=”obj1” /> </predicate> … </init>	(:init (attr obj0 obj1) …)
(6) Um atributo parametrizado qualquer *attr* booleano (tipo primitivo Boolean) de um objeto *obj0*, da classe *Class0*, com valor *true* e com parâmetros de valores (*o0, o1, ... on*). Por exemplo, um atributo *conectada(riodejaneiro) = true* de um objeto *saopaulo* da classe *Cidade*. Obs: atributos deste tipo com valor *false* são ignorados na PDDL. 	<init> <predicate id="attr"> <parameter object="obj0"/> <parameter object="o0"/> <parameter object="o1"/> … <parameter object="on"/> </predicate> … </init>	(:init (attr obj0 o0 o1 ... on) …)

(7) Um atributo parametrizado qualquer *attr* não-primitivo do tipo *Class1* de um objeto *obj0*, da classe *Class0*, com valor *obj1* e com parâmetros de valores (*o0*, *o1*, ... *on*). Por exemplo, um atributo *conectadaCom(riodejaneiro) = estradaSPRJ* de um objeto *sãopaulo* da classe *Cidade*. 	<init> <predicate id="attr"> <parameter object="obj0"/> <parameter object="o0"/> <parameter object="o1"/> ... <parameter object="on"/> <parameter object="obj1"/> </predicate> ... </init>	(:init (attr obj0 o0 o1 ... on obj1) …)
(8) Um atributo qualquer *attr* booleano (tipo primitivo Boolean) de valor *true* de um objeto *global* da classe *Class0* com *stereotype* definido como <<*utility*>>. Obs: atributos deste tipo com valor *false* são ignorados na PDDL. 	<init> <predicate id="attr"> </predicate> ... </init>	(:init (attr) …)
(9) Um atributo qualquer *attr* não-primitivo, do tipo *Class1*, de valor *obj1* de um objeto *global* da classe *Class0* com *stereotype* definido como <<*utility*>>. 	<init> <predicate id="attr"> <parameter object="obj1"/> </predicate> ... </init>	(:init (attr obj1) …)

(10) Um atributo parametrizado qualquer *attr* booleano (tipo primitivo Boolean), com valor *true*, de um objeto *global* da classe *Class0* (com *stereotype* definido como <<*utility*>>), com parâmetros de valores (*o0, o1, ... on*). Por exemplo, um atributo *conectada(saopaulo, riodejaneiro) = true* de um objeto *global* de uma classe com stereotype <<*utility*>>. Obs: atributos deste tipo com valor *false* são ignorados na PDDL.
 
 	
	<init>
	  <predicate id="attr">
	  <parameter
	  object="o0"/>
	  <parameter
	  object="o1"/>
	  ...
	  <parameter
	  object="on"/>
	  </predicate>
	  ...
	</init>
		(:init
		  (attr o0 o1 ...
		  on)
		  …
		)

(11) Um atributo parametrizado qualquer *attr* não-primitivo do tipo *Class1*, com valor *obj1*, de um objeto *global* da classe *Class0*, com parâmetros de valores (*o0, o1, ... on*). Por exemplo, um atributo *conectadaCom(sãopaulo, riodejaneiro) = estradaSPRJ* de um objeto *global* de uma classe com stereotype <<*utility*>>.
 
 	
	<init>
	  <predicate id="attr">
	  <parameter
	  object="o0"/>
	  <parameter
	  object="o1"/>
	  ...
	  <parameter
	  object="on"/>
	  <parameter
	  object="obj1"/>
	  </predicate>
	  ...
	</init>
		(:init
		  (attr o0 o1 ...
		  on obj1)
		  …
		)

(12) Um atributo qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float), com valor igual a *num*, de um objeto *obj0* da classe *Class0*. Por exemplo, o atributo *capacidade* = 10 do objeto *aviao1* da classe *Veículo*.
 
 	
	<init>
	  <equals>
	  <function id="attr">
	  <parameter
	  object="obj0" />
	  </function>
	  <value number="num" />
	  </equals>
	  ...
	</init>
		(:init
		  (= (attr obj0) num)
		  …
		)

(13) Um atributo parametrizado qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float), com valor igual a *num*, de um objeto *obj0* da classe *Class0*, com parâmetros de valores (*o0, o1, ... on*). Por exemplo, um atributo *distancia(riodejaneiro) = 500* de um objeto *saopaulo* da classe *Cidade*. 	<init> <equals> <function id="attr"> <parameter object="obj0" /> <parameter object="o0" /> <parameter object="o1" /> ... <parameter object="on" /> </function> <value number="num" /> </equals> ... </init>	(:init (= (attr obj0 o0 o1 ... on) num) …)
(14) Um atributo qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float), com valor igual a *num*, de um objeto *global* da classe *Class0* com *stereotype* definido como *<<utility>>*. Por exemplo, um atributo *combustivelTotalGasto = 0* de um objeto *global* de uma classe com stereotype *<<utility>>*. 	<init> <equals> <function id="attr"> </function> <value number="num" /> </equals> ... </init>	(:init (= (attr) num) …)
(15) Um atributo parametrizado qualquer *attr* do tipo Inteiro ou Float (tipos primitivos Int e Float), com valor igual a *num*, de um objeto *global* da classe *Class0* (com *stereotype* definido como *<<utility>>*), com parâmetros de valores (*o0, o1, ... on*). Por exemplo, um atributo *distancia(saopaulo,riodejaneiro) = 500* de um objeto *global* de uma classe com stereotype *<<utility>>*. 	<init> <equals> <function id="attr"> <parameter object="o0" /> <parameter object="o1" /> ... <parameter object="on" /> </function> <value number="num" /> </equals> ... </init>	(:init (= (attr o0 o1 ... on) num) …)

Observação: Outros casos não contidos nesta tabela não são tratados.

Como visto no Capítulo 3, é possível definir associações e valores de atributos no repositório do domínio (*Repositório*). Estas associações entre os agentes e recursos e seus valores de atributos também são representados no estado inicial, pois estes representam fatos imutáveis que também fazem parte deste estado. Assim, esses fatos (imutáveis) do *Repositório* são mapeados da mesma maneira daquela apresentada na Tabela 16.

No exemplo do domínio *Logística* e do problema de planejamento apresentados no Capítulo 3, o estado inicial representado em PDDL seria da seguinte forma:

```
(:init
  (pertence aeroportoSP saopaulo)
  (pertence aeroportoRJ1 riodejaneiro)
  (pertence localSP1 saopaulo)
  (pertence localRJ1 riodejaneiro)
  (pertence localSP2 saopaulo)
  (pertence localRJ2 riodejaneiro)
  (estaNo caminhao1 localSP1)
  (estaEm pkg1 localSP2)
  (estaEm pkg2 localSP2)
  (estaParadoNo aviao1 aeroportoSP)
  (estaParadoNo aviao2 aeroportoRJ1)
  (estaEm pkg3 localRJ2)
  (estaEm pkg4 aeroportoRJ1)
  (estaNo caminhao2 localRJ1)
  (= (capacidade caminhao1) 4)
  (= (carregamentoCorrente caminhao1) 0)
  (= (capacidade caminhao2) 2)
  (= (carregamentoCorrente caminhao2) 0)
  (= (capacidade aviao1) 10)
  (= (carregamentoCorrente aviao1) 0)
  (= (capacidade aviao2) 10)
  (= (carregamentoCorrente aviao2) 0)
)
```

4.5.2.3. Estado Objetivo – (:goal)

Todos os atributos, com seu respectivos valores, e associações entre objetos (Agentes e Recursos) encontrados no *Snapshot Objetivo* do problema são representados no elemento `:goal` da PDDL. Estes valores dos atributos e as associações definem o estado objetivo (final) do problema de planejamento.

Finalmente, conforme visto no procedimento *XMLtoXPDDLProblem*, o passo 4.2 representa o processo que o itSIMPLE realiza para formar o estado objetivo do problema em XPDDL. Neste passo todos os atributos (e seus respectivos valores) e associações entre

objetos são representadas no elemento :`goal` (no caso da XPDDL, <goal>) da mesma forma daquela apresentada na Tabela 16 no tópico anterior.

No exemplo do domínio *Logística* e do problema de planejamento apresentados no Capítulo 3, o estado objetivo representado em PDDL seria da seguinte forma:

```
(:goal
 (and
   (pertence aeroportoSP saopaulo)
   (pertence aeroportoRJ1 riodejaneiro)
   (pertence localSP1 saopaulo)
   (pertence localRJ1 riodejaneiro)
   (pertence localSP2 saopaulo)
   (pertence localRJ2 riodejaneiro)
   (estaNo caminhao1 localSP1)
   (estaParadoNo aviao1 aeroportoSP)
   (estaParadoNo aviao2 aeroportoRJ1)
   (estaEm pkg3 localSP1)
   (estaEm pkg4 localSP1)
   (estaEm pkg1 localRJ1)
   (estaEm pkg2 localRJ1)
   (estaNo caminhao2 localRJ1)
   (= (capacidade caminhao1) 4)
   (= (carregamentoCorrente caminhao1) 0)
   (= (capacidade caminhao2) 2)
   (= (carregamentoCorrente caminhao2) 0)
   (= (capacidade aviao1) 10)
   (= (carregamentoCorrente aviao1) 0)
   (= (capacidade aviao2) 10)
   (= (carregamentoCorrente aviao2) 0)
 )
)
```

Este capítulo apresentou a ferramenta itSIMPLE que representa uma das principais contribuições do presente trabalho. De fato, o itSIMPLE é a implementação dos conceitos do *Ambiente* apresentados no Capítulo 3. Esta implementação foi realizada, principalmente, para consolidar e verificar os pontos discutidos nesta proposta. Durante a elaboração e desenvolvimento da ferramenta itSIMPLE, uma variedade de domínios e problemas de planejamentos (muitos deles clássicos) foram modelados com o objetivo de guiar o processo de desenvolvimento deste protótipo, como, por exemplo, o domínio *Logística* ilustrado no decorrer da apresentação dos capítulos anteriores. Alguns destes domínios e problemas modelados realçaram os principais pontos positivos no uso do itSIMPLE. No próximo capítulo são apresentados alguns desses modelos de domínios na forma de estudos de casos para ferramenta e, conseqüentemente, para o *Ambiente* apresentado.

Capítulo 5

5. Estudos de Casos

Neste capítulo são apresentados alguns estudos de casos de modelagem e análise de domínios de planejamento automático realizados durante a elaboração do presente trabalho com o objetivo de verificar os conceitos do *Ambiente* proposto, bem como os processos de modelagem e análise descritos anteriormente, e ressaltar as vantagens na utilização de uma ferramenta de suporte durante o design destes domínios. De fato, foram modelados diversos domínios clássicos de planejamento durante a elaboração deste trabalho utilizando os processos e os conceitos do *Ambiente*. Muitos destes domínios guiaram o desenvolvimento do protótipo da ferramenta itSIMPLE apresentada no Capítulo 4.

Entre os domínios modelados, aqueles de maior destaque foram selecionados e estes são apresentados neste capítulo focando principalmente nos processos de modelagem com a UML.P, análise dos aspectos dinâmicos e testes com planejadores. Primeiramente, são apresentadas a modelagem e análise de um dos domínios mais conhecidos na área do Planejamento Automático em IA, o chamado "*Mundo de Blocos"*, que neste trabalho recebe algumas *extensões*, como será visto nos próximos tópicos, para torná-lo mais complexo. Em seguida é apresentado o modelo do domínio "*Logística"* (e também como este é modelado), mas com características adicionais (consumo de combustível, postos de combustível, entre outras) que o tornam mais *complexo e realístico* em relação aquele apresentado durante o decorrer dos capítulos 3 e 4. Por fim, com o objetivo de ressaltar o potencial do *Ambiente* e dos conceitos apresentados neste trabalho, bem como levantar futuros trabalhos, o capítulo é

finalizado com uma breve apresentação da modelagem de um domínio de planejamento em manufatura, de mesmo nível de complexidade que os problemas reais de planejamento, chamado "*Montagem Seqüencial de Carros em Linhas de Montagem"* (NGUYEN, 2003), que resultou em uma publicação no ICAPS 2006 (VAQUERO et al., 2006). Este último domínio foi apresentado pela Renault em 2003 para a competição ROADEF'2005, onde equipes do mundo inteiro desenvolveram algoritmos capazes de ordenar os carros a serem montados de modo a evitar sobrecargas na linha de montagem.

5.1. Domínio Mundo de Blocos Estendido

5.1.1. Descrição

O *Mundo de Blocos* é um dos domínios mais conhecidos na área de Planejamento Automático em IA. Este domínio clássico vem sendo utilizando até hoje para ilustrar os principais conceitos teóricos e práticos sobre sistemas de planejamento. Neste trabalho será utilizado um domínio *Mundo de Blocos* um pouco mais elaborado, ou seja, o domínio clássico ganhará características extras que o tornam um pouco mais complexo. Este domínio mais elaborado é chamado aqui de *Mundo de Blocos Estendido*.

No *Mundo de Blocos Estendido,* além de braços robóticos rearranjarem um conjunto de blocos sobre uma determinada mesa, estes braços se movimentam de uma mesa de trabalho a outra (para as quais tem acesso) e podem com isso levar também blocos de uma mesa a outra. Blocos podem ser posicionados uns sobre os outros pelos braços robóticos em qualquer mesa do domínio. Para que acidentes sejam evitados, dois braços robóticos não podem trabalhar em uma mesma mesa simultaneamente, é necessário que a mesa esteja livre para que um braço se movimente até ela e inicio o trabalho de rearranjo dos blocos. A Figura 75 ilustra o domínio do *Mundo de Blocos Estendido* que será modelado e analisado nos próximos tópicos.

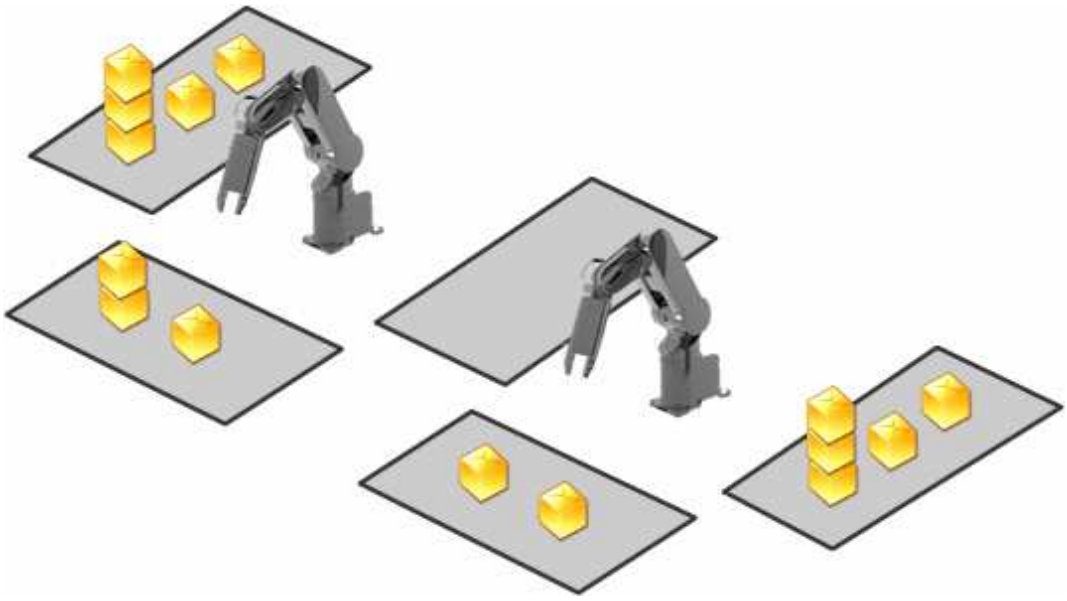


Figura 75 – Ilustração do *Mundo de Blocos Estendido*.

Conforme mostra a Figura 75, o domínio a ser modelado possui 2 (dois) braços robóticos, 5 (cinco) mesas de trabalho e 15 (quinze) blocos distribuído nas mesas. Cada braço tem acesso a apenas três mesas, onde uma delas pode ser acessada por ambos os braços.

5.1.2. Modelagem do Domínio

Conforme descrito nos capítulos anteriores, o processo de modelagem é iniciado pela construção do *Diagrama de Casos de Uso*. Baseado nas características do *Mundo de Blocos Clássico* e na descrição do *Mundo de Blocos Estendido* acima, o diagrama poderia ser construído com apenas cinco *casos de uso* (com suas respectivas descrições), sendo eles: "*Pick up block*", "*Put down block*", "*Stack block*", "*Unstack block*" e "*Change table*". Todos esses *casos de uso* são realizados pelo braço robótico, ou seja, neste domínio só existe um grupo de agentes, chamado aqui de *Arm*. A Figura 76 demonstra o *Diagrama de Casos de Uso* do domínio *Mundo de Blocos Estendido*.

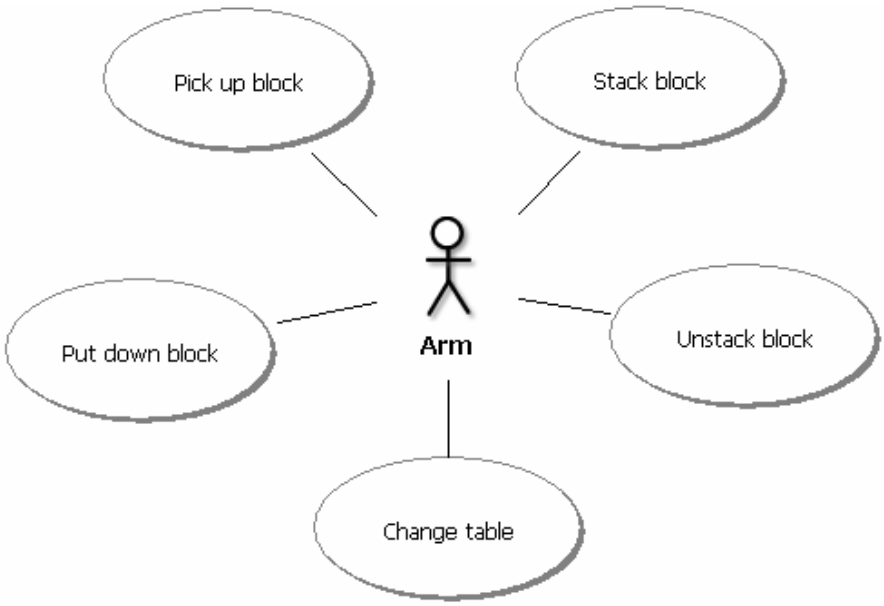


Figura 76 – *Diagrama de Casos de Uso* do *Mundo de Blocos Estendido*.

A descrição de cada um dos casos de uso presentes no *Diagrama de Casos de Uso* acima é apresentada na Tabela 17 a seguir.

Tabela 17 - Descrição dos *Caso de Usos* do *Mundo de Blocos Estendido*.

Caso de Uso: *Pick up block*
Agente: Arm **Descrição:** O agente Arm pega um bloco que está posicionado sobre a mesa. ***Restrições:*** **Pré-condição** - Arm precisa estar disponível, ou seja, sem estar segurando nenhum bloco; - Arm deve estar na mesma mesa que o bloco; - O bloco precisa estar livre, ou seja, sem nenhum outro bloco sobre o mesmo; - O bloco precisa estar sobre a mesa. **Pós-condições** - Arm passa a segurar o bloco e deixa de estar disponível; - O bloco não esta mais livre, e nem na mesa; - O bloco deixa de estar sobre a mesa.
Caso de Uso: *Put down block*
Agente: Arm **Descrição:** O agente Arm deixa um bloco sobre a mesa na qual está trabalhando.

Restrições:

Pré-condição
- O braço (Arm) precisa estar segurando o bloco.

Pós-condições
- O bloco passa a estar sobre a mesa;
- O bloco fica livre, ou seja, não existe nenhum bloco sobre o mesmo;
- Arm passa a ficar disponível.

Caso de Uso: *Stack block*

Agente: Arm

Descrição: O agente Arm deixa um bloco x qualquer sobre um outro bloco y em uma mesa.

Restrições:

Pré-condição
- O braço (Arm) precisa estar segurando o bloco x;
- Arm deve estar na mesma mesa que o bloco y, onde será colocado o bloco x;
- O bloco y deve estar livre.

Pós-condições
- Arm passa a estar disponível;
- O bloco x passa a estar livre sobre o bloco y;
- O bloco y deixa de estar livre.

Caso de Uso: *Unstack block*

Agente: Arm

Descrição: O agente Arm pega um bloco x qualquer que está sobre um outro bloco y em uma mesa.

Restrições:

Pré-condição
- Arm deve estar disponível;
- Arm deve estar na mesma mesa que o bloco x a ser pego;
- O bloco x deve estar livre;
- O bloco x deve estar sobre um outro bloco y.

Pós-condições
- Arm passa a segurar o bloco x;
- Arm deixa de estar disponível;
- O bloco x deixa de estar livre;
- O bloco y fica livre.

Caso de Uso: *Change table*

Agente: Arm

Descrição: O agente Arm muda de mesa de trabalho (da mesa de partida para a mesa de destino). O mesmo pode estar segurando um bloco ou não.

Restrições:

Pré-condição
- Arm deve estar em uma mesa de partida;
- Arm deve ter acesso à mesa de destino.

Pós-condições
- Arm passa a estar na mesa de destino.

Seguindo o processo de modelagem proposto, a estrutura estática do domínio (representada através do *Diagrama de Classes*) é modelada principalmente com base na descrição dos *casos de uso* apresentada na Tabela 17. Diante desta descrição pode-se dizer que os principais elementos deste domínio são os blocos, os braços e as mesas, sendo que os braços são os agentes do domínio e os blocos e mesas são os recursos do mesmo. Sendo assim o *Diagrama de Classes* possui apenas três classes chamadas aqui de *Arm*, *Block* e *Table* (conjunto dos tipos na definição do domínio), sendo que *Arm* é uma *Classe Agente* e *Block* e *Table* são *Classes Recursos*.

Neste modelo a classe *Block* terá dois atributos, sendo eles: *clear* (Boolean) que identifica se o bloco está livre e *atttable* (*Table*) que identifica em qual mesa se encontra o bloco. Esta classe pode ainda estar associada à classe *Table* pela associação *ontable* que indica o fato do bloco estar sobre a uma determinada mesa e também pode estar associada com ela mesma através da associação *on* indicando quando um bloco esta sobre outro. Em relação à classe *Table*, esta possui apenas um atributo chamado *available* (Boolean) que representa o fato da mesa estar disponível, ou seja, não há nenhum braço robótico trabalhando na mesma. A classe agente *Arm* também possui apenas um atributo chamado *handempty* (Boolean) que indica o fato do braço não estar segurando nenhum bloco. *Arm* possui também duas associações com a classe *Table* e uma com a classe *Block* que são: *at* indicando em qual mesa o braço se encontra; *hasAccessTo* que indica quais mesas o agente tem acesso; e *holding* representando o fato do braço segurar um bloco. Ainda em relação a *Arm*, esta classe é a única capaz de realizar ações sendo essas *pickUp, putDown, stack, unstack* e *changeTable*, conforme visto no *Diagrama de Casos de Uso* da Figura 76. O *Diagrama de Classes* resultante da modelagem da estrutura estática do modelo é apresentada na Figura 77, que leva em consideração as classes, atributos, associações e restrições apresentadas até o momento.

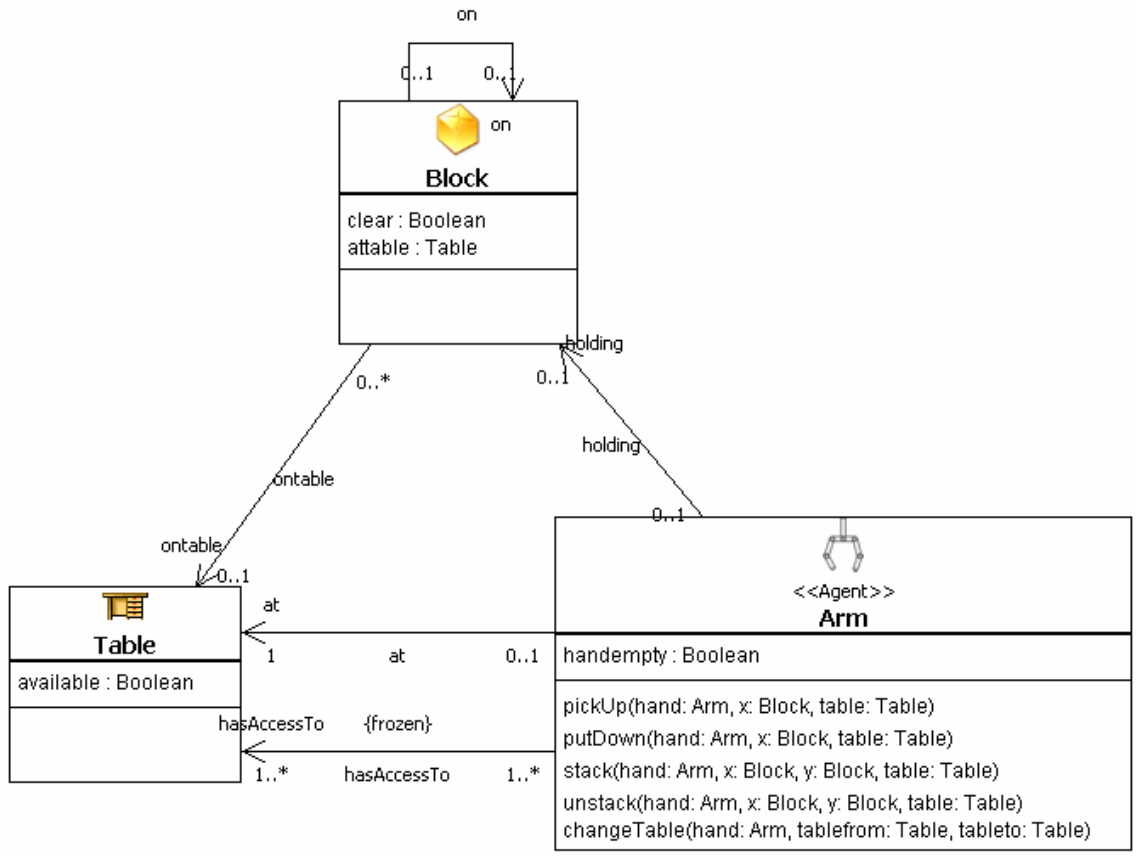


Figura 77 – *Diagrama de Classes* do *Mundo de Blocos Estendido*.

Neste modelo pode-se dizer que apenas duas classes possuem aspectos dinâmicos relevantes a serem representados e analisados, sendo elas *Arm* (*Classe Agente*) e *Block* (*Classe Recurso*). Iniciando pela classe *Arm*, pode-se modelar o comportamento da mesma levando em consideração os seguintes pontos:

- Supõe-se que um objeto agente do tipo *Arm* pode ser encontrado em apenas dois estados relevantes, sendo estes "*Arm holding block x at a table*" e "*Arm empty at a table*";
- As ações que podem afetar um objeto do tipo *Arm* são todas aqueles que este executa, ou seja: *pickUp*, *putDown*, *stack*, *unstack* e *changeTable* (realizadas pela própria classe *Arm*);

- As pré e pós-condições das ações que os objetos da classe *Arm* realizam são extraídas das descrições casos de uso e estas são representadas em OCL;
- Os atributos e associações relevantes para representação dos estados, conforme proposto neste trabalho, são: *handempty* e *holding*.

Focalizando exclusivamente na classe *Arm* é possível traçar as transições entre os estados (as ações) e suas respectivas pré e pós-condições no contexto *Arm*. A Figura 78 apresenta o *Diagrama de Estados* da classe *Arm*.

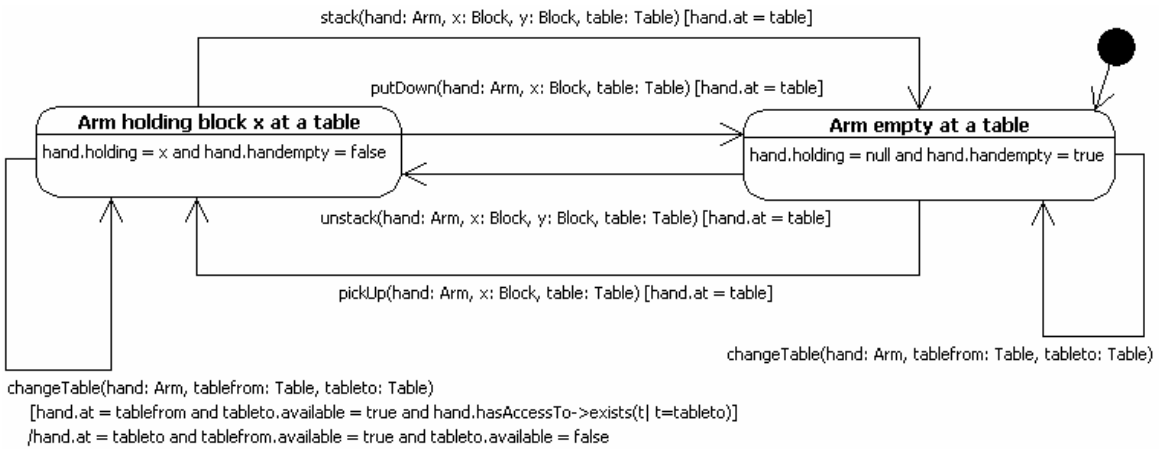


Figura 78 – *Diagrama de Estados* da classe *Arm*.

Em relação à classe *Block* alguns pontos relevantes são:

- Supõe-se que um objeto do tipo *Block* pode ser encontrado em apenas três estados relevantes, sendo eles "*Block x on table*", "*Block x being holded by an arm*" e "*Block x clear on a block y at a table*";
- As ações que podem afetar um objeto do tipo *Block* são: *pickUp*, *putDown*, *stack* e *unstack*;

- Novamente, as pré e pós-condições das ações sobre os objetos da classe *Block* são extraídas das descrições casos de uso e estas são representadas em OCL;
- Os atributos e associações relevantes para representação dos estados da classe *Block*, conforme proposto neste trabalho, são: *attable, ontable* e *on.*

A Figura 79 apresenta o *Diagrama de Estados* com foco na classe *Block*.

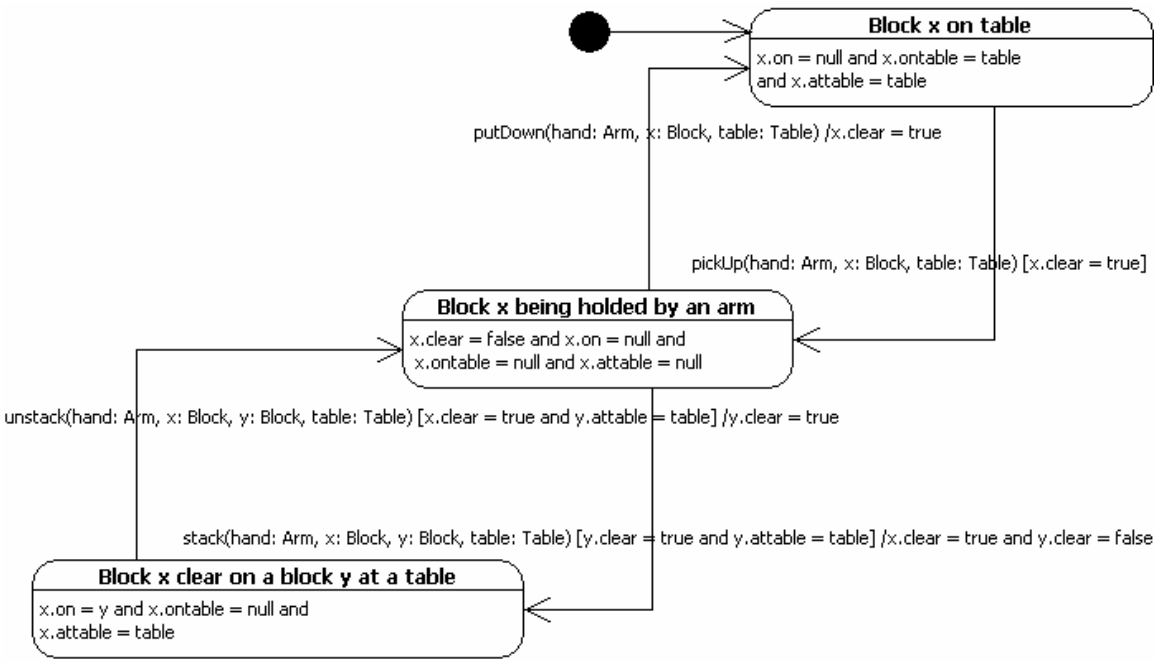


Figura 79 – *Diagrama de Estados* da classe *Block.*

Como resultado da união das expressões dos *Diagramas de Estados* das duas classes *Arm* e *Block* representados anteriormente, segue abaixo a representação das ações em OCL.

```
context Arm::pickUp(hand: Arm, x: Block, table: Table)
pre:
   -- Block conditions
     x.clear = true and x.on = null and x.ontable = table and x.attable = table and
   -- Arm conditions
     hand.at = table and hand.holding = null and hand.handempty = true

post:
   -- Block conditions
     x.clear = false and x.on = null and x.ontable = null and x.attable = null and
   -- Arm conditions
     hand.holding = x and hand.handempty = false
```

```
context Arm::putDown(hand: Arm, x: Block, table: Table)
pre:
   -- Block conditions
     x.clear = false and x.on = null and x.ontable = null and x.attable = null and
   -- Arm conditions
     hand.at = table and hand.holding = x and hand.handempty = false

post:
   -- Block conditions
     x.clear = true and x.on = null and x.ontable = table and x.attable = table and
   -- Arm conditions
     hand.holding = null and hand.handempty = true


context Arm::unstack(hand: Arm, x: Block, y: Block, table: Table)
pre:
   -- Block conditions
     x.clear = true and y.attable = table and x.on = y and
     x.ontable = null and x.attable = table and
   -- Arm conditions
     hand.at = table and hand.holding = null and hand.handempty = true

post:
   -- Block conditions
     y.clear = true and
     x.clear = false and x.on = null and x.ontable = null and x.attable = null and
   -- Arm conditions
     hand.holding = x and hand.handempty = false


context Arm::stack(hand: Arm, x: Block, y: Block, table: Table)
pre:
   -- Block conditions
     y.clear = true and y.attable = table and
     x.clear = false and x.on = null and x.ontable = null and x.attable = null and
   -- Arm conditions
     hand.at = table and hand.holding = x and hand.handempty = false

post:
   -- Block conditions
     x.clear = true and y.clear = false and
     x.on = y and x.ontable = null and x.attable = table and
   -- Arm conditions
     hand.holding = null and hand.handempty = true


context Arm::changeTable(hand: Arm, tablefrom: Table, tableto: Table)
pre:
   -- Arm conditions
     hand.at = tablefrom and tableto.available = true and
     hand.hasAccessTo->exists(t| t = tableto)

post:
   -- Arm conditions
     hand.at = tableto and tablefrom.available = true and tableto.available = false
```

Conforme visto no Capítulo 3, é possível verificar os *Diagramas de Estados* (não considerando as expressões OCL) através das Redes de Petri nas análises *Modular* e *de*

Interfaces. O próximo tópico mostrará exemplos de análise para o modelo do *Mundo de Blocos Estendido*.

Para completar o modelo do domínio é necessário ainda representar os agentes e recursos do mesmo. Conforme descrito no Capítulo 3, é possível utilizar o *Diagrama de Objetos* (chamado aqui neste contexto de *Repositório*) para representar estes objetos no modelo, mas para este estudo de caso, estes serão apenas enumerados. Assim, conforme ilustra a Figura 75, pode-se dizer que o conjunto de agentes é composto pelos objetos *hand1* e *hand2* da classe *Arm*, e o conjunto de recursos é composto pelos objetos *table1, table2, table3, table4* e *table5* da classe *Table*, bem como os objetos *b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14* e *b15* todos da classe *Block*.

5.1.3. Análise do Modelo do Domínio

Com base nos *Diagramas de Estados* gerados na modelagem dos aspectos dinâmicos é possível analisar o modelo tanto modularmente quando as interfaces de seus módulos em Redes de Petri (*Análise Modular* e *Análise de Interfaces*). No modelo do *Mundo de Blocos Estendido* sendo construído, algumas análises são interessantes a serem realizadas como, por exemplo, analisar individualmente os módulos das classes *Arm* e *Block*, bem como suas inter-relações. A Figura 80 e a Figura 81 demonstram, respectivamente, as análises *Modular* e *de Interfaces* do modelo.

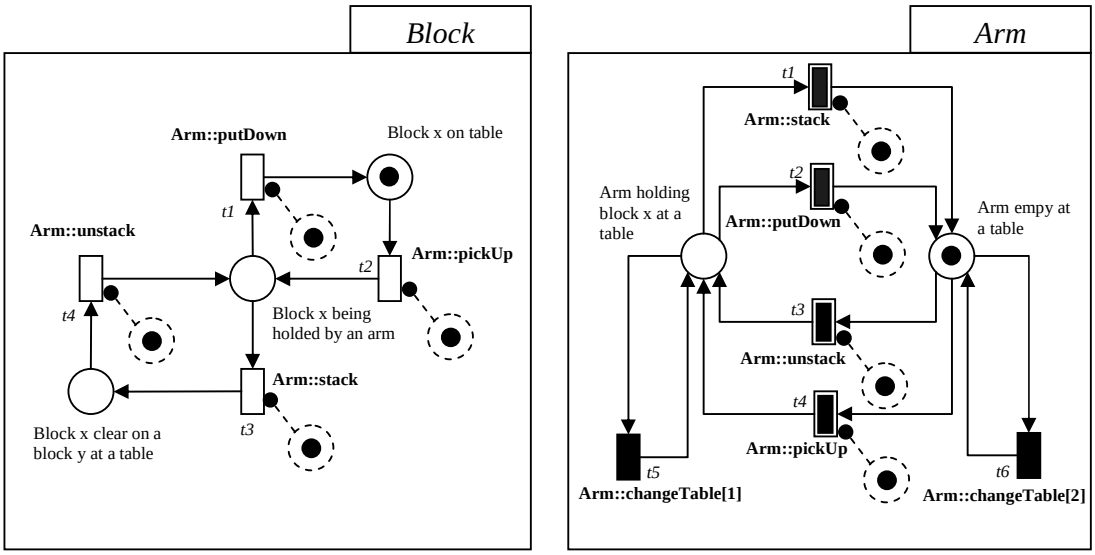


Figura 80 – *Análise de Modular* do comportamento das classes *Block* e *Arm*.

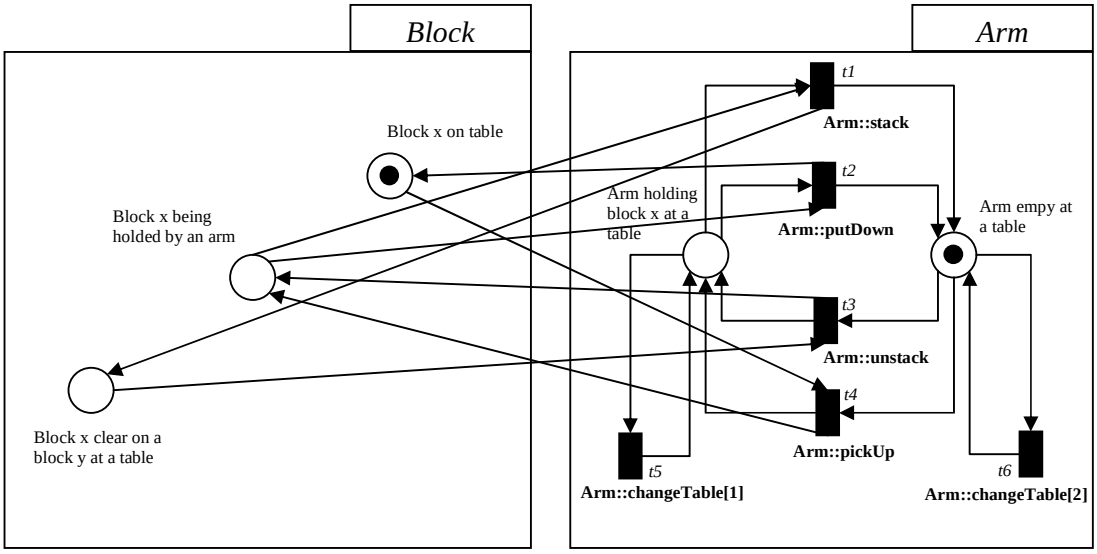


Figura 81 – *Análise de Interfaces* dos comportamentos das classes *Block* e *Arm*.

5.1.4. Modelagem do Problema

Conforme visto no Capítulo 3, os problemas de planejamento podem ser modelados através de dois diagramas *Snapshots* (*Diagrama de Objetos*) representando o estado inicial e o estado final (objetivo) do problema. No presente estudo de caso do domínio *Mundo de Blocos Estendido* um problema interessante seria aquele onde os braços robóticos tivessem que rearranjar os blocos nas mesas de tal maneira que um braço dependa do outro para o arranjo final desejado, ou seja, os braços devem trocar blocos para atingir o objetivo. Um problema como este é apresentado na Figura 82 (representando o *Snapshot Inicial*) e na Figura 83 (representado o *Snapshot Objetivo*) onde os dois estados são representados.

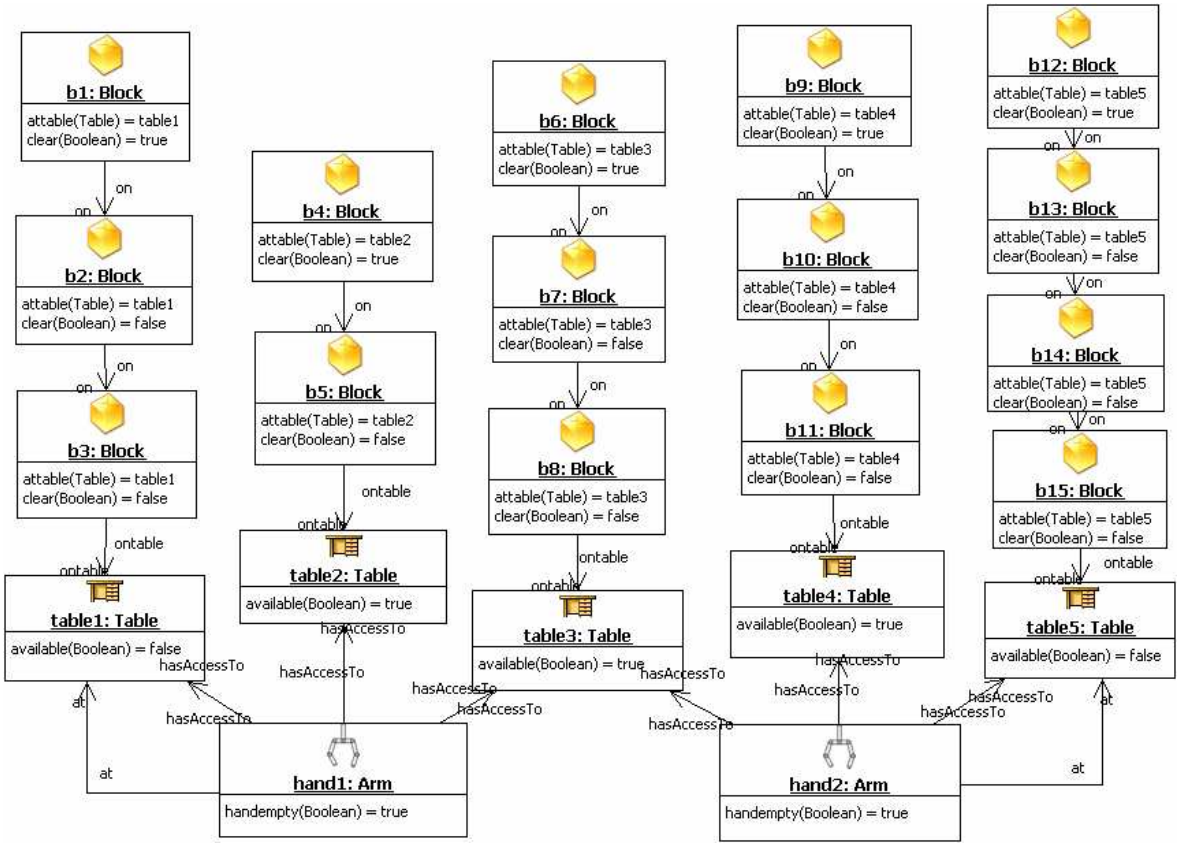


Figura 82 – *Snapshot Inicial* de um problema de planejamento no *Mundo de Blocos Estendido*.

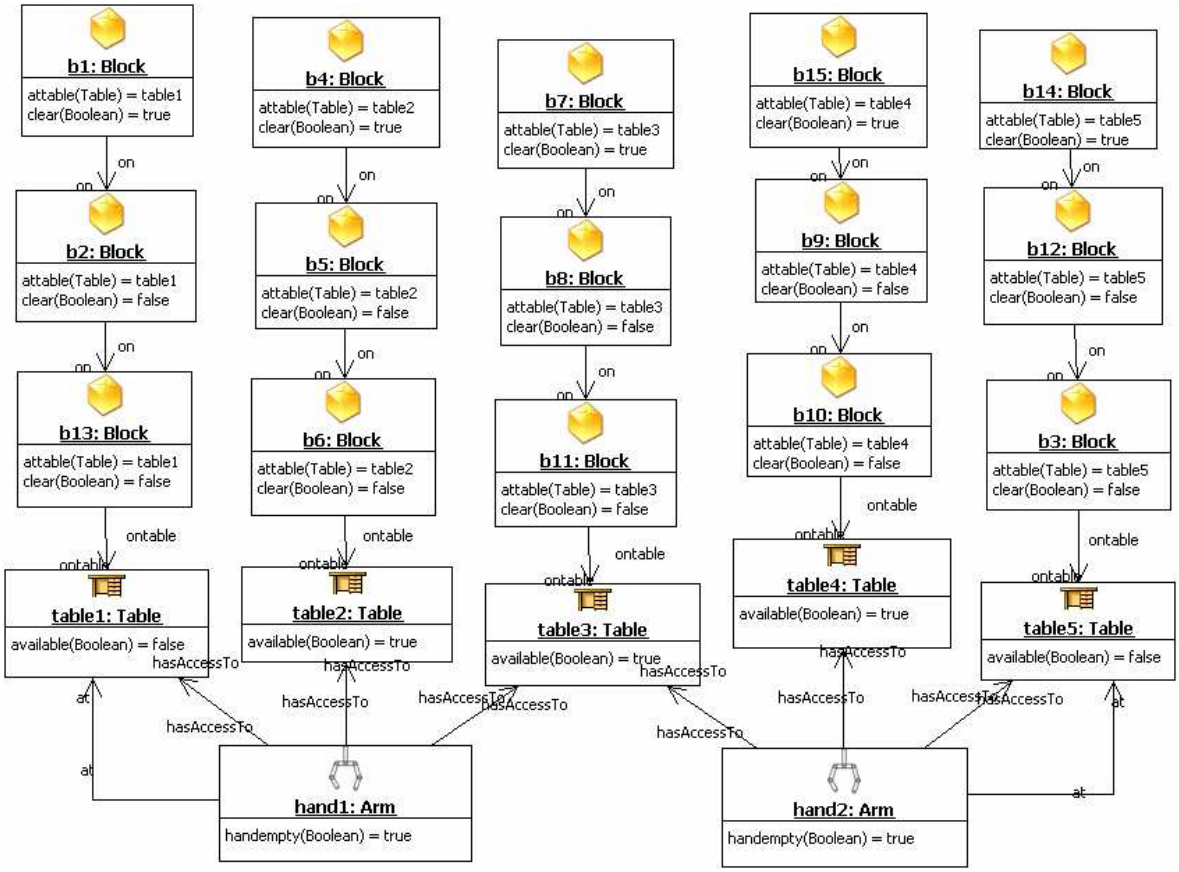


Figura 83 – *Snapshot Objetivo* de um problema de planejamento no *Mundo de Blocos Estendido*.

Neste problema de planejamento, o planejador deve utilizar os agentes para modificar a posição dos blocos para que o estado objetivo seja estabelecido. No próximo tópico é apresentada uma possível solução, fornecida por um algoritmo geral de planejamento, para o problema em questão.

5.1.5. Teste com Planejador

Neste estudo de caso, a fase de teste com planejador, para verificação e refinamento do modelo, foi realizada com o algoritmo Metric-FF (HOFFMANN, 2003). Para que o Metric-FF fosse utilizado na resolução do problema, foi necessário usufruir o processo de tradução para PDDL disponibilizado pela ferramenta itSIMPLE. Os arquivos PDDL com o modelo do

domínio e com o modelo do problema são encontrados anexados a este trabalho no Anexo A – Modelos de Domínios em PDDL. Vale citar que as restrições em PDDL (características da PDDL3.0) foram ignoradas, pois o planejador utilizado não é capaz de tratar tais informações. Assim a PDDL gerada possui as características disponíveis na versão PDDL2.1.

O planejador Metric-FF resolveu o problema modelado no tópico anterior (representado em PDDL) com um plano formado por 82 passos. O plano-solução para este problema de planejamento, ilustrado na Figura 82 e na Figura 83, é representado a seguir.

```
0: UNSTACK HAND2 B12 B13 TABLE5
1: CHANGETABLE HAND1 TABLE1 TABLE3
2: UNSTACK HAND1 B6 B7 TABLE3
3: CHANGETABLE HAND1 TABLE3 TABLE2
4: PUTDOWN HAND1 B6 TABLE2
5: PUTDOWN HAND2 B12 TABLE5
6: CHANGETABLE HAND1 TABLE2 TABLE1
7: UNSTACK HAND2 B13 B14 TABLE5
8: STACK HAND2 B13 B12 TABLE5
9: CHANGETABLE HAND2 TABLE5 TABLE4
10: CHANGETABLE HAND2 TABLE4 TABLE3
11: UNSTACK HAND2 B7 B8 TABLE3
12: CHANGETABLE HAND1 TABLE1 TABLE2
13: UNSTACK HAND1 B4 B5 TABLE2
14: PUTDOWN HAND1 B4 TABLE2
15: PICKUP HAND1 B5 TABLE2
16: STACK HAND1 B5 B6 TABLE2
17: PICKUP HAND1 B4 TABLE2
18: STACK HAND1 B4 B5 TABLE2
19: CHANGETABLE HAND1 TABLE2 TABLE1
20: STACK HAND2 B7 B8 TABLE3
21: CHANGETABLE HAND2 TABLE3 TABLE5
22: UNSTACK HAND2 B13 B12 TABLE5
23: PUTDOWN HAND2 B13 TABLE5
24: UNSTACK HAND2 B14 B15 TABLE5
25: STACK HAND2 B14 B12 TABLE5
26: PICKUP HAND2 B13 TABLE5
27: CHANGETABLE HAND2 TABLE5 TABLE3
28: PUTDOWN HAND2 B13 TABLE3
29: CHANGETABLE HAND2 TABLE3 TABLE5
30: PICKUP HAND2 B15 TABLE5
31: CHANGETABLE HAND2 TABLE5 TABLE4
32: CHANGETABLE HAND1 TABLE1 TABLE3
33: PICKUP HAND1 B13 TABLE3
34: CHANGETABLE HAND1 TABLE3 TABLE1
35: PUTDOWN HAND1 B13 TABLE1
36: STACK HAND2 B15 B9 TABLE4
37: CHANGETABLE HAND2 TABLE4 TABLE3
38: UNSTACK HAND2 B7 B8 TABLE3
39: UNSTACK HAND1 B1 B2 TABLE1
40: PUTDOWN HAND1 B1 TABLE1
41: UNSTACK HAND1 B2 B3 TABLE1
42: STACK HAND1 B2 B13 TABLE1
43: PICKUP HAND1 B1 TABLE1
44: STACK HAND1 B1 B2 TABLE1
45: STACK HAND2 B7 B8 TABLE3
46: CHANGETABLE HAND2 TABLE3 TABLE4
47: PICKUP HAND1 B3 TABLE1
48: CHANGETABLE HAND1 TABLE1 TABLE3
49: STACK HAND1 B3 B7 TABLE3
50: CHANGETABLE HAND1 TABLE3 TABLE1
51: CHANGETABLE HAND2 TABLE4 TABLE3
52: UNSTACK HAND2 B3 B7 TABLE3
53: CHANGETABLE HAND2 TABLE3 TABLE5
54: PUTDOWN HAND2 B3 TABLE5
55: UNSTACK HAND2 B14 B12 TABLE5
56: PUTDOWN HAND2 B14 TABLE5
57: PICKUP HAND2 B12 TABLE5
58: STACK HAND2 B12 B3 TABLE5
59: PICKUP HAND2 B14 TABLE5
60: STACK HAND2 B14 B12 TABLE5
61: CHANGETABLE HAND2 TABLE5 TABLE4
62: UNSTACK HAND2 B15 B9 TABLE4
63: PUTDOWN HAND2 B15 TABLE4
64: UNSTACK HAND2 B9 B10 TABLE4
65: STACK HAND2 B9 B15 TABLE4
66: UNSTACK HAND2 B10 B11 TABLE4
67: PUTDOWN HAND2 B10 TABLE4
68: UNSTACK HAND2 B9 B15 TABLE4
69: STACK HAND2 B9 B10 TABLE4
70: PICKUP HAND2 B15 TABLE4
71: STACK HAND2 B15 B9 TABLE4
72: PICKUP HAND2 B11 TABLE4
73: CHANGETABLE HAND2 TABLE4 TABLE3
74: PUTDOWN HAND2 B11 TABLE3
75: UNSTACK HAND2 B7 B8 TABLE3
76: PUTDOWN HAND2 B7 TABLE3
77: PICKUP HAND2 B8 TABLE3
78: STACK HAND2 B8 B11 TABLE3
79: PICKUP HAND2 B7 TABLE3
80: STACK HAND2 B7 B8 TABLE3
81: CHANGETABLE HAND2 TABLE3 TABLE5
```

Esta possível solução, ilustrada acima, faz com que os agentes (*hand1* e *hand2*) executem as ações de forma a levar o sistema do estado inicial ao estado objetivo (estados que foram modelados anteriormente). É possível perceber que a mesa *table3* é bastante utilizada para a troca de blocos, ou seja, a passagem de blocos de uma área de trabalho de um braço para outra área de trabalho de outro braço. Como discutido anteriormente, o projetista pode verificar como os recursos e agentes são utilizados nos planos e se estas utilizações se fazem adequada.

5.1.6. Observações

O domínio *Mundo de Blocos Estendido* foi um dos primeiros domínio estudados e modelados utilizando a ferramenta itSIMPLE. Um dos principais objetivos da modelagem e análise deste domínio foi investigar o potencial do *Ambiente* proposto para problemas de planejamento simples, fechados e clássicos que podem ser facilmente representados em linguagens que antecedem a PDDL (MCDERMOTT, 1998) como, por exemplo, STRIPS (FIKES; NILSSON, 1971) e que conseqüentemente podem ser resolvidos por técnicas de planejamento não tão recentes (planejadores clássicos).

Neste estudo de caso, os processos de modelagem e análise do domínio são relativamente simples já que suas características são bem conhecidas e muitas delas já foram bem estudas por muitos pesquisadores na área. Mesmo com a simplicidade do domínio algumas vantagens na utilização do *Ambiente* se tornam mais evidentes, sendo algumas delas: alguns diagramas no *Ambiente* propiciam um mapeamento relativamente claro entre o domínio real e seu modelo como, por exemplo, o *Diagrama de Casos de Uso*, *Diagrama de Classes* e *de Objetos* (fato que é evidenciado com a utilização de imagens associadas aos elementos dos diagramas); os estados criados nos *Snapshots* refletem claramente os estados de um domínio real, ou seja, o projetista identifica rapidamente as características do estado real nos *Snapshots* e este fato pode contribuir nas verificações do modelo; é possível perceber também que o

projetista pode criar diversas variações do domínio (variações no número de agentes e recursos, bem como suas configurações) dando flexibilidade no estudo do mesmo, mas vale citar que, neste caso, quanto maior o número de agentes e recursos no domínio, maior sua complexidade.

Mesmo o domínio *Mundo de Blocos Estendido* sendo simples suas características estão presentes em diversos outros domínios como, por exemplo, manufatura flexível, organização de containeres em navios e portos, gerenciamento de estoques, entre outros. Estes domínios, considerados mais realísticos, podem ser modelados reutilizando modelos já desenvolvidos como é o caso do *Mundo de Blocos*. Este fato faz com que o presente estudo de caso seja importante na construção de outros modelos de domínio de planejamento bem como a construção de uma possível base de modelos básicos que podem compor modelos mais complexos. Vale citar que esta reutilização de modelos e conhecimento é também um dos aspectos presentes na ferramenta itSIMPLE, bem como a manutenção dos mesmos.

5.2. Domínio Logística Estendido

5.2.1. Descrição

O domínio *Logística* é também um domínio clássico muito conhecido na área de Planejamento Automático em IA. No presente estudo de caso este domínio clássico é estendido com características que o tornam mais complexo e, conseqüentemente, mais desafiante. Esta extensão no domínio clássico é chamada aqui de *Logística Estendido.*

O domínio *Logística Estendido* possui as mesmas características daquele apresentado no Capítulo 3 (ou seja, pacotes são entregues em lugares específicos através de dos agentes caminhões e aviões observadas as restrições), porém algumas característica adicionais, sendo elas: neste domínio existem distâncias estabelecidas entre os lugares (locais e aeroportos) do

domínio, principalmente entre os aeroportos e entre os lugares de uma mesma cidade; os pacotes possuem pesos determinados; os agentes possuem limitações tanto de combustível quanto de carregamento, isto é, os caminhões e os aviões (com seus respectivos consumos) possuem tanques de combustível limitados, bem como uma capacidade máxima de carregamento de peso permitida durante o transporte; toda movimentação dos agentes de um lugar a outro está associado a um consumo de combustível; em alguns lugares do domínio existem estações de combustível, com diferentes preços de combustível, nas quais os veículos (agentes) podem reabastecer o tanque; e neste domínio existem diferentes números de agentes e recursos se comparado com o domínio modelado no Capítulo 3. Com essas características, o planejamento realizado sobre o domínio *Logística Estendido* tem por objetivo estabelecer a entrega de todos os pacotes em seus respectivos lugares de entrega considerando as autonomias dos agentes, minimizando ao máximo o combustível consumido bem como os gastos na compra de combustível para estabelecer tais entregas. A Figura 84 ilustra o domínio do *Logística Estendido* que será modelado e analisado nos próximos tópicos.

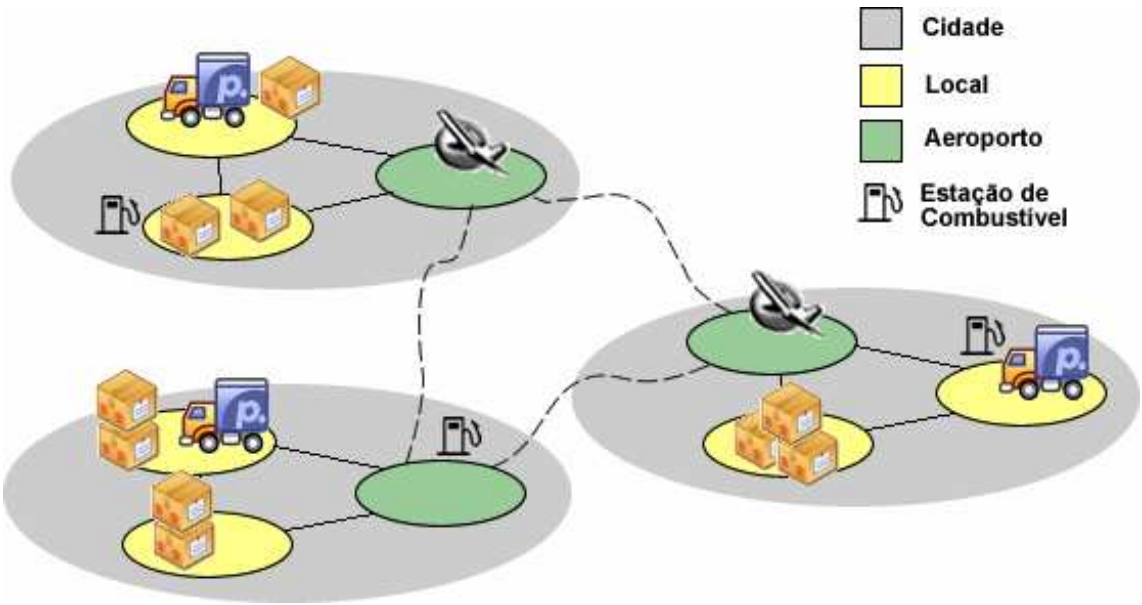


Figura 84 – Ilustração do domínio *Logística Estendido*.

Conforme mostra a Figura 84, o domínio a ser modelado possui: 3 (três) cidades onde cada uma delas possui 2 (dois) locais e um aeroporto, 3 (três) caminhões, 2 (dois) aviões, 3 (três) estações de combustível e 10 (dez) pacotes distribuídos em seus respectivos lugares. Cada um dos veículos do domínio em questão possui valores determinados de capacidade de carregamento, capacidade de combustível e consumo. As distâncias entre os lugares de uma mesma cidade e as distâncias entre os aeroportos são estabelecidas conforme será visto nos próximos tópicos durante a modelagem.

5.2.2. Modelagem do Domínio

Baseado nas características do domínio *Logística* clássico e na descrição do *Logística Estendido* acima, o *Diagrama de Casos de Uso* poderia ser construído inicialmente conforme a Figura 85. Pode-se perceber que a utilização de imagens (ícones) favorece o entendimento do diagrama perante os participantes na modelagem.

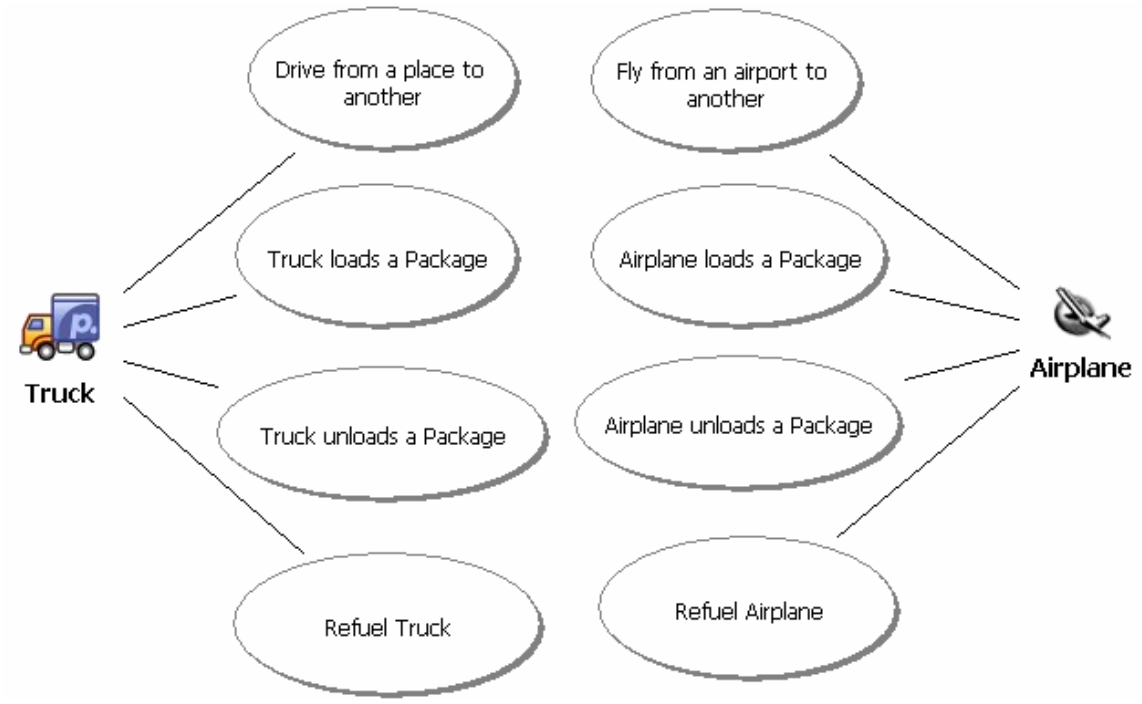


Figura 85 – *Diagrama de Casos de Uso* do *Logística Estendido*.

A descrição de cada um dos *casos de uso* presentes no *Diagrama de Casos de Uso* da Figura 85 é apresentada na Tabela 18 a seguir.

Tabela 18 - Descrição dos *Caso de Usos* do domínio *Logística Estendido*.

Caso de Uso: *Drive from a place to another*

Agente: Truck

Descrição: O agente Truck se locomove de um lugar a outro (seja este um local ou um aeroporto) da mesma cidade.

Restrições:

Pré-condição
- Truck deve estar em um lugar de partida pertencente a uma cidade e lugar de destino deve pertencer a essa mesma cidade;
- Truck deve ter autonomia de combustível para chegar ao lugar de destino. Esta autonomia é baseada na distância entre os lugares e o consumo do Truck (Km/L).

Pós-condições
- Truck passa a estar no lugar de destino;
- O combustível de Truck é consumido de acordo com a distância entre os lugares e o consumo do veículo.

Caso de Uso: *Truck loads a Package*

Agente: Truck

Descrição: O agente Truck carrega um pacote com um determinado peso.

Restrições:

Pré-condição
- Truck deve estar no mesmo lugar que o pacote a ser carregado;
- A capacidade do Truck não pode ser excedida ao carregar o pacote.

Pós-condições
- O pacote passa a estar dentro do Truck e deixa de estar no lugar;
- O carregamento do Truck é atualizado com o peso do pacote.

Caso de Uso: *Truck unloads a Package*

Agente: Truck

Descrição: O agente Truck descarrega um pacote em um lugar.

Restrições:

Pré-condição
- Truck deve estar em um lugar;
- O pacote deve estar dentro do Truck.

Pós-condições
- O pacote passa a estar no lugar e deixa de estar dentro do Truck;
- O carregamento do Truck é atualizado.

Caso de Uso: *Refuel Truck*

Agente: Truck

Descrição: O agente Truck abastece seu tanque de combustível em um lugar com uma estação de combustível (o tanque de combustível é sempre abastecido totalmente).

Restrições:

Pré-condição
- Truck deve estar em um lugar onde há uma estação de combustível;
- O tanque do Truck não pode estar cheio.

Pós-condições
- O tanque do Truck passa a ficar cheio.

Caso de Uso: *Fly from an airport to another*

Agente: Airplane

Descrição: O agente Airplane voa de um aeroporto a outro.

Restrições:

Pré-condição
- Airplane deve estar em um aeroporto de partida;
- Airplane deve ter autonomia de combustível para chegar ao aeroporto de destino. Esta autonomia é baseada na distância entre os aeroportos e o consumo do Airplane (Km/L).

Pós-condições
- Airplane passa a estar no aeroporto de destino;
- O combustível de Airplane é consumido de acordo com a distância entre os aeroportos e o consumo do veículo.

Caso de Uso: *Airplane loads a Package*

Agente: Airplane

Descrição: O agente Airplane carrega um pacote com um determinado peso.

Restrições:

Pré-condição
- Airplane deve estar no mesmo aeroporto que o pacote a ser carregado;
- A capacidade do Airplane não pode ser excedida ao carregar o pacote.

Pós-condições
- O pacote passa a estar dentro do Airplane e deixa de estar no aeroporto;
- O carregamento do Airplane é atualizado com o peso do pacote.

Caso de Uso: *Airplane unloads a Package*

Agente: Airplane

Descrição: O agente Airplane descarrega um pacote em um aeroporto.

Restrições:

Pré-condição
- Airplane deve estar em um aeroporto;
- O pacote deve estar dentro do Airplane.

Pós-condições
- O pacote passa a estar no aeroporto e deixa de estar dentro do Airplane;
- O carregamento do Airplane é atualizado.

Caso de Uso: *Refuel Airplane*

Agente: Airplane

Descrição: O agente Airplane abastece seu tanque de combustível em um aeroporto com uma estação de combustível (o tanque de combustível é sempre abastecido totalmente).

Restrições:

Pré-condição
- Airplane deve estar em um aeroporto onde há uma estação de combustível;
- O tanque do Airplane não pode estar cheio.

Pós-condições
- O tanque do Airplane passa a ficar cheio.

Novamente, seguindo o processo de modelagem proposto, a estrutura estática do domínio (representada no *Diagrama de Classes*) é modelada com base na descrição dos *casos de uso* apresentada na Tabela 18. A construção do *Diagrama de Classes* para o domínio *Logística Estendido* é similar àquela apresentada como exemplo no Capítulo 3 para o domínio *Logística* clássico. Assim, além das classes *City, Place, Location, Airport, Package, Vehicle, Truck* e *Airplane* do domínio clássico, a presente extensão possui ainda as classes *FuelStation* (representando a estação de combustível onde os veículos podem reabastecer) e *Utility* (representando a que possuirá todas as variáveis globais do domínio), sendo a primeira uma *Classe Recurso* e a segunda uma classe global (*stereotype* <<*utility*>>).

De modo a estender o domínio clássico neste estudo de caso, as características referentes às limitações e uso de combustível, bem como as restrições de carregamento, são modeladas aqui utilizando atributos e associações de classes do modelo da estrutura estática. No modelo do domínio *Logística Estendido*, diferentemente do domínio clássico, a classe *Package* possui um atributo chamado *weight* (Float) representando o peso do pacote (em Kg). Já a classe *Vehicle* possui cinco atributos, sendo eles: *capacity* (Float), identificando a capacidade de carregamento do veículo (Kg); *currentLoad* (Float), representando o carregamento corrente (Kg); *fuelCapacity* (Float), identificando a capacidade do tanque de combustível (capacidade representada neste estudo de caso em Litros); *fuelLevel* (Float) representando a quantidade

corrente de combustível (também em Litros); e *fuelBurn* (Float) representando o consumo médio do veículo, por exemplo, em Km/L. A classe *FuelStation* possui um atributo chamado *fuelPrice* que identifica o preço (em Reais – R$) da unidade de combustível (no caso, o Litro). Além disso, a classe *Place* possui uma associação *has* com a classe *FuelStation* de forma a identificar aonde os agentes poderão encontrar as estações de combustível. Finalmente, a classe *Utility* possui três atributos: *distance(p1:Place, p2:Place)* (Float) simbolizando as distâncias entre os lugares do domínio (valores em Km); *totalFuelUsed* (Float) representando a variável global que armazenará o valor total de combustível usado (em Litros) para resolver os problemas de planejamento (variável que deve ser minimizada); e *totalFuelCost* (Float) representando o total de dinheiro (R$) gasto com combustível durante para durante após a execução dos planos-soluções dos problemas de planejamento (variável que também deve ser minimizada).

Devido ao fato dos veículos consumirem combustível é necessário que estes possam realizar ações de reabastecimento do tanque de combustível como mostra o *Diagrama de Casos de Uso* da Figura 85. Para isso, as classes *Truck* e *Airplane* possuem, além das ações de movimentação, carregamento de descarregamento, ações que fazem com que os veículos abasteçam seus tanques nas estações de combustível como é o caso da ação *refuelTruck* e *refuelAirplane.* O *Diagrama de Classes* resultante da modelagem da estrutura estática do modelo é apresentada na Figura 86, que leva em consideração as classes, atributos, associações e restrições discutidas até o momento.

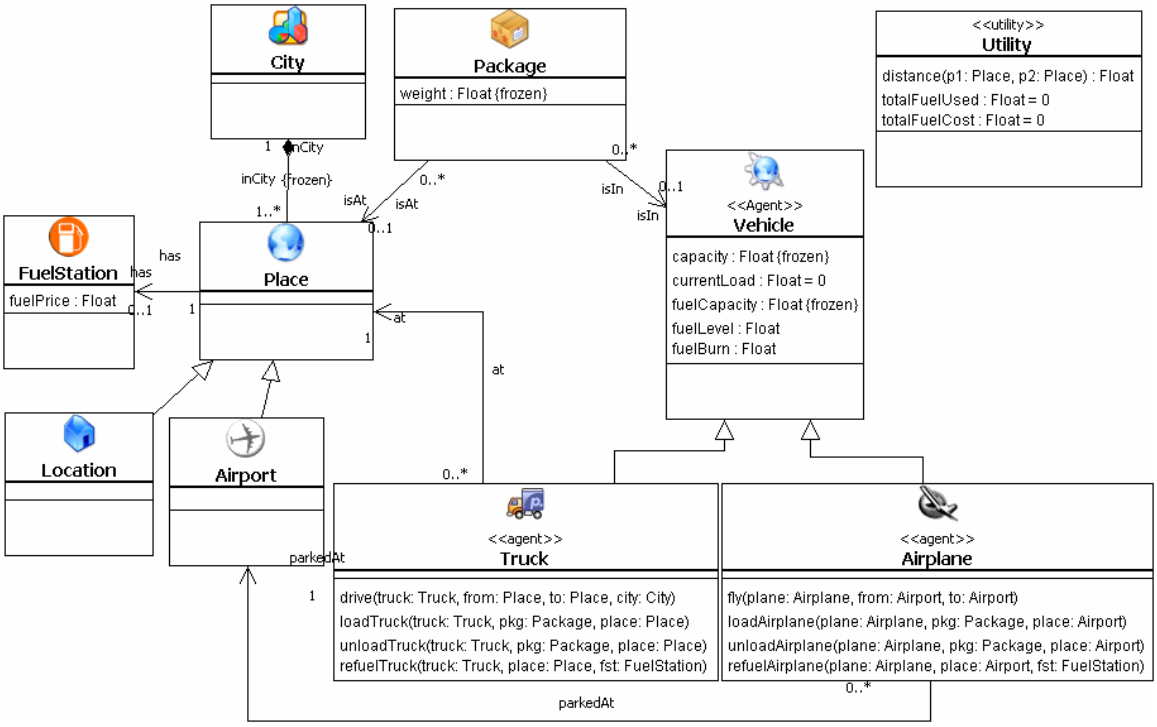


Figura 86 – *Diagrama de Classes* do domínio *Logística Estendido*.

Como no domínio *Logística* modelado no Capítulo 3, neste estudo de caso apenas três classes possuem aspectos dinâmicos relevantes a serem representados e analisados, sendo elas *Package* (*Classe Recurso*), *Truck* e *Airplane* (*Classes Agentes*). A modelagem desses aspectos dinâmicos do domínio *Logística Estendido* acaba sendo muito parecida com aquela apresentada nos capítulos anteriores. Assim, iniciando pela classe *Package*, pode-se modelar o comportamento da mesma através do *Diagrama de Estados* ilustrado na Figura 87 a seguir. Já o comportamento da classe *Truck* pode ser modelado conforme mostra o *Diagrama de Estados* da Figura 88. Em relação à classe *Airplane*, a Figura 89 demonstra o *Diagrama de Estados* resultante da modelagem dos aspectos dinâmicos dessa classe. Os três diagramas foram construídos baseados nas descrições dos *casos de uso* apresentados na Tabela 18.

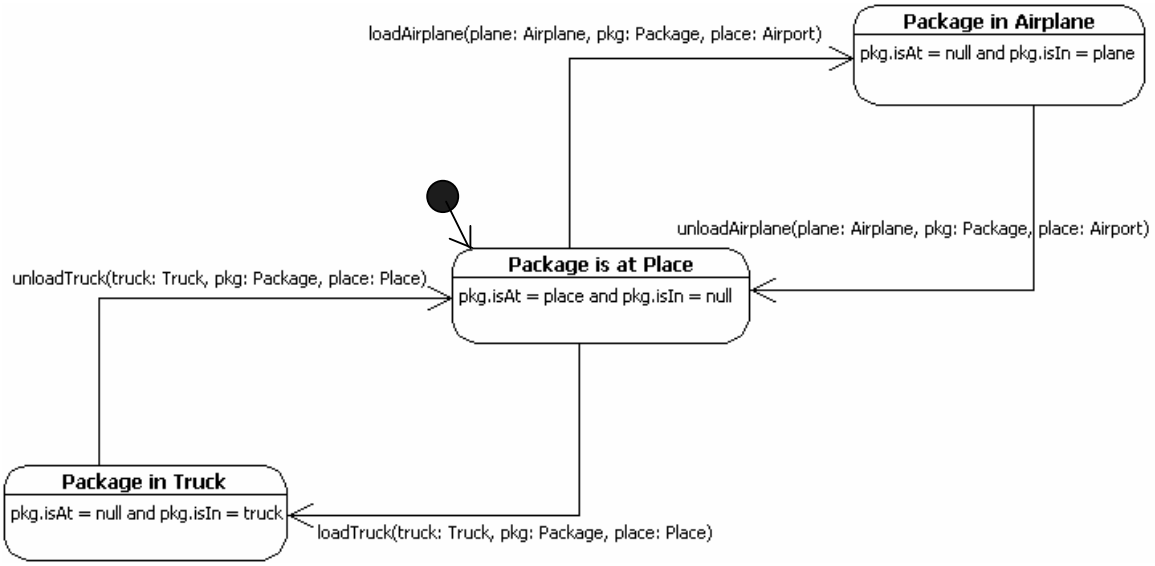


Figura 87 – *Diagrama de Estados* da classe *Package*.

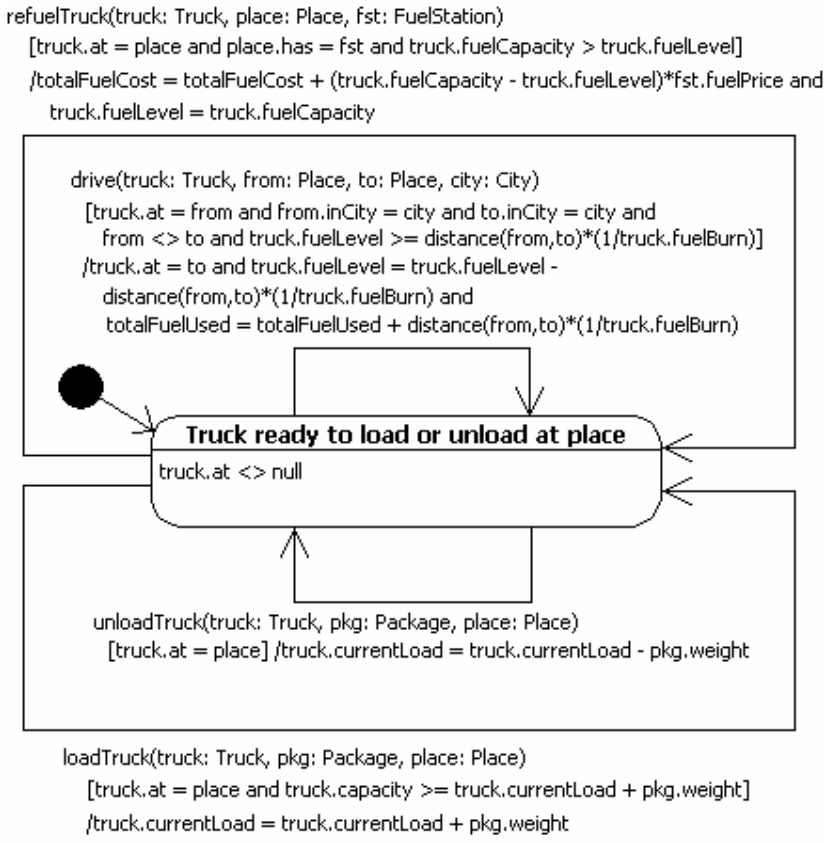


Figura 88 – *Diagrama de Estados* da classe *Truck.*

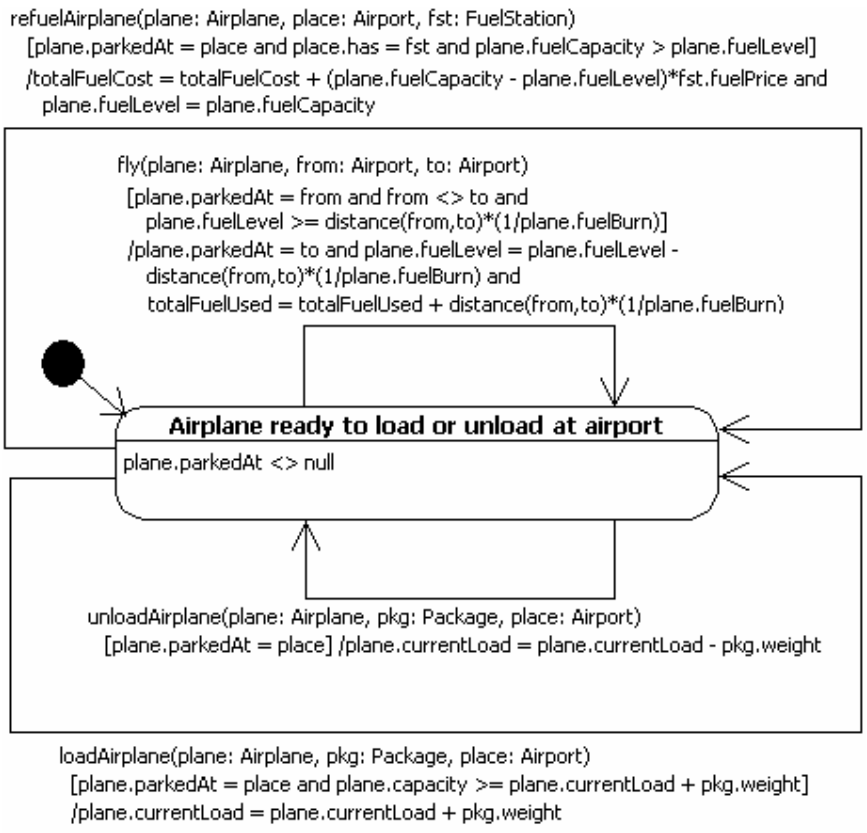


Figura 89 – *Diagrama de Estados* da classe *Airplane*.

Como resultado da união das expressões dos *Diagramas de Estados* das classes *Package*, *Truck* e *Airplane* representados anteriormente, segue abaixo a representação das ações em OCL.

```
context Truck::drive(truck: Truck, from: Place, to: Place, city: City)
pre:
   -- Truck conditions
     truck.at = from and from.inCity = city and to.inCity = city and from <> to and
     truck.fuelLevel >= distance(from,to)*(1/truck.fuelBurn) and truck.at <> null

post:
   -- Truck conditions
     truck.at = to and truck.fuelLevel = truck.fuelLeve –
       distance(from,to)*(1/truck.fuelBurn) and totalFuelUsed = totalFuelUsed +
       distance(from,to)*(1/truck.fuelBurn)
```

```
context Truck::loadTruck(truck: Truck, pkg: Package, place: Place)
pre:
   -- Truck conditions
     truck.at = place and truck.capacity >= truck.currentLoad + pkg.weight and
     truck.at <> null and
   -- Package conditions
     pkg.isAt = place and pkg.isIn = null

post:
   -- Truck conditions
     truck.currentLoad = truck.currentLoad + pkg.weight and
   -- Package conditions
     pkg.isAt = null and pkg.isIn = truck


context Truck::unloadTruck(truck: Truck, pkg: Package, place: Place)
pre:
   -- Truck conditions
     truck.at = place and truck.at <> null and
   -- Package conditions
     pkg.isAt = null and pkg.isIn = truck

post:
   -- Truck conditions
     truck.currentLoad = truck.currentLoad - pkg.weight and
   -- Package conditions
     pkg.isAt = place and pkg.isIn = null


context Truck::refuelTruck(truck: Truck, place: Place, fst: FuelStation)
pre:
   -- Truck conditions
     truck.at = place and place.has = fst and truck.fuelCapacity > truck.fuelLevel
     and truck.at <> null

post:
   -- Truck conditions
     totalFuelCost = totalFuelCost + (truck.fuelCapacity –
       truck.fuelLevel)*fst.fuelCost and truck.fuelLevel = truck.fuelCapacity


context Airplane::fly(plane: Airplane, from: Airport, to: Airport)
pre:
   -- Airplane conditions
     plane.parkedAt = from and from <> to and plane.fuelLevel >=
      distance(from,to)*(1/plane.fuelBurn) and plane.parkedAt <> null

post:
   -- Airplane conditions
     plane.parkedAt = to and plane.fuelLevel = plane.fuelLevel –
       distance(from,to)*(1/plane.fuelBurn) and
     totalFuelUsed = totalFuelUsed + distance(from,to)*(1/plane.fuelBurn)


context Airplane::loadAirplane(plane: Airplane, pkg: Package, place: Airport)
pre:
   -- Airplane conditions
     plane.parkedAt = place and plane.capacity >= plane.currentLoad + pkg.weight
     and plane.parkedAt <> null and
   -- Package conditions
     pkg.isAt = place and pkg.isIn = null

post:
   -- Airplane conditions
     plane.currentLoad = plane.currentLoad + pkg.weight and
   -- Package conditions
     pkg.isAt = null and pkg.isIn = plane
```

```
context Airplane::unloadAirplane(plane: Airplane, pkg: Package, place: Airport)
pre:
   -- Airplane conditions
     plane.parkedAt = place and plane.parkedAt <> null and
   -- Package conditions
     pkg.isAt = null and pkg.isIn = plane

post:
   -- Airplane conditions
     plane.currentLoad = plane.currentLoad - pkg.weight and
   -- Package conditions
     pkg.isAt = place and pkg.isIn = null


context Airplane::refuelAirplane(plane: Airplane, place: Airport, fst: FuelStation)
pre:
   -- Airplane conditions
     plane.parkedAt = place and place.has = fst and
     plane.fuelCapacity > plane.fuelLevel and plane.parkedAt <> null

post:
   -- Airplane conditions
     totalFuelCost = totalFuelCost + (plane.fuelCapacity –
       plane.fuelLevel)*fst.fuelCost and
     plane.fuelLevel = plane.fuelCapacity
```

Finalmente, para completar o modelo do domínio é necessário ainda representar os agentes e recursos do mesmo. Assim, conforme ilustra a Figura 84, pode-se dizer que o conjunto de agentes é composto pelos objetos *truck1*, *truck2* e *truck3* da classe *Truck*, e *plane1* e *plane2* da classe *Airplane.* Já o conjunto de recursos é composto pelos objetos: *saopaulo, riodejaneiro* e *belohorizonte* da classe *City*; os lugares *locSP1*, *locSP2* e *airportSP* pertencentes a cidade *saopaulo*; os lugares *locRJ1*, *locRJ2* e *airportRJ* pertencentes a cidade *riodejaneiro*; os lugares *locBH1*, *locBH2* e *airportBH* pertencentes a cidade *belohorizonte*; as estações de combustível *stf1, stf2* e *stf3* da classe *FuelStation*; os objetos *pkg1, pkg2, pkg3, pkg4, pkg5, pkg6, pkg7, pkg8, pkg9* e *pkg10* todos da classe *Package*; e um objeto global da classe *Utility* para representar as variáveis e declarações globais.

5.2.3. Análise do Modelo do Domínio

Como visto no Capítulo 3, com base nos *Diagramas de Estados* gerados na modelagem dos aspectos dinâmicos é possível realizar as análises *Análise Modular* e *Análise de Interfaces* utilizando as Redes de Petri. Similarmente ao modelo do domínio *Logística* apresentado nos

capítulos anteriores, as classes relevantes a serem analisadas são: *Package*, *Truck* e *Airplane*. Devido ao fato das análises modulares e de interfaces dessas classes no modelo do domínio *Logística Estendido* serem similares àquelas realizadas no Capítulo 3 (com diferença apenas nas ações de reabastecimento), neste tópico será apresentada somente a *Análise de Interfaces* com as três classes do modelo do domínio *Logística Estendido*. Assim, a Figura 90 demonstra a *Análise de Interfaces* do modelo.

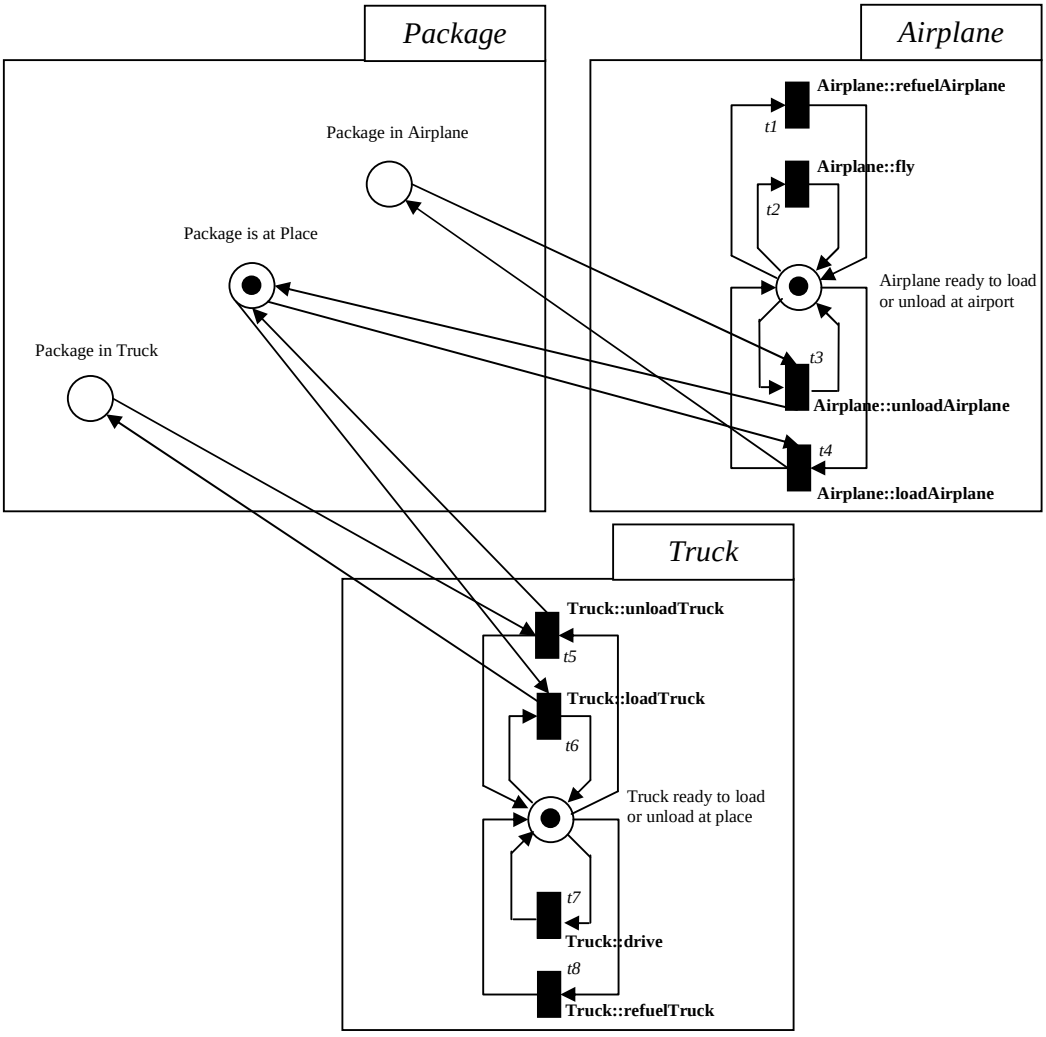


Figura 90 – *Análise de Interfaces* do comportamento das classes *Package*, *Truck* e *Airplane*.

5.2.4. Modelagem do Problema

No presente estudo de caso do domínio *Logística Estendido* um problema de planejamento interessante é demonstrado na Figura 91 e na Figura 92 utilizando os *Snapshots* para representar o estado inicial e o estado objetivo, respectivamente, onde as os pacotes são posicionados em seus ligares de partida, as capacidades de combustível e de carregamentos dos agentes são definidas, os pesos dos pacotes são determinados, os preços do combustível estabelecidos em cada estação, e as distâncias entre as cidades representadas de forma a fazer com que os agentes se mobilizem apara realizar as entregas conforme o estado objetivo.

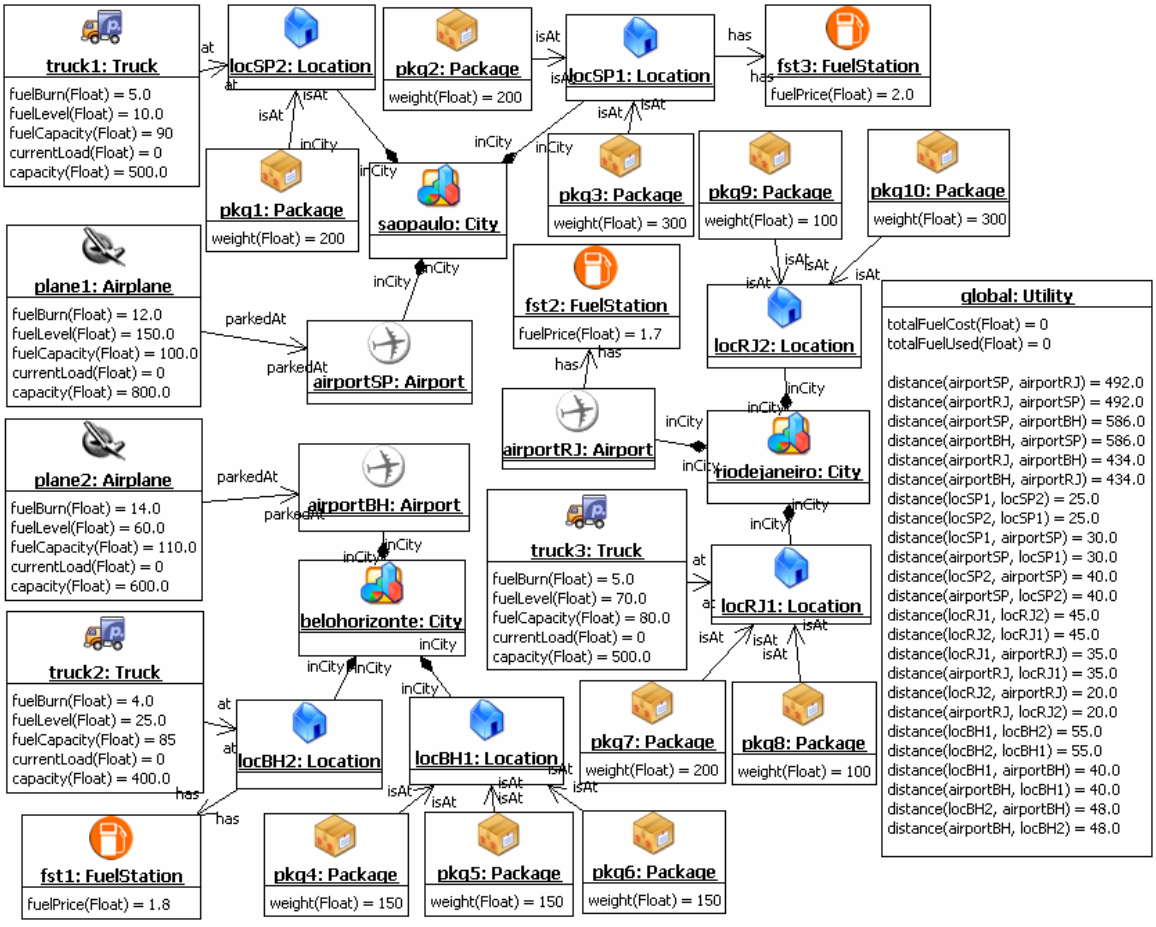


Figura 91 – *Snapshot Inicial* de um problema de planejamento no *Logística Estendido*.

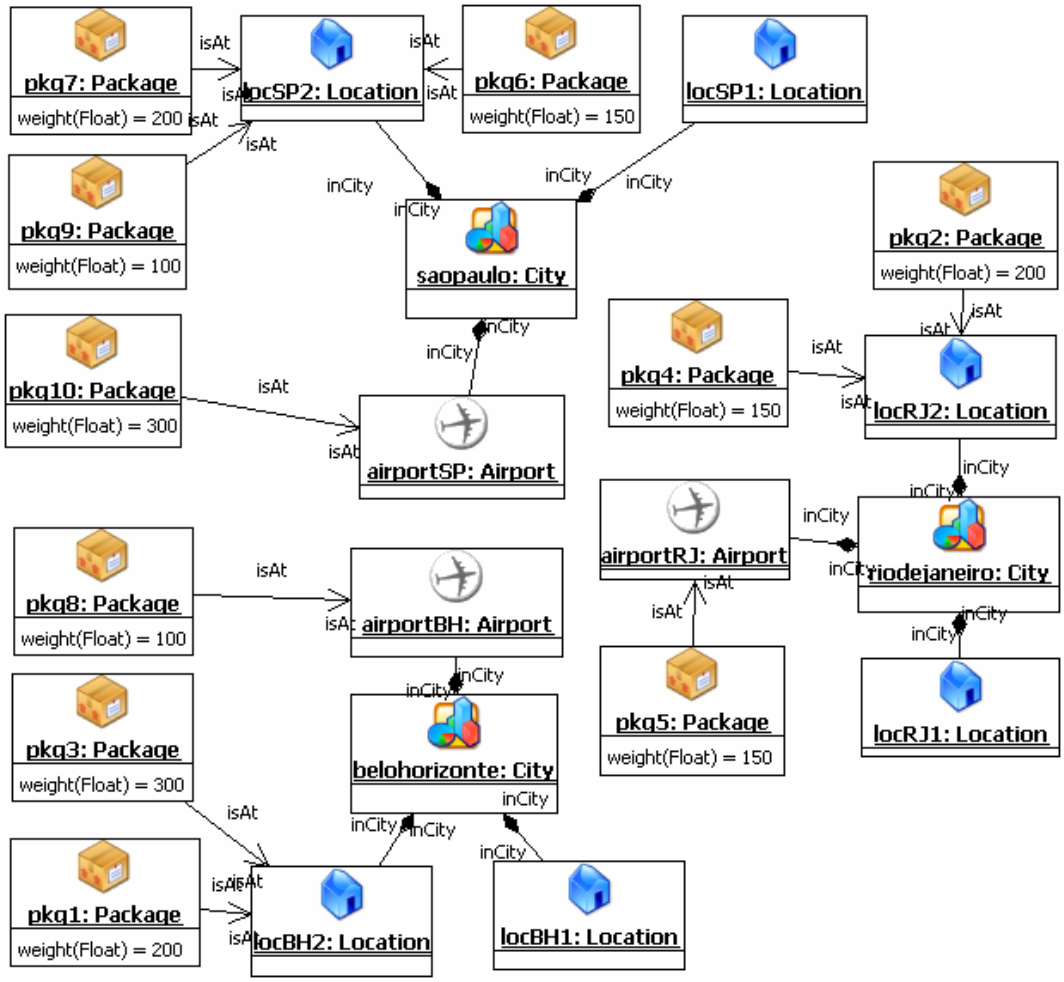


Figura 92 – *Snapshot Objetivo* de um problema de planejamento no *Logística Estendido*.

Neste estudo de caso, além de utilizar os agentes para realizar as entregas dos pacotes de forma correta, o planejador deve minimizar tanto o combustível utilizado durante as entregas quanto o valor gasto nos reabastecimentos de combustível. No próximo tópico são apresentadas algumas soluções, fornecida por alguns algoritmos gerais de planejamento, para o problema em questão.

5.2.5. Teste com Planejador

Conforme discutido no Capítulo 3 (no tópico *Testes com Planejadores*), é recomendado que o projetista teste o modelo com mais de um planejador para comparar possíveis soluções. Assim, neste estudo de caso, a fase de testes com planejadores para verificação e refinamento do modelo foi realizada utilizando dois algoritmos, sendo eles o Metric-FF (HOFFMANN, 2003) e o SGPlan5.2 (HSU et al., 2006) (um dos sucessores do Metric-FF). Para isso, o modelo foi primeiramente representado em PDDL através do processo de tradução do itSIMPLE. Depois disto foi acrescentada uma métrica ao modelo em PDDL já que o planejador deve minimizar o uso de combustível, bem como os gastos com reabastecimentos (ou seja, minimizar as variáveis globais *totalFuelUsed* e *totalFuelCost*). A métrica adicionada ao modelo (no arquivo do problema em PDDL) foi a seguinte:

```
(:metric minimize (+ (totalFuelUsed) (totalFuelCost)))
```

Com a métrica acima o planejador procura resolver o problema minimizando as varáveis de interesse neste estudo de caso. As representações em PDDL do modelo do domínio *Logística Estendido* e do problema de planejamento em questão podem ser encontradas anexadas a este trabalho no Anexo A – Modelos de Domínios em PDDL. Vale citar novamente que as restrições em PDDL foram ignoradas.

Primeiramente, o modelo foi testado com o Metric-FF que resolveu o problema modelado no tópico anterior (representado em PDDL) com um plano formado por 94 passos. O plano-solução, fornecido pelo Metric-FF, para o problema de planejamento, ilustrado na Figura 91 e na Figura 92, é representado a seguir.

```
0:  FLY PLANE1 AIRPORTSP AIRPORTBH
1:  LOADTRUCK TRUCK1 PKG1 LOCSP2
2:  LOADTRUCK TRUCK3 PKG8 LOCRJ1
3:  DRIVE TRUCK3 LOCRJ1 AIRPORTRJ RIODEJANEIRO
4:  UNLOADTRUCK TRUCK3 PKG8 AIRPORTRJ
5:  DRIVE TRUCK3 AIRPORTRJ LOCRJ1 RIODEJANEIRO
6:  LOADTRUCK TRUCK3 PKG7 LOCRJ1
7:  DRIVE TRUCK3 LOCRJ1 AIRPORTRJ RIODEJANEIRO
8:  UNLOADTRUCK TRUCK3 PKG7 AIRPORTRJ
9:  DRIVE TRUCK3 AIRPORTRJ LOCRJ2 RIODEJANEIRO
10: LOADTRUCK TRUCK3 PKG10 LOCRJ2
11: DRIVE TRUCK3 LOCRJ2 AIRPORTRJ RIODEJANEIRO
12: UNLOADTRUCK TRUCK3 PKG10 AIRPORTRJ
13: DRIVE TRUCK3 AIRPORTRJ LOCRJ2 RIODEJANEIRO
14: LOADTRUCK TRUCK3 PKG9 LOCRJ2
15: DRIVE TRUCK3 LOCRJ2 AIRPORTRJ RIODEJANEIRO
16: UNLOADTRUCK TRUCK3 PKG9 AIRPORTRJ
17: FLY PLANE2 AIRPORTBH AIRPORTSP
18: FLY PLANE1 AIRPORTBH AIRPORTRJ
19: LOADAIRPLANE PLANE1 PKG7 AIRPORTRJ
20: LOADAIRPLANE PLANE1 PKG8 AIRPORTRJ
21: LOADAIRPLANE PLANE1 PKG9 AIRPORTRJ
22: REFUELAIRPLANE PLANE1 AIRPORTRJ FST2
23: LOADAIRPLANE PLANE1 PKG10 AIRPORTRJ
24: FLY PLANE1 AIRPORTRJ AIRPORTSP
25: UNLOADAIRPLANE PLANE1 PKG7 AIRPORTSP
26: UNLOADAIRPLANE PLANE1 PKG9 AIRPORTSP
27: UNLOADAIRPLANE PLANE1 PKG10 AIRPORTSP
28: REFUELTRUCK TRUCK2 LOCBH2 FST1
29: DRIVE TRUCK2 LOCBH2 LOCBH1 BELOHORIZONTE
30: LOADTRUCK TRUCK2 PKG4 LOCBH1
31: DRIVE TRUCK2 LOCBH1 AIRPORTBH BELOHORIZONTE
32: UNLOADTRUCK TRUCK2 PKG4 AIRPORTBH
33: DRIVE TRUCK2 AIRPORTBH LOCBH1 BELOHORIZONTE
34: LOADTRUCK TRUCK2 PKG5 LOCBH1
35: DRIVE TRUCK2 LOCBH1 AIRPORTBH BELOHORIZONTE
36: UNLOADTRUCK TRUCK2 PKG5 AIRPORTBH
37: DRIVE TRUCK2 AIRPORTBH LOCBH1 BELOHORIZONTE
38: LOADTRUCK TRUCK2 PKG6 LOCBH1
39: DRIVE TRUCK2 LOCBH1 AIRPORTBH BELOHORIZONTE
40: UNLOADTRUCK TRUCK2 PKG6 AIRPORTBH
41: DRIVE TRUCK1 LOCSP2 LOCSP1 SAOPAULO
42: REFUELTRUCK TRUCK1 LOCSP1 FST3
43: LOADTRUCK TRUCK1 PKG2 LOCSP1
44: DRIVE TRUCK1 LOCSP1 AIRPORTSP SAOPAULO
45: LOADTRUCK TRUCK1 PKG9 AIRPORTSP
46: UNLOADTRUCK TRUCK1 PKG1 AIRPORTSP
47: LOADAIRPLANE PLANE1 PKG1 AIRPORTSP
48: UNLOADTRUCK TRUCK1 PKG2 AIRPORTSP
49: LOADAIRPLANE PLANE1 PKG2 AIRPORTSP
50: LOADTRUCK TRUCK1 PKG7 AIRPORTSP
51: DRIVE TRUCK1 AIRPORTSP LOCSP2 SAOPAULO
52: UNLOADTRUCK TRUCK1 PKG7 LOCSP2
53: UNLOADTRUCK TRUCK1 PKG9 LOCSP2
54: DRIVE TRUCK1 LOCSP2 LOCSP1 SAOPAULO
55: LOADTRUCK TRUCK1 PKG3 LOCSP1
56: DRIVE TRUCK1 LOCSP1 AIRPORTSP SAOPAULO
57: UNLOADTRUCK TRUCK1 PKG3 AIRPORTSP
58: FLY PLANE1 AIRPORTSP AIRPORTRJ
59: REFUELAIRPLANE PLANE1 AIRPORTRJ FST2
60: UNLOADAIRPLANE PLANE1 PKG2 AIRPORTRJ
61: LOADTRUCK TRUCK3 PKG2 AIRPORTRJ
62: DRIVE TRUCK3 AIRPORTRJ LOCRJ2 RIODEJANEIRO
63: UNLOADTRUCK TRUCK3 PKG2 LOCRJ2
64: FLY PLANE1 AIRPORTRJ AIRPORTBH
65: LOADAIRPLANE PLANE1 PKG4 AIRPORTBH
66: LOADAIRPLANE PLANE1 PKG5 AIRPORTBH
67: UNLOADAIRPLANE PLANE1 PKG8 AIRPORTBH
68: LOADAIRPLANE PLANE1 PKG6 AIRPORTBH
69: UNLOADAIRPLANE PLANE1 PKG1 AIRPORTBH
70: LOADTRUCK TRUCK2 PKG1 AIRPORTBH
71: DRIVE TRUCK2 AIRPORTBH LOCBH2 BELOHORIZONTE
72: REFUELTRUCK TRUCK2 LOCBH2 FST1
73: UNLOADTRUCK TRUCK2 PKG1 LOCBH2
74: FLY PLANE1 AIRPORTBH AIRPORTRJ
75: REFUELAIRPLANE PLANE1 AIRPORTRJ FST2
76: UNLOADAIRPLANE PLANE1 PKG4 AIRPORTRJ
77: UNLOADAIRPLANE PLANE1 PKG5 AIRPORTRJ
78: FLY PLANE1 AIRPORTRJ AIRPORTSP
79: LOADAIRPLANE PLANE1 PKG3 AIRPORTSP
80: UNLOADAIRPLANE PLANE1 PKG6 AIRPORTSP
81: LOADTRUCK TRUCK1 PKG6 AIRPORTSP
82: DRIVE TRUCK1 AIRPORTSP LOCSP2 SAOPAULO
83: FLY PLANE1 AIRPORTSP AIRPORTBH
84: UNLOADAIRPLANE PLANE1 PKG3 AIRPORTBH
85: UNLOADTRUCK TRUCK1 PKG6 LOCSP2
86: DRIVE TRUCK2 LOCBH2 AIRPORTBH BELOHORIZONTE
87: LOADTRUCK TRUCK2 PKG3 AIRPORTBH
88: DRIVE TRUCK2 AIRPORTBH LOCBH2 BELOHORIZONTE
89: UNLOADTRUCK TRUCK2 PKG3 LOCBH2
90: DRIVE TRUCK3 LOCRJ2 AIRPORTRJ RIODEJANEIRO
91: LOADTRUCK TRUCK3 PKG4 AIRPORTRJ
92: DRIVE TRUCK3 AIRPORTRJ LOCRJ2 RIODEJANEIRO
93: UNLOADTRUCK TRUCK3 PKG4 LOCRJ2
```

Já o planejador SGPlan5.2 resolveu o mesmo problema com apenas 42 passos, sendo eles:

```
0:  DRIVE TRUCK2 LOCBH2 LOCBH1 BELOHORIZONTE
1:  LOADTRUCK TRUCK2 PKG4 LOCBH1
2:  LOADTRUCK TRUCK2 PKG5 LOCBH1
3:  DRIVE TRUCK2 LOCBH1 AIRPORTBH BELOHORIZONTE
4:  UNLOADTRUCK TRUCK2 PKG4 AIRPORTBH
5:  LOADAIRPLANE PLANE2 PKG4 AIRPORTBH
6:  UNLOADTRUCK TRUCK2 PKG5 AIRPORTBH
7:  LOADAIRPLANE PLANE2 PKG5 AIRPORTBH
8:  DRIVE TRUCK3 LOCRJ1 AIRPORTRJ RIODEJANEIRO
9:  FLY PLANE2 AIRPORTBH AIRPORTRJ
10: UNLOADAIRPLANE PLANE2 PKG4 AIRPORTRJ
11: LOADTRUCK TRUCK3 PKG4 AIRPORTRJ
12: UNLOADAIRPLANE PLANE2 PKG5 AIRPORTRJ
13: REFUELAIRPLANE PLANE2 AIRPORTRJ FST2
14: FLY PLANE2 AIRPORTRJ AIRPORTSP
15: DRIVE TRUCK3 AIRPORTRJ LOCRJ2 RIODEJANEIRO
16: LOADTRUCK TRUCK3 PKG9 LOCRJ2
17: UNLOADTRUCK TRUCK3 PKG4 LOCRJ2
18: LOADTRUCK TRUCK3 PKG10 LOCRJ2
19: DRIVE TRUCK3 LOCRJ2 AIRPORTRJ RIODEJANEIRO
20: UNLOADTRUCK TRUCK3 PKG9 AIRPORTRJ
21: UNLOADTRUCK TRUCK3 PKG10 AIRPORTRJ
22: FLY PLANE2 AIRPORTSP AIRPORTRJ
23: LOADAIRPLANE PLANE2 PKG9 AIRPORTRJ
24: LOADAIRPLANE PLANE2 PKG10 AIRPORTRJ
25: FLY PLANE2 AIRPORTRJ AIRPORTSP
26: UNLOADAIRPLANE PLANE2 PKG9 AIRPORTSP
27: UNLOADAIRPLANE PLANE2 PKG10 AIRPORTSP
28: DRIVE TRUCK1 LOCSP2 LOCSP1 SAOPAULO
29: LOADTRUCK TRUCK1 PKG2 LOCSP1
30: REFUELTRUCK TRUCK1 LOCSP1 FST3
31: DRIVE TRUCK1 LOCSP1 AIRPORTSP SAOPAULO
32: UNLOADTRUCK TRUCK1 PKG2 AIRPORTSP
33: LOADAIRPLANE PLANE1 PKG2 AIRPORTSP
34: FLY PLANE1 AIRPORTSP AIRPORTRJ
35: UNLOADAIRPLANE PLANE1 PKG2 AIRPORTRJ
36: LOADTRUCK TRUCK1 PKG9 AIRPORTSP
37: DRIVE TRUCK1 AIRPORTSP LOCSP2 SAOPAULO
38: LOADTRUCK TRUCK3 PKG2 AIRPORTRJ
39: DRIVE TRUCK3 AIRPORTRJ LOCRJ2 RIODEJANEIRO
40: UNLOADTRUCK TRUCK1 PKG9 LOCSP2
41: UNLOADTRUCK TRUCK3 PKG2 LOCRJ2
```

De fato, ambas soluções, ilustradas anteriormente, fazem com que os agentes (*truck1, truck2, truck3, plane1* e *plane2*) executem as ações de forma a levar o sistema do estado inicial ao estado objetivo de maneira otimizada (minimizando as variáveis *totalFuelUsed* e *totalFuelCost*). No presente caso, o planejador SGPlan5.2 apresentou uma solução mais adequada em relação a redução do gasto de combustível e o do gasto com reabastecimentos dos veículos, o que reforça a realização de avaliações e testes com diversos planejadores. Novamente, o projetista pode verificar como os recursos e agentes são utilizados nos planos e se estas utilizações se fazem adequada.

5.2.6. Observações

Um dos principais objetivos da modelagem e análise deste domínio foi investigar o potencial do *Ambiente* proposto para problemas de planejamento com características mais complexas e mais realísticas que exigem mais do conhecimento do projetista. De fato, representar modelos mais complexos diretamente em PDDL se tornaria uma tarefa relativamente árdua para aqueles que não são familiarizados com a linguagem.

Neste estudo de caso, os processos de modelagem e análise do domínio são mais elaborados em relação ao estudo de caso anterior. Como visto, as ações, representadas em OCL, são mais complexas neste domínio, principalmente, devido à utilização de atributos numéricos. Esta utilização de propriedades numérica fornece grande flexibilidade e poder de expressão na modelagem de muitos domínios reais.

Novamente, é possível perceber que o projetista pode criar diversas variações do domínio *Logística Estendido* como, por exemplo, diversificando o número de agentes e recursos, bem como a topologia do domínio (cidades, locais e aeroportos). Como citado anteriormente, é possível também reutilizar modelos já existentes para tornar o domínio mais desafiante e mais próximo dos problemas reais como, por exemplo, unir características do domínio clássico

Mundo de Blocos ao domínio *Logística Estendido* seria uma boa prática, fazendo com que os pacotes sejam empilhados sobre *pallets* para que estes sejam então entregues aos clientes. De fato, o domínio *Logística Estendido* pode servir como base para a construção de diversos modelos mais elaborados de domínios de planejamento (por exemplo, Redes Logísticas de produtos envolvendo diversos meios de transportes aéreos, marítimos e terrestres).

5.3. Montagem Seqüencial de Carros em Linhas de Montagem

5.3.1. Descrição

No processo de planejamento do processo de montagem de carros nas linhas de montagem das grandes montadoras de veículos, os pedidos dos clientes são enviados às fábricas em tempo real. Estas fábricas devem estabelecer diariamente quais pedidos serão atendidos baseados na capacidade de produção das linhas disponíveis, nas restrições e nas datas de entrega. Assim, as fábricas precisam escalonar os veículos que serão colocados nas linhas para cada dia de produção, satisfazendo um conjunto de restrições complexas e principalmente as restrições da própria planta das fábricas. O estabelecimento da seqüência diária dos veículos nas linhas de produção tem um efeito direto na fábrica e na qualidade dos veículos. Uma seqüência inapropriada resulta em um fluxo de trabalho desbalanceado, perda de tempo e baixa produção, o que impacta nas vendas e na competitividade da montadora. Este domínio de planejamento abrange características interessantes tais como gerenciamento de recursos, escalonamento, otimização, minimização de custos, flexibilidade, entre outras características que quando combinadas tornam o problema ainda mais desafiante.

O presente domínio de planejamento *Montagem Seqüencial de Carros em Linhas de Montagem* foi inspirado em um dos desafios apresentados em uma grande competição de sistemas dedicados, chamada ROADEF Challenge. Este domínio de planejamento foi apresentado pela Renault em 2003 para a quarta edição da competição ROADEF Challenge

2005, onde pesquisadores desenvolveram sistemas dedicados a ordenar carros nas linhas de montagem de forma otimizada. Devido ao fato deste domínio possuir grandes desafios reais, o presente estudo de caso visou modelar e analisar tal domínio baseado na descrição do mesmo apresentada pela própria Renault, descrição que pode ser encontrada em (NGUYEN, 2003) e que será brevemente apresentada neste tópico. Os principais objetivos na modelagem e análise deste domínio de maior complexidade (próximo dos problemas reais) foram: verificar os conceitos e métodos propostos neste trabalho e levantar futuros trabalhos e necessidades para o *Ambiente* (itSIMPLE) bem como para os processos de modelagem em UML e de análise em Redes de Petri. A realização deste estudo de caso resultou em uma publicação no ICAPS 2006 (VAQUERO et al., 2006) que apresenta os processos de modelagem e análise deste domínio utilizando os conceitos apresentados no Capítulo 3.

Baseado na descrição do domínio e do problema de planejamento disponibiliza pela Renault (NGUYEN, 2003), de um modo geral, uma fábrica de automóveis possui três áreas importantes na montagem dos veículos, sendo elas: o *body shop* (1) onde são preparados os blanks dos automóveis; o *paint shop* (2) onde é realizada a pintura dos veículos; e a *assembly line* (3), ou seja, linha de montagem dos carros (NGUYEN, 2003). Cada veículo passa por essas três áreas na seguinte ordem: *body shop*, *paint shop* e *assembly line*. A Figura 93 ilustra estas áreas da montagem dos carros

Figura 93 – Principais áreas na *Montagem Seqüencial de Carros em Linhas de Montagem*.
Fonte: (NGUYEN, 2003).

O presente estudo de caso de modelagem e análise do domínio *Montagem Seqüencial de Carros em Linhas de Montagem* focaliza exclusivamente nos requisitos das áreas de pintura, *paint shop* (2), e montagem, *assembly line* (3), pois o *body shop* (1) não restringe a seqüência diária dos carros nas linhas. De fato, a ordem dos veículos não pode ser alterada durante os processos de pintura e montagem na produção diária em uma fábrica. Para identificar cada automóvel nos processos de pintura e montagem, os veículos são devidamente identificados antes mesmo de entrarem no *paint shop* com as seguintes características principais: um identificador único; sua seqüência na produção do dia; a cor escolhida pelo cliente; e as características especiais que serão montadas na linha de montagem como, por exemplo, teto solar, vidros elétricos, rodas de liga leve, entre outras.

Um sistema de planejamento que provê uma seqüência de carros nas linhas de montagem deve levar em consideração, além das características de cada veículo, os requisitos do *paint shop* (2) e do *assembly line* (3). A seguir são apresentados os requisitos extraídos das descrições encontradas em (NGUYEN, 2003).

Requisitos do *paint shop* (2). Esta parte da planta da fábrica deve considerar a minimização do solvente de pintura que utilizado para lavar a *pistola de spray* a cada troca de cor entre dois carros consecutivos. Implicitamente, agrupar os veículos de mesma cor evita grandes volumes de troca de tinta nas pistolas, minimizando assim as lavagens dos bicos do da *pistola de spray*. Em outras palavras, é necessário realizar o maior número de pinturas consecutivas com a mesma cor. Mesmo pintando vários veículos com uma mesma cor, a pistola de spray deve ser regularmente lavada, ou seja, existe um limite máximo de pinturas consecutivas que deve ser respeitado. Por exemplo, uma pistola de spray pode ter um limite de pintura de 10 carros consecutivos sem ter que lavar os bicos, e quando a pistola atingir o 11^{o} carro com a mesma cor, esta deve passar pelo processo de lavagem do bico (NGUYEN, 2003).

Requisitos do *assembly line* (3). O mais importante requisito encontrado na linha de montagem é a suavização o fluxo de montagem. Carros que precisam de características especiais (características que requerem operações extras de montagem como, ar condicionado, teto solar, entre outras) devem ser devidamente espaçados na seqüência de montagem do dia para evitar sobrecargas nas linhas. Este requisito está associado a uma razão N/P definida para cada característica especial, o que quer dizer que no máximo N carros em cada P carros na seqüência devem possuir tal característica. Por exemplo, a característica especial "teto solar" possui uma razão N/P = 1/3 significando não deve haver mais do que um carro em qualquer seqüência de 3 carros na ordem de montagem do dia (alguns exemplos de razões são 1/2, 2/5, 1/8, entre outros). Durante a elaboração da seqüência de carros de um dia de produção D, é necessário que o sistema de planejamento leve em consideração a seqüência estabelecida no dia anterior D-1. Já os carros que devem ser produzidos no dia D+1 são completamente ignorados. A violação destas razões N/P deve ser minimizada pelos planejadores durante a elaboração da seqüência de pintura e montagem dos carros. O domínio de planejamento fica mais interessante quando os carros dos pedidos dos clientes possuem mais de uma característica especial (fato que ocorre freqüentemente nas fábricas), assim o planejador deve procurar respeitar todas as razões.

Considerando ambos os requisitos (*paint shop* e *assembly line*), é necessário que o sistema de planejamento minimize tanto o número de trocas de cor na *pistola de spray* quanto o número de violações dos razões N/P de cada característica especial na ordem de montagem dos carros durante a definição da seqüência dos carros a serem montados em um dia de produção.

No presente estudo de caso, será modelado e analisado um domínio *Montagem Seqüencial de Carros em Linhas de Montagens* que possui uma oficina de pintura *paint shop* (com uma *pistola de spray*) e uma linha de montagem *assembly line* (com uma equipe de montagem)

onde o foco é atender um dia de produção de forma otimizada. Para isso, serão considerados neste estudo de caso oito carros a serem pintados e montados e estes podem ter apenas as características especiais “teto solar”, “ar condicionado” e “rodas especiais” de acordo com o pedido de cada cliente. A Figura 94 ilustra o domínio que será modelado nos próximos tópicos.

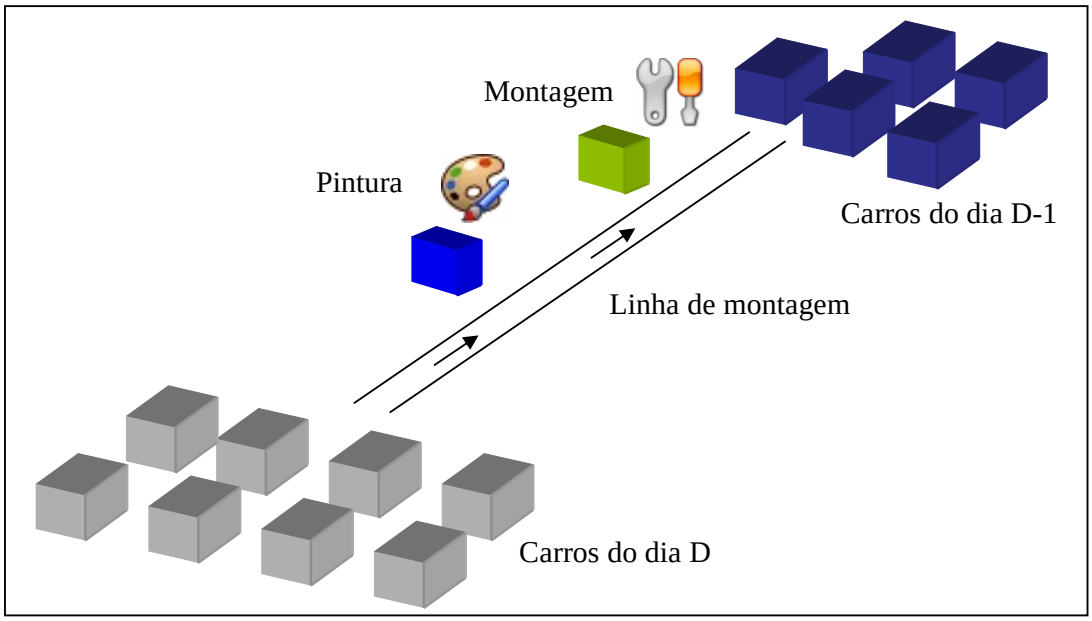


Figura 94 – Ilustração do domínio *Montagem Seqüencial de Carros em Linhas de Montagem*.

O modelo do domínio *Montagem Seqüencial de Carros em Linhas de Montagem* e seus respectivos processos de modelagem, análise e testes serão brevemente descritos e mostrados nos próximos tópicos. Maiores detalhes sobre esses processos podem ser encontrados em (VAQUERO, et al., 2006).

5.3.2. Modelagem do Domínio

Baseado nas características do domínio descritas anteriormente e nas descrições encontradas em (NGUYEN, 2003) o *Diagrama de Casos de Uso* poderia ser construído inicialmente

conforme a mostra Figura 95. Neste diagrama foram representados três agentes, sendo eles: a transportador (*Transporter*) que tem o papel de retirar os veículos do *body shop* e colocá-los no *paint shop* para então seguir para a montagem; o *SprayGun* que tem o papel de pintar os carros na ordem em que os estes foram colocados pelo *Transporter*; e o *Assembler* (um time de montagem) que tem o papel de mostrar os carros com suas respectivas características especiais.

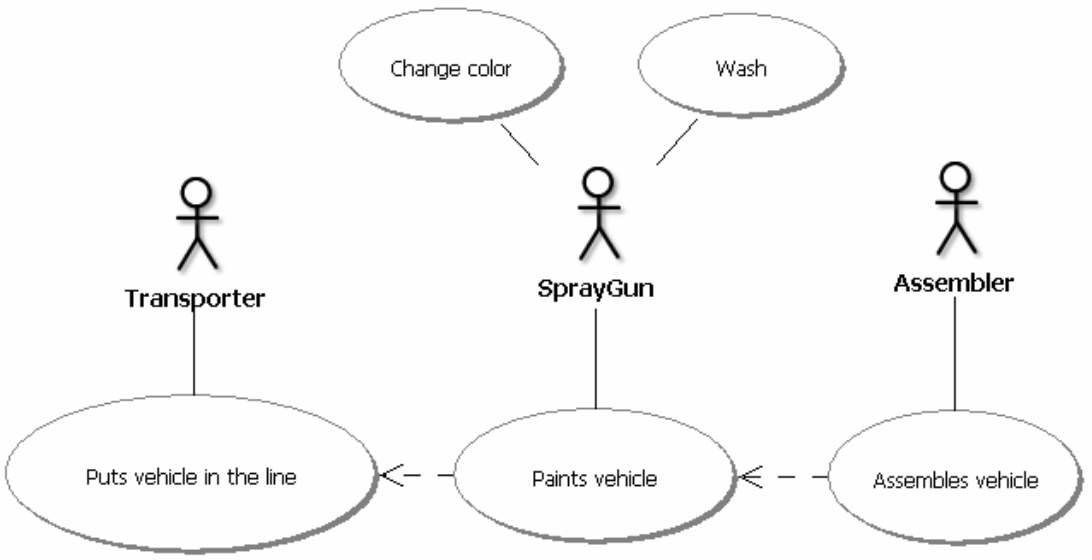


Figura 95 – *Diagrama de Casos de Uso* do domínio *Montagem Seqüencial de Carros em Linhas de Montagem*.

A descrição de cada um dos *casos de uso* presentes no *Diagrama de Casos de Uso* da Figura 95 é apresentada na Tabela 19 a seguir.

Tabela 19 - Descrição dos *Caso de Usos* do domínio *Montagem Seqüencial de Carros em Linhas de Montagem*.

Caso de Uso: *Puts vehicle in line*

Agente: Transporter

Descrição: O agente Transporter posiciona um veículo em uma linha para ser pintado e montado. Ao posicionar um veículo na linha o agente está definindo sua ordem na mesma.

Restrições:

Pré-condição
- O veiculo deve estar no body shop esperando para ser posicionado em uma linha, ou seja, a ordem do veículo não foi definida ainda em nenhuma linha;
- A linha na qual o veículo será posicionado possui a capacidade de montar as características especiais do mesmo.

Pós-condições
- A ordem e a linha do veículo são determinadas e ficam registradas no veículo. O veículo passa a ser o último na seqüência da linha.

Caso de Uso: *Paints vehicle*

Agente: SprayGun

Descrição: O agente SprayGun pinta o veículo na linha a qual ele pertence.

Restrições:

Pré-condição
- SprayGun deve pertencer a mesma linha onde o veículo se encontra;
- SprayGun deve estar com a cor que o carro precisa ser pintado;
- O limite de pinturas sucessivas com a mesma cor não deve ter sido atingido;
- O veículo é realmente o próximo a ser pintado;
- O veiculo não pode ter siso pintado antes e nem montado.

Pós-condições
- O veículo passa a estar pintado da cor determinada;
- Se o bico do SprayGun estava limpo, este deixa de estar, pois acaba de realizar uma pintura.

Caso de Uso: *Change color*

Agente: SprayGun

Descrição: O agente SprayGun muda a cor do seu spray.

Restrições:

Pré-condição
- O pico do spray deve estar limpo para a troca.

Pós-condições
- SprayGun passa a estar com a nova cor.

Caso de Uso: *Wash*

Agente: SprayGun

Descrição: O agente SprayGun lava o bico da pistola.

Restrições:

Pré-condição
- A pistola não pode estar limpa;
- A pistola deve ter pintado pelo menos um veículo.

Pós-condições
- A pistola passa a estar limpa.

Caso de Uso: *Assembles vehicle*

Agente: Assembler

Descrição: O agente Assembler faz a montagem do veículo e de todas as suas características especiais.

Restrições:

Pré-condição
- Assembler deve trabalhar na mesma linha onde o veículo se encontra;
- O veículo deve ser o próximo a ser montado;
- O veículo já deve ter sido pintado.

Pós-condições
- O veículo passa a estar montado com todas as características especiais.

Para melhor entender o processo de pintura dos veículos, a Figura 96 apresenta o *Diagrama de Atividades* da UML (OMG, 2001) utilizado para auxiliar a descrição dos casos de uso do agente SprayGun em relação às restrições durante as pinturas. De fato, o *Diagrama de Atividades* não foi contemplado durante a proposta da UML.P, mas, novamente, este estudo de caso visa o levantamento de novas necessidades durante a modelagem para futuros trabalhos. Assim, a utilização dos *Diagramas de Atividades* para complementar a descrição dos casos de uso faz-se relevante.

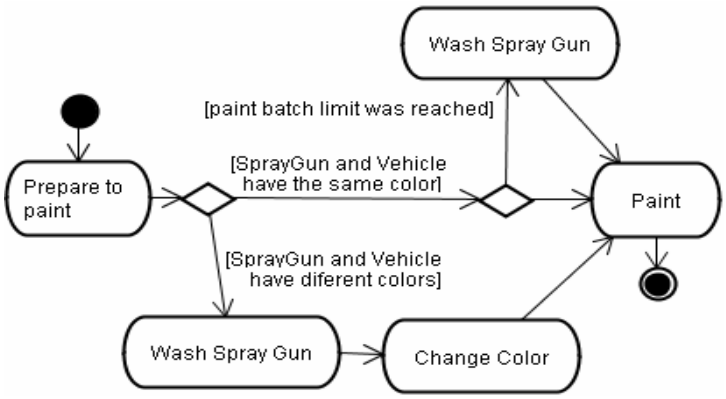


Figura 96 – *Diagrama de Atividades* para a pintura de um veículo.

Seguindo o processo de modelagem proposto, a estrutura estática do domínio (representada no *Diagrama de Classes*) é modelada com base na descrição dos *casos de uso* apresentada na Tabela 19. Diante desta descrição pode-se dizer que os principais elementos deste domínio que devem ser considerados no Diagrama de Classes são: os transportadores, as linhas de montagem com suas respectivas oficinas de pintura e montagem, os veículos, as características especiais e as cores de pintura. Conforme dito anteriormente, os agentes são apenas os transportadores (*Transporter*), as pistolas de spray (*SprayGun*) e os montadores (*Assembler*). A Figura 97 demonstra o *Diagrama de Classes* resultante da modelagem da estrutura estática do domínio. Os atributos e associações mais relevantes deste diagrama serão discutidos em seguida (VAQUERO et al., 2006).

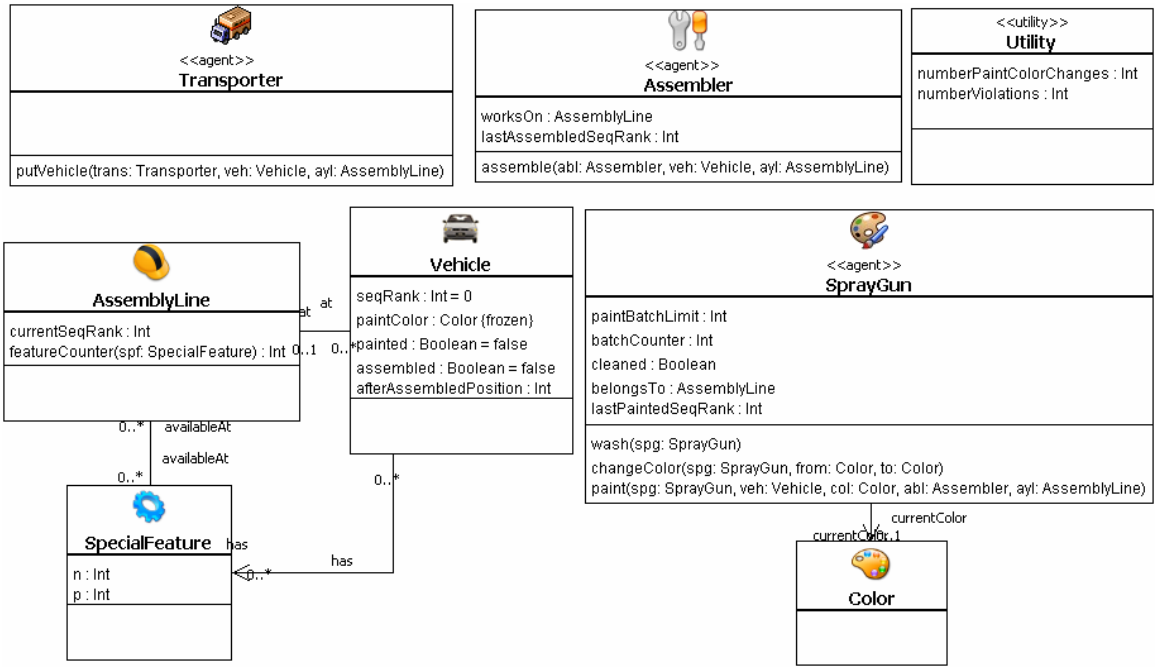


Figura 97 – *Diagrama de Classes* do domínio *Montagem Seqüencial de Carros em Linhas de Montagem*.

Neste modelo, a classe *Transporter* é capaz de executar a ação *putVehicle* que posiciona um veículo em uma linha para a pintura e a montagem. A classe *SprayGun* possui alguns

atributos e associações de destaque, sendo eles: *paintBatchLimit* (Int) que armazena o limite máximo de pinturas consecutivas com a mesma cor; *batchCounter* (Int) representa um contador de pinturas consecutivas; *lastPaintedSeqRank* (Int) identifica a posição do último veículo pintado; a associação *currentColor* representa a cor corrente do spray. Já a classe *Assembler* possui atributos como: *worksOn* (*AssemblyLine*) representando em qual linha o mesmo trabalho; e *lastAssembledSeqRank* (Int) que identifica a posição do último veículo montado pelo *Assembler*. A classe *SpecialFeature* representa as características especiais do domínio e esta possui, além dos atributos *n* e *p* (Int), uma associação *availableAt* com *AssemblyLine* identificando se a linha é capaz de realizar a montagem de tais características. A classe *AssemblyLine* representa a linha onde cada veículo será pintado e montado, por isso, esta possui um atributo chamado *currentSeqRank* (Int) que armazena a última posição da seqüência de carros que serão pintados e montados. Já a classe *Vehicle* possui também alguns atributos e associações de destaque, sendo eles: *seqRank* (Int) que identifica a ordem de entrada no *paint shop* de uma determinada linha; a associação *at* que identifica em qual linha o veículo se encontra após posicionado pelo *Transporter*; *paintColor* (Color) representando a cor que o veículo deverá ser pintado; os atributos de status *painted* e *assembled* (Booelan); e finalmente as associação *has* que identifica quais características especiais o veículo possui.

Como o objetivo neste domínio é minimizar as violações das razões N/P e o uso de solvente, conseqüentemente, o número de mudança de cor do spray, as variáveis globais *numberViolations* (Int) e *numberPaintColorChanges* (Int) são representadas na classe *Utility* com stereotype <<*utility*>>. Assim, o planejador deve procurar minimizar estas varáveis durante o processo de planejamento para a determinação da seqüência de pintura e montagem dos carros. Essas variáveis devem ser incrementadas a cada ação correspondente. Por exemplo, *numberPaintColorChanges* deve ser incrementado a cada vez que um agente *SprayGun* efetua uma troca de cor. Já a variável *numberViolations* pode ser incrementada

neste modelo a cada montagem realizada pelo agente *Assembler*. Para que esta última variável seja incrementada é preciso estabelecer uma forma de reconhecer tais violações nas razões N/P. Para isso, o presente modelo utiliza o atributo *featureCounter(spf: specialFeature)* (Int) da classe *AssemblyLine* que armazena o número de veículos (já montados na linha) que possuem a característica especial *spf* nos últimos P veículos já montados na linha. Esse atributo deve ser observado pelo planejador para que seu valor não ultrapasse o valor N da característica especial. A cada montagem de um veículo em uma linha o valor do atributo *featureCounter(spf: specialFeature)* é corrigido com o auxílio de um outro atributo da classe *Vehicle*, chamado *afterAssembledPosition* (Int). Com esse atributo é possível medir exatamente quantos veículos possuem a característica *spf* nos últimos P carros montados. A utilização destas variáveis e atributos podem ser melhor compreendida na representação das ações em OCL durante a modelagem dos aspectos dinâmicos.

Muitas das características citadas acima (atributos e associações) foram introduzidas ao modelo devido a sucessivas interações entre os processos de modelagem da estrutura estática e dos aspectos dinâmicos. Em relação aos aspectos dinâmicos, neste estudo de caso apenas quatro classes possuem aspectos dinâmicos relevantes a serem representados e analisados, sendo elas *Vehicle* (*Classe Recurso*), *Transporter*, *SprayGun*, e *Assembler* (*Classes Agentes*). Os *Diagramas de Estados* destas classes são mostrados nas figuras a seguir.

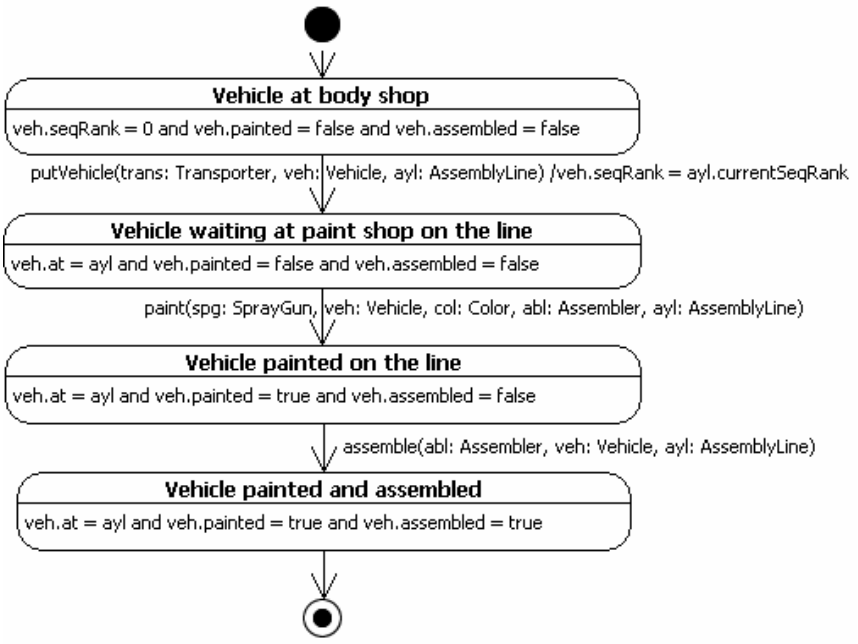


Figura 98 – *Diagrama de Estados* da classe *Vehicle*.

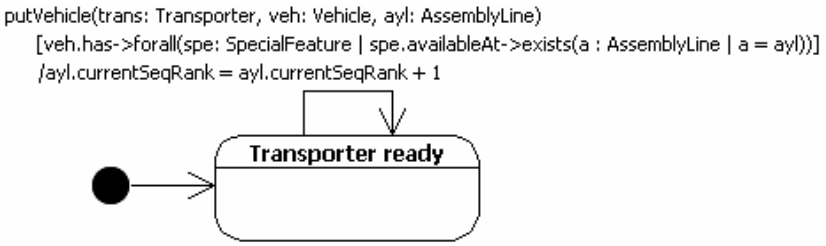


Figura 99 – *Diagrama de Estados* da classe *Transporter*.

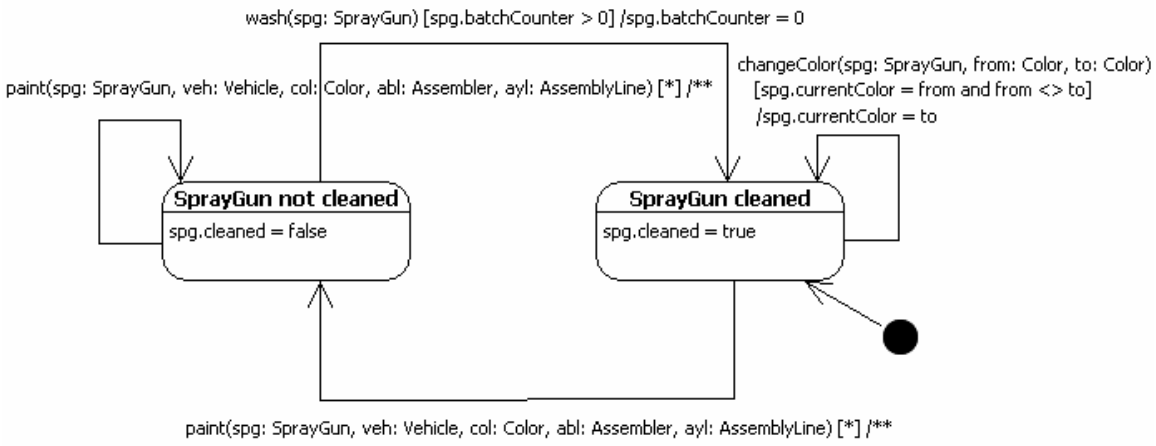


Figura 100 – *Diagrama de Estados* da classe *SprayGun*.
* e ** todas as expressões são apresentadas nas representações em OCL a seguir.

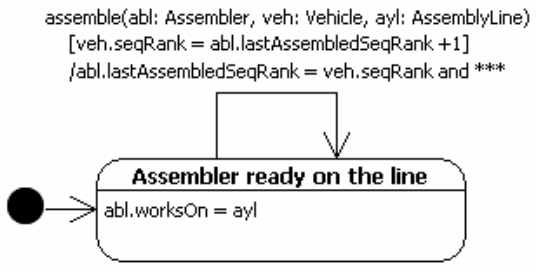


Figura 101 – *Diagrama de Estados* da classe *Assembler*.
*** todas as expressões são apresentadas nas representações em OCL a seguir.

Como resultado da união das expressões dos *Diagramas de Estados* das classes *Vehicle*, *Transporter*, *SprayGun*, e *Assembler* representados anteriormente, segue abaixo a representação das ações em OCL.

```
context Transporter::putVehicle(trans: Transporter, veh: Vehicle, ayl: AsemblyLine)
pre:
       -- Vehicle conditions
       veh.seqRank = 0 and veh.painted = false and veh.assembled = false and
       -- Transporter conditions
       veh.has->forall(spe: SpecialFeature |
            spe.availableAt->exists(a:AssemblyLine| a = ayl))

post:
       -- Vehicle conditions
       veh.at = ayl and veh.seqRank = ayl.currentSeqRank and
       -- Transporter conditions
       ayl.currentSeqRank = ayl.currentSeqRank + 1


context SprayGun::paint(spg: SprayGun, veh: Vehicle, col: Color, abl: Assembler,
                        ayl: AssemblyLine)
pre:

       -- SprayGun conditions
       spg.belongsTo = ayl and spg.currentColor = col and veh.paintColor = col and
       veh.seqRank = self.lastPaintedSeqRank + 1 and
       self.batchCounter < self.paintBatchLimit and
          -- check if the assembler already assembled the last car
       abl.worksOn = ayl and spg.lastPaintedSeqRank = abl.lastAssembledSeqRank and
       -- Vehicle conditions
       veh.at = ayl and veh.painted = false and veh.assembled = false

post:
       -- SprayGun conditions
       spg.batchCounter = spg.batchCounter + 1 and
       spg.lastPaintedSeqRank = veh.seqRank and spg.cleaned = false and
       -- Vehicle conditions
       veh.painted = true
```

```
context SprayGun::wash(spg: SprayGun)
pre:
       -- SprayGun conditions
       spg.cleaned = false and spg.batchCounter > 0

post:
       -- SprayGun conditions
       spg.cleaned = true and spg.batchCounter = 0



context SprayGun::changeColor(spg: SprayGun, from: Color, to: Color)
pre:
       -- SprayGun conditions
       spg.cleaned = true and spg.currentColor = from and not(from = to)

post:
       -- SprayGun conditions
       spg.currentColor = to and
          -- counting the number of changes for optimization
       numberPaintColorChanges = numberPaintColorChanges + 1



context Assembler::assemble(abl: Assembler, veh: Vehicle, ayl: AssemblyLine)
pre:
       -- Assembler conditions
       abl.worksOn = ayl and veh.seqRank = abl.lastAssembledSeqRank + 1 and
       -- Vehicle conditions
       veh.at = ayl and veh.painted = true and veh.assembled = false

post:
       -- Assembler conditions
       abl.lastAssembledSeqRank = veh.seqRank and
         -- set the number of assembled vehicle with each special feature on the
         --  line
       veh.has->forAll(spe:SpecialFeature |
              if( ayl.vehicle->exists(v:Vehicle | v.afterAssembledPosition = spe.p
                and not(v.has->exists(s:SpecialFeature | s = spe))) then
                     ayl.featureCounter(spe) = ayl.featureCounter(spe) + 1
              endif)
       and
       ayl.specialFeature->forall(spe:SpecialFeature |
              if( not(veh.has->exists(s:SpecialFeature | s = spe)) and
                ayl.vehicle->exists(v:Vehicle | v.afterAssembledPosition = spe.p and
                v.has->exists(s:SpecialFeature | s = spe)) and
                ayl.featureCounter(spe) > 0 ) then
                     ayl.featureCounter(spe) = ayl.featureCounter(spe) - 1
              endif)
       and
         -- check for violations
       veh.has->forAll(spe:SpecialFeature|
           if ( ayl.featureCounter(spe) > spe.n ) then
                    numberViolations = numberViolations + 1
           endif)
         -- correct the position of all cars that were assembled
       ayl.vehicle->forAll(v: Vehicle |
              if (v.assembled = true) then
                     v.afterAssembledPosition = v.afterAssembledPosition + 1
              endif)
       and
       veh.afterAssembledPosition = 1 and
       -- Vehicle conditions
       veh.assembled = true
```

De fato, a modelagem das ações em OCL no presente estudo de caso é mais complexa se comparada com os estudos de casos apresentados anteriormente, utilizando expressões mais elaboradas tanto nas pré-condições quanto nas pós-condições. Vale citar que um dos objetivos deste estudo de caso foi também mostrar o potencial da abordagem proposta no presente trabalho (principalmente em relação a UML e a OCL) na modelagem e análise não somente dos domínios clássicos, mas também daqueles de maiores níveis complexidade.

Finalmente, para completar o modelo do domínio é necessário ainda representar os agentes e recursos do mesmo. Assim, conforme ilustra a Figura 94, pode-se dizer que o conjunto de agentes é composto pelos objetos *t1* da classe *Transporter*, *s1* da classe *SprayGun*, e *a1* da classe *Assembler*. Já o conjunto de recursos é composto pelos objetos: *vehicle1, vehicle2, vehicle3, vehicle3, vehicle4, vehicle5, vehicle6, vehicle* e *vehicle8* da classe *Vehicle* representando os veículos que devem ser pintados e montados no dia D*; vehicleLD1, vehicleLD2, vehicleLD3, vehicleLD3, vehicleLD4* e *vehicleLD5* da classe *Vehicle* representando os veículos já montados no dia D-1 (estes também são considerados durante as possíveis violações); *ayl1* da classe *AssemblyLine* representando a única linha disponível no domínio; *spf1, spf2 e spf3* da classe *SpecialFeature* representando as três características especiais disponíveis (teto solar, ar condicionado e rodas especiais); e, finalmente, as duas cores disponíveis na fábrica *red* e *blue* da classe *Color*.

5.3.3. Análise do Modelo do Domínio

Como visto no Capítulo 3, com base nos *Diagramas de Estados* gerados na modelagem dos aspectos dinâmicos é possível realizar as análises *Análise Modular* e *Análise de Interfaces* utilizando as Redes de Petri. Como visto anteriormente, as classes relevantes a serem analisadas são: *Vehicle*, *Transporter*, *SprayGun* e *Assembler*. Neste tópico será apresentada

convenientemente apenas a *Análise de Interfaces*. Assim, a Figura 102 demonstra a *Análise de Interfaces* do modelo.

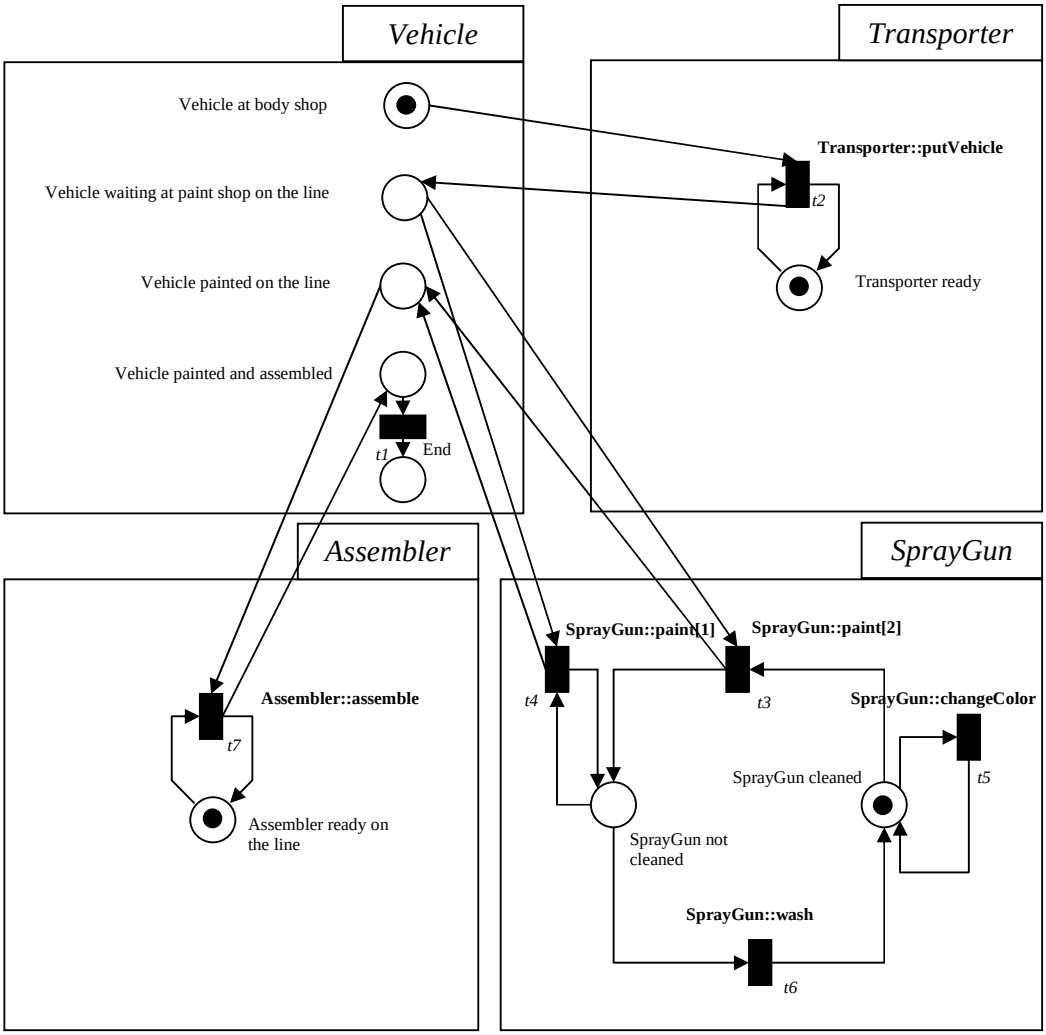


Figura 102 – *Análise de Interfaces* do comportamento das classes *Vehicle, Transporter, SprayGun* e *Assembler.*

5.3.4. Modelagem do Problema

No presente estudo de caso do domínio *Montagem Seqüencial de Carros em Linhas de Montagem* o problema de planejamento consiste em fazer com que os agentes executem as devidas ações de modo a pintar e montar todos os carros do dia D minimizando o número de

mudanças de cor o número de violações das razões N/P das características especiais disponíveis. Os estados inicial e objetivo do problema são apresentados utilizando os *Snapshots* da Figura 103 e da Figura 104 (onde todos os veículos do dia D devem estar pintados e montados), respectivamente.

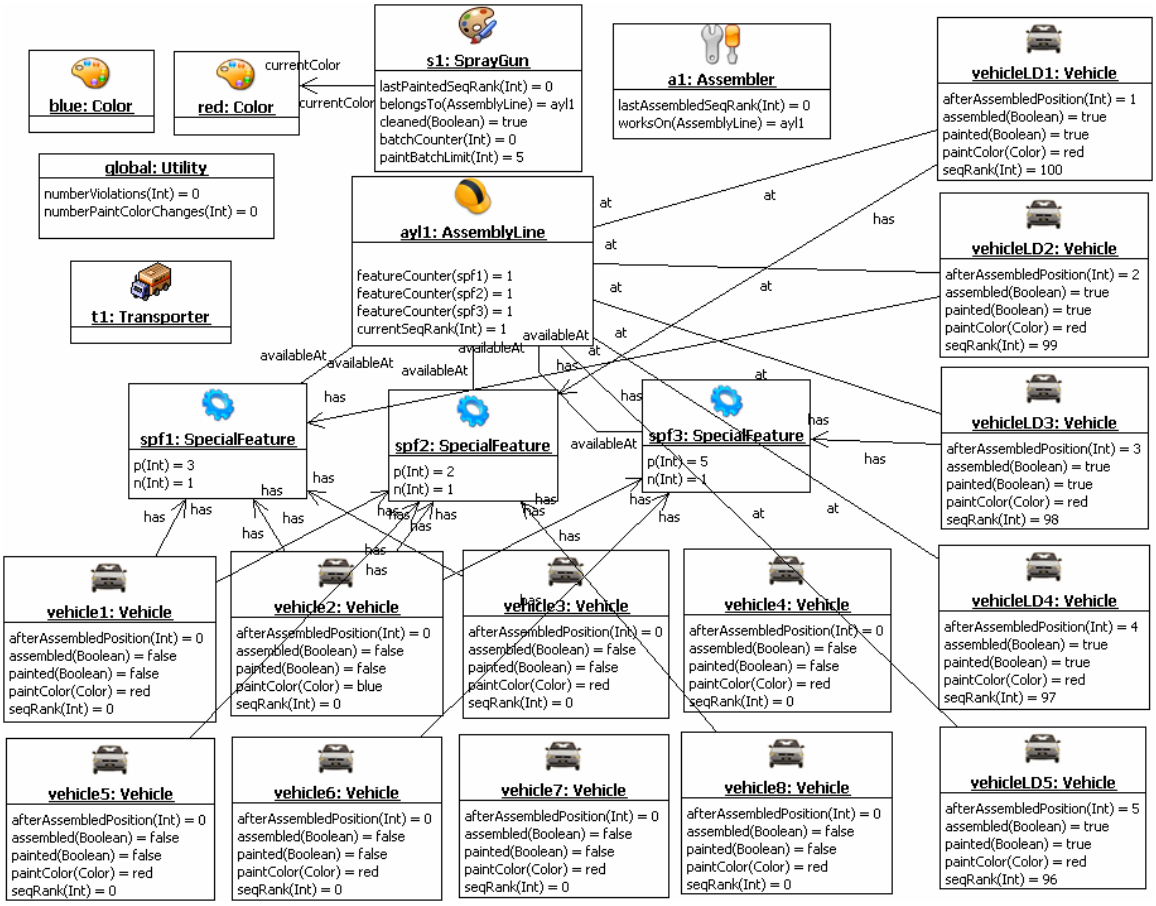


Figura 103 – *Snapshot Inicial* de um problema de planejamento no *Montagem Seqüencial de Carros em Linhas de Montagem.*

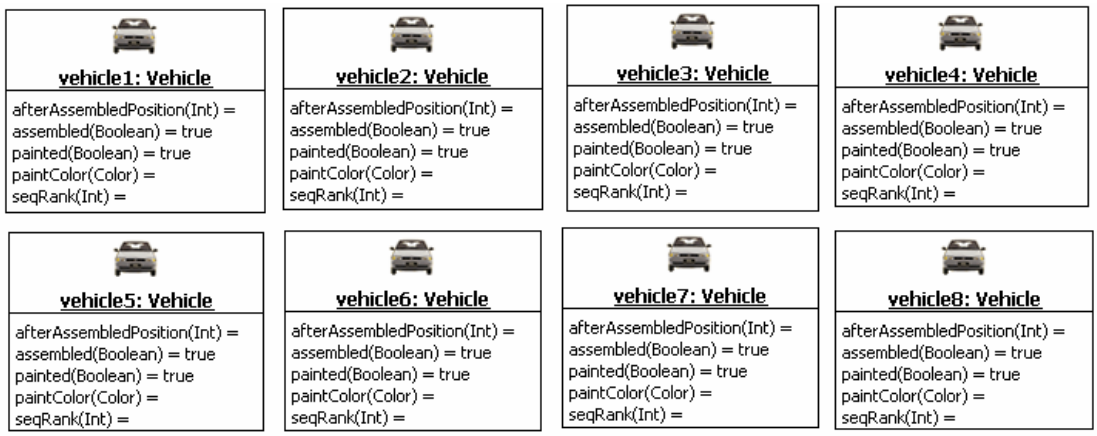


Figura 104 – *Snapshot Objetivo* de um problema de planejamento no *Montagem Seqüencial de Carros em Linhas de Montagem.*

No próximo tópico é apresentada uma possível solução, fornecida por um algoritmo geral de planejamento, para o problema em questão.

5.3.5. Teste com Planejador

Neste estudo de caso, a fase de teste com planejador, para verificação e refinamento do modelo, foi realizada também com o algoritmo Metric-FF (HOFFMANN, 2003). Para que o Metric-FF fosse utilizado na resolução do problema, foi necessária uma tradução para PDDL do domínio e do problema de planejamento. Devido ao fato deste estudo de caso utilizar expressões OCL (principalmente nas ações) de maior complexidade, o itSIMPLE não foi capaz de traduzir o modelo para a linguagem PDDL, pois o processo de tradução disponibilizado pela ferramenta possui limitações (conforme demonstradas no Capítulo 4). De fato, o presente estudo de caso foi realizado com o objetivo de levantar novas necessidades e futuros trabalho também para a evolução da ferramenta itSIMPLE.

Mesmo com a limitação da ferramenta em relação à tradução, o modelo foi traduzido manualmente (mantendo a coerência) para a PDDL e a esta representação foi acrescentada uma métrica já que o planejador deve minimizar o número de violações das razões N/P, bem

como o número de trocas de cor realizadas pela pistola (ou seja, minimizar as variáveis globais *numberViolations* e *numberPaintColorChanges,* respectivamente). A métrica adicionada ao modelo (no arquivo do problema em PDDL) foi a seguinte:

```
(:metric minimize (+ (numberViolations) (numberPaintColorChanges)))
```

Com a métrica acima o planejador procura resolver o problema minimizando as varáveis de interesse neste estudo de caso. As representações em PDDL do modelo do domínio *Montagem Seqüencial de Carros em Linhas de Montagem* e do problema de planejamento em questão podem ser encontradas novamente anexadas a este trabalho no Anexo A – Modelos de Domínios em PDDL. Vale citar novamente que as restrições em PDDL foram ignoradas.

Para este estudo de caso, o planejador Metric-FF resolveu o problema, representado em PDDL, com um plano formado por 26 passos. O plano-solução, fornecido pelo Metric-FF, para o problema de planejamento, ilustrado na Figura 103 e na Figura 104, é mostrado a seguir.

0: PUTVEHICLE T1 VEHICLE4 AYL1
1: PUTVEHICLE T1 VEHICLE1 AYL1
2: PUTVEHICLE T1 VEHICLE6 AYL1
3: PUTVEHICLE T1 VEHICLE5 AYL1
4: PUTVEHICLE T1 VEHICLE3 AYL1
5: PUTVEHICLE T1 VEHICLE8 AYL1
6: PUTVEHICLE T1 VEHICLE2 AYL1
7: PUTVEHICLE T1 VEHICLE7 AYL1
8: PAINT S1 VEHICLE4 RED AYL1 A1
9: ASSEMBLE A1 VEHICLE4 AYL1
10: PAINT S1 VEHICLE1 RED AYL1 A1
11: ASSEMBLE A1 VEHICLE1 AYL1
12: PAINT S1 VEHICLE6 RED AYL1 A1
13: ASSEMBLE A1 VEHICLE6 AYL1
14: PAINT S1 VEHICLE5 RED AYL1 A1
15: ASSEMBLE A1 VEHICLE5 AYL1
16: PAINT S1 VEHICLE3 RED AYL1 A1
17: ASSEMBLE A1 VEHICLE3 AYL1
18: PAINT S1 VEHICLE8 RED AYL1 A1
19: ASSEMBLE A1 VEHICLE8 AYL1
20: PAINT S1 VEHICLE2 RED AYL1 A1
21: WASH S1
22: CHANGECOLOR S1 RED BLUE
23: ASSEMBLE A1 VEHICLE2 AYL1
24: PAINT S1 VEHICLE7 BLUE AYL1 A1
25: ASSEMBLE A1 VEHICLE7 AYL1

Na solução apresentada pelo Metric-FF é possível perceber que todos os carros passam pela pintura e em seguida pela montagem de forma a finalizar os oito carros a serem entregues aos clientes. Outro aspecto interessante do plano-solução acima é que os carros são agrupados por cor na seqüência de montagem e que a pistola é lavada e esta troca de cor durante a

pintura dos carros. Para melhor visualizar o plano e verificar se as razões não foram violadas na seqüência proposta pelo planejador, a Tabela 20 a seguir mostra a ordem de entrada na linha dos carros juntamente com suas características especiais, onde X, Y e Z representam as características especiais do domínio *spf1, spf2* e *spf3,* respectivamente.

Tabela 20 - Plano fornecido pelo Metric-FF para um problema de planejamento do domínio *Montagem Seqüencial de Carros em Linhas de Montagem.*

D-1 (últimos veículos)					Dia de Produção D							
vLD5	**vLD4**	**vLD3**	**vLD2**	**vLD1**	**v4**	**v1**	**v6**	**v5**	**v3**	**v8**	**v7**	**v2**
red	red	red	red	red	red	red	red	red	red	red	red	blue
			X			X			X			X
				Y		Y		Y		Y		Y
		Z					Z					Z

O plano representado na Tabela 20 demonstra que todas as razões foram respeitadas fazendo com que o valor de violações seja zero. Analisando o plano, o projetista pode verificar como os recursos e agentes são utilizados e se suas utilizações se fazem adequada.

5.3.6. Observações

Um dos principais objetivos da modelagem e análise deste domínio foi investigar o potencial do *Ambiente* proposto frente aos problemas de planejamento com características mais complexas e mais realísticas identificando novas necessidades e evoluções da ferramenta e com isso levantando futuros trabalhos. Neste estudo de caso o modelo do domínio resultante possui características que vão além do que o itSIMPLE consegue tratar (principalmente no que tange as ações em OCL), mas não além do que é proposto no *Ambiente* apresentado neste trabalho. Além disso, é possível perceber que representar tal modelo diretamente em PDDL seria uma tarefa com excessivos refinamentos e ajustes para se chegar a um modelo passível

de teste com os planejadores, principalmente se o projetista não tiver profundos conhecimentos da linguagem PDDL e das limitações dos planejadores.

A utilização dos conceitos do *Ambiente*, bem como a abordagem orientada a objetos, e do foco na reutilização do conhecimento durante a modelagem do domínio *Montagem Seqüencial de Carros em Linhas de Montagem* fez com que o modelo gerado neste estudo de caso fosse bem abrangente, ou seja, o projetista pode criar configurações interessantes de domínios e problemas de planejamento com pequenas variações no modelo apresentado neste estudo de caso. Um exemplo interessante seria analisar um domínio (e seus respectivos problemas) com mais de uma linha de montagem de capacidade de montagem de características especiais diferentes. De fato, domínios como estes se tornam mais desafiantes, principalmente para os planejadores, e seus modelos são facilmente derivados do modelo apresentado neste trabalho, bastando apenas realizar modificações no conjunto de agentes e recursos, bem como os estados inicial e objetivo dos problemas de planejamento. Mais informações sobre presente estudo de caso podem ser encontradas em (VAQUERO et al., 2006).

As necessidades geradas a partir deste estudo de caso, bem como os anteriores, para evolução do *Ambiente* e da ferramenta serão apresentadas no próximo capítulo, no tópico referente aos trabalhos futuros deste trabalho.

Capítulo 6

6. Conclusões e Trabalhos Futuros

6.1. Conclusões

Neste trabalho foi apresentado um *Ambiente* de design que integrada processos de análise de requisitos, especificação, modelagem, análise e testes de modelos de domínios de planejamento. Neste ambiente são utilizadas linguagens e representações bem conhecidas tanto na Engenharia (principalmente a de Software e a de Requisitos) quando especificamente na área de pesquisa de Planejamento Automático em Inteligência Artificial, tais como UML, OCL, Redes de Petri, PNML e PDDL, para facilitar o trabalho do projetista com tais domínio. O presente trabalho apresentou também a ferramenta chamada itSIMPLE (*Integrated Tools Software Interface for Modeling Planning Environment*), implementada em Java™, que viabiliza a utilização do *Ambiente* e de seus conceitos.

Muitos pesquisadores utilizam a PDDL como linguagem inicial para a representação dos domínios de planejamento, mas este linguagem não foi projetada para ser utilizada nas fases iniciais de projetos (fases de requisitos e design) tornando principalmente o trabalho de modelagem muito exaustivo e muitas vezes inviável para domínios com níveis elevados de complexidade, ou seja, domínio reais de planejamento. Diante disso, este trabalho mostrou a utilização de linguagens mais usuais nas fases inicias de um ciclo de vida de projeto para auxiliar o projetista principalmente na realização de processos disciplinados de modelagem e análise de domínio reais de planejamento. Além da utilização de linguagens comuns, o

trabalho mostrou a possível integração entre as linguagens para facilitar a realização dos diversos processo de design deste tipo de domínio. Esta integração, presente na ferramenta itSIMPLE, faz com que o projetista análise um mesmo modelo em diversos pontos de vistas melhorando assim a qualidade do modelo gerado (principalmente devido aos sucessivos refinamentos).

Um dos pontos importantes na utilização da ferramenta é que a equipe de design pode trabalhar em um ambiente que favorece a uniformidade do conhecimento sendo representado e modelado, ou seja, os diferentes pontos de vistas dos participantes (especialistas, usuários, pesquisadores, entre outros) podem ser explorados em um único ambiente de forma a tornar o modelo o mais coerente e correto possível. Este fato é fortalecido com a construção de diagramas e a utilização de imagens associadas aos seus elementos facilitando a comunicação da equipe que possui diferentes níveis de conhecimento sobre o domínio sendo investigado.

Os estudos de casos apresentados neste trabalho mostram o potencial da ferramenta, e principalmente do ambiente, nos processos de modelagem e análise de modelos de domínios de planejamento. De fato, após realizar todo o processo proposto o projetista possui um modelo mais claro com grandes possibilidades de reutilização (devido, entre outros fatores, a abordagem orientada a objetos). Com os modelos resultantes a equipe consegue entender e verificar o domínio sem profundos conhecimentos em uma linguagem especifica. Assim, a PDDL é utilizada apenas em algumas fases do design como, por exemplo, os testes de modelos com planejadores, e não para as fases iniciais de requisitos, modelagem e análise.

Além da ferramenta desenvolvida, o presente trabalho gerou produções acadêmicas relevantes para a área de pesquisa de Planejamento Automático e principalmente para a área da Engenharia do Conhecimento aplicada ao Planejamento Automático. As publicação, tanto internacionais quanto nacionais, resultantes deste trabalho são (VAQUERO; TONIDANDEL; SILVA, 2005) (VAQUERO, et al., 2006) (SANTOS; SILVA; VAQUERO, 2005),

(TONIDANDEL; VAQUERO; SILVA, 2006), o que demonstra o impacto do trabalho perante a comunidade de pesquisa em Planejamento Automático.

No próximo tópico são discutidos alguns trabalhos futuros, evidenciados durante a elaboração dos conceitos apresentados neste trabalho, que podem de alguma forma potencializar o presente projeto, bem como sua linha de pesquisa.

6.2. Trabalhos Futuros

Acredita-se que os trabalhos que deverão ser desenvolvidos futuramente podem agregar valor ao trabalho apresentado e, conseqüentemente, potencializar a área de pesquisa de Planejamento Automático não só no cenário nacional, mas também no cenário internacional. Alguns trabalhos que deverão ser desenvolvidos como seqüência ao presente projeto são:

1. Desenvolvimento de um mecanismo de validação e verificação automática das Redes de Petri geradas no *Ambiente*. Estas verificações e validações das redes podem gerar informações importantes que devem ser utilizadas durante o processo de planejamento automático de modo a reduzir o espaço de busca pela solução. Poderá ser realizado um estudo sobre os ganhos na modelagem com a realização destes processos de verificação e validação dos modelos;
2. Estudo de outros diagramas da UML para identificar como estes podem contribuir no processo de modelagem dos domínio de planejamento;
3. Busca de novas linguagem a serem utilizadas na modelagem e análise de domínios de planejamento e agregá-las a ferramenta itSIMPLE como, por exemplo, a linguagem Z (SPIVEY, 1992) ou a linguagem B (ABRIAL, 1996);
4. Realização de estudos de casos com domínios reais onde os níveis de complexidade são maiores se comparados com os domínios clássicos, para conceber a evolução da ferramenta com novas funcionalidades de auxílio ao projetista;

5. Desenvolvimento de mecanismos de avaliação, análise, visualização e simulação dos planos gerados pelos planejadores durante as fases de testes do modelos. Isto pode ajudar no processo de refinamento do modelo durante o processo de testes com planejadores;
6. Integração da ferramenta itSIMPLE com uma biblioteca de planejadores que podem ficar a disposição do projetista para a análise e o refinamento do modelo;
7. Desenvolvimento de algoritmos de planejamento capazes de tratar todas as informações geradas durante o processo de modelagem e análise na ferramenta itSIMPLE de forma a utilizar o máximo de conhecimento possível.

Finalmente, com o apóio dos conceitos do *Ambiente* e da ferramenta itSIMPLE e com as evoluções da pesquisa e do desenvolvimento tanto do presente trabalho quanto da própria área do Planejamento Automático em IA, acredita-se na possibilidade da realização de processos de design sólidos e bem explorados no desenvolvimento de sistemas reais e complexos de planejamento automático.

Referências Bibliográficas

ABRIAL, J.R. *The B-Book, Assigning programs to meanings*, Cambridge University Press. 1996.

BACCHUS, F. *The Power of Modeling – a Response to PDDL2.1*. Journal of Artificial Intelligence Research 20:125 - 132. 2003.

BARESI, L. and PEZZÈ, M. *On Formalizing UML with High-Level Petri Nets*. In G. Agha and F. De Cindio (ens.) Concurrent Object-Oriented Programming and Petri Nets (a special volume in the Advances in Petri Nets series); pages 271-300. Number 2001 of Lecture Notes in Computer Science – Spring Verlag. 2001.

BARRET, A. and WELD, D. *Partial Order Planning: Evaluation Possible Efficiency Gains*. Washington: Dept. of Computer Science and Engineering, University of Washington. Technical Report 92-05-01. 1992.

BERTOLIN, J.C.G. *Uma Análise de Técnicas da Psicologia para a Elicitação de Requisitos de Software*. In: Semana acadêmica do CPGCC, 1998, Porto Alegre. Semana Acadêmica do CPGCC. Porto Alegre, RS: CPGCC da UFRGS, 1998. v. 1. p. 117-120.

BILLINGTON, J., CHRISTENSEN, S., VAN HEE, K., KINDLER, E., KUMMER, O., PETRUCCI, L., POST, R., STEHNO, C. and WEBER, M. *The Petri Net Markup Language: Concepts, Technology, and Tools*. In Proceedings of the ICATPN 2003 to be held in Eindhoven, Netherlands, June 2003. Disponível em http://www.informatik.hu-berlin.de/top/pnml/download/about/PNML_CTT.pdf.

BLUM, A. L. and FURST, M. L. *Fast Planning Through Planning Graph Analysis*. Artificial Intelligence, 90:281-300. 1995.

BLYTHE, J. and RATNAKAR, V. *Helping end users modify procedures by instruction.* ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

BODDY, M. *Imperfection Match: PDDL2.1 and Real Applications.* Journal of Artificial Intelligence Research 20:133 - 137. 2003.

BOEHM, B.W. *Software risk management*. IEEE Computer Society Press: Washington. 1989.

BONET, B. and GEFFNER, H. *Planning as heuristic search: new results*. In Proceedings EUROPEAN CONFERENCE ON PLANNING – ECP'99., Durham Durham, UK. 1999.

BORRAJO, D., FERNÁNDEZ, S., FUENTETAJA, R., ARIAS J. D. and VELOSO, M. *Tool for automatically acquiring control knowledge for planning*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

BRAY, T., PAOLI, J., MCQUEEN, C.M., MALER, E. and YERGEAU, F. *Extensible Markup Language (XML) 1.0 – Third Edition*. Available in: http://www.w3.org/TR/REC-xml/. 2004.

CHAPMAN, D. *Planning for Conjunctive Goals*. Artificial Intelligence, 32:333-377. 1987.

CHEUNG, K.S., CHOW, K.O. and CHEUNG, T.Y. *Deriving scenarios of object interaction through Petri net*. In: Proceedings of Technology of Object-Oriented Language and System. 1998.

CHRISTENSEN, S. and PETRUCCI, L. *Modular analysis of Petri nets*. The Computer Journal 2000 43(3):224-242. 2000.

CONROW E.H. and SHISHIDO P.S. *Implementing Risk Management on Software Intensive Projects*. IEEE Software, v. 14, n. 3 (mai/jun), p. 83-89. 1997.

CURRIE, K. and TATE, A. *O-Plan: the Open Planning Architecture*. Artificial Intelligence Vol. 52, pp. 49-86, Elsevier. 1991.

D'SOUZA, F.D. and WILLS, A.C. *Object, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesley. United States of America and Canada. 1999.

DALEY, P., FRANK, J., IATAURO, M., MCGANN, C. and TAYLOR, W. *PlanWorks: A debugging environment for constraint based planning systems*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

DÖLL, L.M. *Proposta de uma Metodologia para Modelagem da Dinâmica de Sistemas Orientados a Objetos Usando Redes de Petri Predicado/Transição*. Dissertação de Mestrado apresentada ao CPGEI-CEFET-PR, Curitiba-PR. 2003.

EDELKAMP, S. and HOFFMANN J. *PDDL 2.2: The Language for Classical Part of the 4th International Planning Competition.* Technical Report, Fachbereich Informatik and Institut für Informatik. Germany. 2004.

EDELKAMP, S. and MEHLER, T. *Knowledge acquisition and knowledge engineering in the ModPlan workbench*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

ERNST, G. and NEWELL, A. *GPS: A Case Study in Generality and Problem Solving*. New York: Academic Press. 1969.

EROL, K., HENDLER, J. and NAU, D.S. *HTN Planning: Complexity and Expressivity*. In Proceedings of International Conference on Artificial Intelligence – AAAI-94. pp. 1123-1128. 1994.

FERNANDES, D. B. *Análise de Sistemas Orientada ao Sucesso: Por que os projetos atrasam?* Editora Ciência Moderna.1ª Edição. 2005.

FIKES, R. E. and NILSSON N. J. *STRIPS: A new approach to the application of theorem proving to problem solving*. Artificial Intelligence, 2:189–208. 1971.

FOX, M. and LONG, D. *PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. Journal of Artificial Intelligence Research 20:61-124. 2003.

GEFFNER, H. *PDDL2.1: Representation vs. Computation*. Journal of Artificial Intelligence Research 20:139-140. 2003.

GENRIGH, H.J. and LAUTENBACH, K. *System modeling with high-level Petri nets*. Theoret. Comp. Sci. Vol. 13, pp. 109-136. 1981.

GEREVINI, A. and LONG, D. *Plan Constraints and Preferences in PDDL3 – The Language of Fifth International Planning Competition*. Technical Report, Department of Electronics for Automation, University of Brescia, Italy, August 2005.

GHALLAB M., NAU D., and TRAVERSO P. *Automated Planning: Theory and Practice*. Morgan Kaufmann. 2004.

GIL, Y., VELOSO, M., CHIEN, S.A., MCDERMOTT, D. and NAU, D. *Symposium Preface. In Planning and Learning: On to Real Applications*. Papers from 1994 AAAI Fall Symposium, number FS-94-01. Publicado por AAAI Press, American Association for Artificial Intelligence, ISBN 0-929280-75-X. 1995.

GOSLING, J., JOY, B, STEELE, G. and BRACHA, G. *The Java™ Language Specification – Second Edition*. The Java™ Series. Addison-Wesley. 2000.

GOUGH, J. *XPDDL 0.1b: A XML version of PDDL*. Disponível em http://www.cis.strath.ac.uk/~jg/XPDDL/. 2004. Acessado em dezembro de 2006.

GREEN, C. *Application of Theorem Proving to Problem Solving*, In Proceedings of the First International Joint Conference on Artificial Intelligence – IJCAI-69. pp.219-239. Menlo Park, CA. 1969.

HELMERT, M. *A planning heuristic based on causal graph analysis*. In Koenig, S., Zilberstein, S., & Koehler, J. (Eds.), Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04), pp. 161–170, Whistler, Canada. Morgan Kaufmann. 2004.

HOFFMANN, J. *Extending FF to numerical state variables*. In Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02), pages 571-575, Lyon, France. 2002.

HOFFMANN, J. *The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables*. Journal of Artificial Intelligence Research. Accepted for special issue on the 3rd International Planning Competition. 2003.

HOFFMANN, J., NEBEL, B. *The FF planning system: fast plan generation through heuristic search*. Journal of Artificial Intelligence Research, 14:253-302. 2001.

HSU, C.W., WAH, B.W., HUANG, R. and CHEN, Y. X. *New Features in SGPlan for Handling Soft Constraints and Goal Preferences in PDDL3.0*. Proc. Fifth International Planning Competition, International Conf. on Automated Planning and Scheduling ICAPS'06, June 2006.

IEEE Std. 830, *IEEE Guide to Software Requirement Specification*. The Institute of Electrical and Electronics Engineers. New York, 1984.

IEEE. *IEEE Software: Measurement Based Process Improvement*. July 1991, v.11(4).

JENSEN, K. *Coulored Petri nets and the invariant-method*. Theoret. Comp. Sci. Vol. 14, pp. 317-336. 1981.

JOSLIN, D. and POLLACK, M.E. *Is 'early commitment' in plan generation ever a good idea?* In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI), pages 1188-1193, Portland, OR, 1996.

KAUTZ, H. and SELMAN, B. *Pushing the envelope: planning, propositional logic, and stochastic search*. In: Proceedings of the 13th Natl. Conf. on Artificial Intelligence (AAAI-96), Portland, OR, pp. 1194-1201. 1996.

KAUTZ, H. and SELMAN, B. *Unifying SAT-based and graph-based planners*. In Proceedings of the 15th International Joint Conference on Artificial Intelligence - IJCAI-99. 1999.

KAUTZ, H., ROZNYAI, D., TEYDAYE-SAHELI, F., NEPH, S. and LINDMARK, M. *SATPLAN: Planning as satisfability*. 2004. Available at: http://www.cs.washington.edu/homes/kautz/satplan/.

KINDLER, E. and WEBER, M. *A Universal Module Concept for Petri Nets. An Implementation-Oriented Approach*. Informatik-Bericht Nr. 150, Humboldt-Universität zu Berlin, April 2001.

KHATIB, L., FRANK, J., SMITH, D., MORRIS, R. and DUNGAN, J. *Interleaved Observation Execution and Rescheduling on Earth Observing Systems*. In Proc. of the ICAPS-03 Workshop on "Plan Execution", Trento, Italy, 2003.

KOEHLER, J.and SCHUSTER, K. *Elevator control as Planning Problem*. In S. Chien, S Kambhampati, and C. Knoblock, editores, Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling, pages 331-338. AAAI Press, Menlo Park. 2000.

KOTONYA, G. and SOMMERVILLE, I. *Requirements engineering: processes and techniques*. Chichester: John Wiley. 1998.

KOWALSKI, R. A. and SERGOT, M. *A logic-based calculus of events*. New Generation Computing. 4:67-95. 1969.

MCALLESTER, D. and ROSENBLITT, D. *Systematic Nonlinear Planning*. American Association for Artificial Intelligence – AAAI. 1991.

MCCALLUM, T. *Position paper on the infrastructure supporting the development of PDDL*. ICAPS 03 – Workshop on PDDL. 2003.

MCCARTHY, J. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts. 1962.

MCCLUSKEY, T. L *Knowledge Engineering: Issues for the AI Planning Community*. Proceedings of the AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning, Toulouse, France, April 2002.

MCCLUSKEY, T. L. *PDDL2.1: A Language with a Purpose?*. ICAPS 03 – Workshop on PDDL. 2003.

MCCLUSKEY, T. L. *Report on the PLANET AGM meeting on ICKEP*. PLANET News – Newsletter of the European Network Excellence in AI Planning. Issue No. 6, May 2003 – page 23. Printed in Ulm, Germany. 2003.

MCCLUSKEY, T. L.; ALER, R.; BORRAJO, D.; HASLUM, P.; JARVIS, P.; I.REFANIDIS; and SCHOLZ, U. *Knowledge Engineering for Planning Roadmap*. 2003. Available at http://scom.hud.ac.uk/planet/.

MCCLUSKEY T. L. and PORTEOUS, J. M. *Two Complementary Techniques in Knowledge Compilation for Planning*. In Proceedings of the 3rd International Workshop on Knowledge Compilation and Speedup Learning. 1993.

MCCLUSKEY T. L., PORTEOUS, L. M., NAIK, J., TAYLOR, C. N. and JONES S. *An Requirements Capture Method and its use in Air Traffic Control Application*. Software-Practice and Experience, 25:47-41. 1995.

MCDERMOTT, D. and HENDLER, J. *Planning: What it is, What it could be, An Introduction to the Special Issue on Planning and Scheduling*. Artificial Intelligence, 76:1-16. 1995.

MCDERMOTT, D. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee. 1998.

MCDERMOTT, D. PDDL2.1 - *The Art of the Possible?. Commentary on Fox and Long*. Journal of Artificial Intelligence Research 20:145 - 148. 2003.

MERLIN, P. *A Study of the Recoverability of Computer Systems*. Thesis, Department of Computer Science, University of California, Irvine, 1974.

MINTON, S. *Learning effective search control knowledge: an explanation approach.* Kluwer Academic Publishers. 1988.

MURATA, T. *Petri Nets: Properties, Analysis and Applications*. In Proceedings of IEEE, v. 77, n. 4, pp. 541-580. 1989.

MUSCETTOLA, N.; NAYAK, P. P.; PELL, B.; and WILLIAMS, B. C. *Remote Agent: To Boldly Go Where No AI System Has Gone Before*. Artificial Intelligence 103(1-2):5-48. 1998.

MYRES, K., and WILKINS, D. *The Act-Editor User's Guide: A Manual for version2.2*. SRI International, Artificial Intelligence Center. 1997.

NGUYEN, A. *Challenge ROADEF'2005: Car Sequencing Problem*. Renault - Information System and Technologies - Advanced Studies. France. 2003.

NEWELL, A. and SIMON, H. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall. 1972.

OMG - Object Management Group. *Unified modeling language specification: version 1.4*. URL http://www.omg.org/uml. 2001.

OMG - Object Management Group. *OCL 2.0 – Object Constraint Language*. URL http://www.omg.org/cgi-bin/doc?ptc/2003-10-14. 2003.

OMG - Object Management Group. *XMI 2.1 – XML Metadata Interchange (XMI)*. URL http://www.omg.org/cgi-bin/doc?formal/2005-09-01. 2005.

PAWSON, R. and MATTHEWS R. *Naked Objects. Naked Objects Group Ltd*. URL http://www.nakedobjects.org. 1999.

PEDNAULT, E. *ADL: Exploring the middle ground between STRIPS and situation calculus*. In Proc. 1st Intl. Conf. on Principles of Knowledge Representation and Reasoning, pp. 324-332. 1989.

PENBERTHY, J. S. and WELD, D. *UCPOP: A sound, complete, partial order planner for ADL*. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pp. 103-114 Boston, MA. Morgan Kaufmann. 1992.

PETERSON, J. L. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall. 1981.

PALUDETTO, M. and DELOUTOUR, J. *UML et les réseaux de Petri: vers une sémantique des modeles dynamiques et une méthodologie de développement des systèm temps reel*. In: L'Object. 1999.

PMI – Project Management Institute. *PMBOK – Project Management Body of Knowledge*. Tradução livre do PMBOK 2000. V1.0, PMI/MG, 2002.

RAMCHANDANI, C. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Massachusetts Institute of Technology, Project MAC, Technical Report 120, February. 1974.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F. and LORENSEN, W. *Object-Oriented Modelling and Design*. Pretince Hall, 1991.

SACERDOTI, E. *Planning in a Hierarchy of Abstraction Spaces*. Artificial Intelligence, 5:115-135. 1974.

SACERDOTI, E. *A Structure for Plans and Behaviors*. New York: North Holland. 1977.

SALDHANA, J. A. and SHATZ, S. M. *UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis*. In: Twelfth International Conference on Software Engineering and Knowledge Engineering. July 6-8. Chicago, IL, USA. 2000.

SANTOS, E. A. *Verificação de Requisitos de Sistemas Utilizando Redes de Petri*. Dissertação de Mestrado, Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos, EPUSP. 2002.

SANTOS, E. A., SILVA, J. R. and VAQUERO, T. S. *Specification and Analysis for Automated Flexible Manufacturing*. In Proceedings of COBEM 2005 – 18th International Congress of Mechanical Engineering, 6 to 11 November 2005, Ouro Preto, Minas Gerais, Brazil. 2005.

SILVA, J.R. and DEL FOYO, P.M.G. *Towards a Unified View of Petri Nets and Object Oriented Modeling*. To appear in 17th. Int. Congress of Mechanical Engineering, São Paulo, December 2003.

SILVA, J R. and SANTOS, E. A. *Applying petri nets to requirements validation* In: *IFAC Symposium on Information Control Problems in Manufacturing. Salvador, 2004*. INCOM'04 : Abstracts.Salvador : IFAC, p. 1. 2004.

SIMPSON, R. M. *GIPO Graphical Interface for Planning with Objects*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

SIMPSON, R.M, T. L. MCCLUSKEY, W. ZHAO, R. S. AYLETT and DONIAT C. *GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning*. Proceedings, 2001 European Conference on Planning, Toledo, Spain. 2001.

SMITH, D. *The Case for Durative Actins: A commentary on PDDL 2.1*. Journal of Artificial Intelligence Research 20:149 – 154. 2003.

SPIVEY, M.J. *The Z Notation: A Reference Manual.* (Second Edition) Prentice-Hall, 1992.

STEFICK, M. *Planning with constraints*. Artificial Intelligence, 16:111-140. 1981.

STUDER, R., BENJAMINS, V., and FENSEL, D. *Knowledge engineering: Principles and methods*. IEEE Transactions on Data and Knowledge Engineering, 25:161 - 97. 1998.

TATE, A. *Project planning using a hierarchic nonlinear planner*. Department of Artificial Intelligence Research Rep. No. 25, University of Edinburgh, Edinburgh. 1976.

TATE, A. *Generalizing project networks*. In Proceedings of the First International Joint Conference on Artificial Intelligence – IJCAI-77. MA, pp. 888-893. 1977.

TATE, A., DRABBLE, B. and DALTON, J. *O-Plan: a Knowledge-Based Planner and its Application to Logistics*. AIAI, University of Edinburgh. 1996.

TATE, A., POLYAK, S. T. and Jarvis, P. *TF Method: An Initial Framework for Modeling and Analyzing Planning Domains*. Technical Report, University of Edinburgh. 1998.

TONIDANDEL, F., VAQUERO, T.S. and SILVA, J.R. *Reading PDDL, Writing an Object-Oriented Model*. In Lecture Notes in Computer Science. International Joint Conference on AI IBERAMIA/SBIA. Springer-Verlag. 2006.

VALK, R. *Object Petri Nets - Using the Nets-within-Nets Paradigm*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. vol. 3098/2004, pp. 819-848. 2004.

VAQUERO, T. S., TONIDANDEL, F. and SILVA, J.R. *The itSIMPLE tool for Modeling Planning Domains*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

VAQUERO, T. S., TONIDANDEL, F. BARROS, L.N. and SILVA, J.R. *On the Use of UML.P for Modeling a Real Application as a Planning Problem*. Short paper in Proceedings of ICAPS 2006 International Conference on Automated Planning and Scheduling, Cumbria, UK. June 2006.

YANG, Q. and TENENBERG, J. *ABTWEAK: Abstracting a Nonlinear, Least Commitment Planner*. In Proceedings of International Conference on Artificial Intelligence – AAAI-90. Boston, pp. 204-209. 1990.

WAH, B. W. and CHEN Y. *Subgoal Partitioning and Global Search for Solving Temporal Planning Problems in Mixed Space*. Int'l J. of Artificial Intelligence Tools, World Scientific Publishing Co. Pte. Ltd., vol. 13, no. 4, pp. 767-790. 2004.

WATANABE, H., TOKUOKA, H., WU, W. and SAEKI, M. *A Technique for Analysing and Testing Object-oriented Software Using Coloured Petri Nets*, IPSJ SIGNotes Software Engineering No.117. 1997.

WEBER, M. and KINDLER, E. *The Petri Net Markup Language*. Lecture In Notes for Computer Science series "Advances in Petri Nets". 2002. Acessível em http://www.informatik.hu-berlin.de/top/pnml/download/about/PNML_LNCS.pdf.

WILKINS, D.E., *Representation in a domain-independent planner*. In: Proceedings IJCAI-83 Karlsruhe, F.R.G. 1983.

WILKINS, D. *Domain-independent planning: representation and plan generation.* Artificial Intelligence, 22:269-301. 1984.

WILKINS, D. *Using the SIPE-2 Planning System: A Manual for SIPE-2, Version5.0.* SRI International, Artificial Intelligence Center. 1999.

WU, K., YANG, Q. and JIANG, Y. *ARMS: Action-relation modelling system for learning action models*. ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling, Monterey, California, USA. 2005.

Anexos

Anexo A – Modelos de Domínios em PDDL

Domínio Mundo de Blocos Estendido

Arquivo do Domínio:

```
(define (domain MundoDeBlocosEstendido)
    (:requirements :typing :negative-preconditions)
    (:types
      Arm - object
      Block - object
      Table - object
    )
    (:predicates
      (on ?blo - Block ?blo1 - Block)
      (holding ?arm - Arm ?blo - Block)
      (ontable ?blo - Block ?tab - Table)
      (at ?arm - Arm ?tab - Table)
      (hasAccessTo ?arm - Arm ?tab - Table)
      (handempty ?arm - Arm)
      (clear ?blo - Block)
      (attable ?blo - Block ?tab - Table)
      (available ?tab - Table)
    )
   (:action pickUp
     :parameters (?hand - Arm ?x - Block ?table - Table)
     :precondition
       (and
         (clear ?x)
         (ontable ?x ?table)
         (attable ?x ?table)
         (at ?hand ?table)
         (handempty ?hand)
       )
     :effect
       (and
         (not (clear ?x))
         (not (ontable ?x ?table))
         (not (attable ?x ?table))
         (holding ?hand ?x)
         (not (handempty ?hand))
       )
    )

    (:action putDown
     :parameters (?hand - Arm ?x - Block ?table - Table)
     :precondition
       (and
         (not (clear ?x))
         (at ?hand ?table)
         (holding ?hand ?x)
         (not (handempty ?hand))
```

```
    )
 :effect
   (and
     (clear ?x)
     (ontable ?x ?table)
     (attable ?x ?table)
     (not (holding ?hand ?x))
     (handempty ?hand)
   )
)

(:action stack
 :parameters (?hand - Arm ?x - Block ?y - Block ?table - Table)
 :precondition
   (and
     (clear ?y)
     (attable ?y ?table)
     (not (clear ?x))
     (at ?hand ?table)
     (holding ?hand ?x)
     (not (handempty ?hand))
   )
 :effect
   (and
     (clear ?x)
     (not (clear ?y))
     (on ?x ?y)
     (attable ?x ?table)
     (not (holding ?hand ?x))
     (handempty ?hand)
   )
)

(:action unstack
 :parameters (?hand - Arm ?x - Block ?y - Block ?table - Table)
 :precondition
   (and
     (clear ?x)
     (attable ?y ?table)
     (on ?x ?y)
     (attable ?x ?table)
     (at ?hand ?table)
     (handempty ?hand)
   )
 :effect
   (and
     (clear ?y)
     (not (clear ?x))
     (not (on ?x ?y))
     (not (attable ?x ?table))
     (holding ?hand ?x)
     (not (handempty ?hand))
   )
)

(:action changeTable
 :parameters (?hand - Arm ?tablefrom - Table ?tableto - Table)
 :precondition
   (and
      (at ?hand ?tablefrom)
```

```
          (available ?tableto)
          (hasAccessTo ?hand ?tableto)
        )
     :effect
       (and
         (at ?hand ?tableto)
         (not (at ?hand ?tablefrom))
         (available ?tablefrom)
         (not (available ?tableto))
       )
    )

)
```

Arquivo do Problema:

```
(define (problem Fifteen_Blocks_Five_Tables_and_Two_Hands)
    (:domain MundoDeBlocosEstendido)
    (:objects
      b1 - Block
      b2 - Block
      b3 - Block
      b4 - Block
      b5 - Block
      b6 - Block
      b7 - Block
      b8 - Block
      b9 - Block
      b10 - Block
      b11 - Block
      b12 - Block
      b13 - Block
      b14 - Block
      b15 - Block
      table1 - Table
      table4 - Table
      table5 - Table
      table2 - Table
      table3 - Table
      hand1 - Arm
      hand2 - Arm
    )
    (:init
      (hasAccessTo hand1 table1)
      (hasAccessTo hand1 table2)
      (hasAccessTo hand1 table3)
      (hasAccessTo hand2 table3)
      (hasAccessTo hand2 table4)
      (hasAccessTo hand2 table5)
      (on b1 b2)
      (on b2 b3)
      (ontable b3 table1)
      (on b4 b5)
      (ontable b5 table2)
      (on b6 b7)
      (on b7 b8)
      (ontable b8 table3)
```

```
    (on b9 b10)
    (on b10 b11)
    (ontable b11 table4)
    (on b12 b13)
    (at hand1 table1)
    (at hand2 table5)
    (on b13 b14)
    (on b14 b15)
    (ontable b15 table5)
    (attable b1 table1)
    (clear b1)
    (attable b2 table1)
    (attable b3 table1)
    (attable b4 table2)
    (clear b4)
    (attable b5 table2)
    (attable b6 table3)
    (clear b6)
    (attable b7 table3)
    (attable b8 table3)
    (attable b9 table4)
    (clear b9)
    (attable b10 table4)
    (attable b11 table4)
    (attable b12 table5)
    (clear b12)
    (attable b13 table5)
    (available table4)
    (available table2)
    (available table3)
    (handempty hand1)
    (handempty hand2)
    (attable b14 table5)
    (attable b15 table5)
  )
  (:goal
    (and
      (on b1 b2)
      (on b4 b5)
      (on b7 b8)
      (on b9 b10)
      (at hand1 table1)
      (at hand2 table5)
      (on b2 b13)
      (ontable b13 table1)
      (on b12 b3)
      (ontable b3 table5)
      (on b8 b11)
      (ontable b11 table3)
      (on b5 b6)
      (ontable b6 table2)
      (ontable b10 table4)
      (on b15 b9)
      (on b14 b12)
      (attable b1 table1)
      (clear b1)
      (attable b2 table1)
      (attable b3 table5)
      (attable b4 table2)
      (clear b4)
```

```
        (attable b5 table2)
        (attable b6 table2)
        (attable b7 table3)
        (clear b7)
        (attable b8 table3)
        (available table2)
        (available table3)
        (attable b9 table4)
        (attable b10 table4)
        (attable b11 table3)
        (attable b12 table5)
        (attable b13 table1)
        (available table4)
        (handempty hand1)
        (handempty hand2)
        (attable b14 table5)
        (clear b14)
        (attable b15 table4)
        (clear b15)
      )
    )
)
```

Domínio Logística Estendido

Arquivo do Domínio:

```
(define (domain Extended_Logistic_Domain)
    (:requirements :typing :fluents :negative-preconditions :equality)
    (:types
      Vehicle - object
      Truck - Vehicle
      Airplane - Vehicle
      Place - object
      Location - Place
      Airport - Place
      Package - object
      City - object
      FuelStation - object
    )
    (:predicates
      (isIn ?pac - Package ?veh - Vehicle)
      (isAt ?pac - Package ?pla - Place)
      (inCity ?pla - Place ?cit - City)
      (at ?tru - Truck ?pla - Place)
      (parkedAt ?air - Airplane ?air1 - Airport)
      (has ?pla - Place ?fue - FuelStation)
    )
    (:functions
      (capacity ?veh - Vehicle)
      (currentLoad ?veh - Vehicle)
      (fuelCapacity ?veh - Vehicle)
      (fuelLevel ?veh - Vehicle)
      (fuelBurn ?veh - Vehicle)
      (weight ?pac - Package)
```

```
      (fuelPrice ?fue - FuelStation)
      (distance ?p1 - Place ?p2 - Place)
      (totalFuelUsed)
      (totalFuelCost)
    )

    (:action drive
     :parameters (?truck - Truck ?from - Place ?to - Place ?city - City)
     :precondition
       (and
         (at ?truck ?from)
         (inCity ?from ?city)
         (inCity ?to ?city)
         (not (= ?from ?to))
         (>= (fuelLevel ?truck) (* (distance ?from ?to) (/ 1 (fuelBurn
?truck))))
       )
     :effect
       (and
         (not (at ?truck ?from))
         (at ?truck ?to)
         (decrease (fuelLevel ?truck) (* (distance ?from ?to) (/ 1
(fuelBurn ?truck))))
         (increase (totalFuelUsed) (* (distance ?from ?to) (/ 1 (fuelBurn
?truck))))
       )
    )

    (:action loadTruck
     :parameters (?truck - Truck ?pkg - Package ?place - Place)
     :precondition
       (and
         (at ?truck ?place)
         (>= (capacity ?truck) (+ (currentLoad ?truck) (weight ?pkg)))
         (isAt ?pkg ?place)
       )
     :effect
       (and
         (increase (currentLoad ?truck) (weight ?pkg))
         (not (isAt ?pkg ?place))
         (isIn ?pkg ?truck)
       )
    )

    (:action unloadTruck
     :parameters (?truck - Truck ?pkg - Package ?place - Place)
     :precondition
       (and
         (at ?truck ?place)
         (isIn ?pkg ?truck)
       )
     :effect
       (and
         (decrease (currentLoad ?truck) (weight ?pkg))
         (isAt ?pkg ?place)
         (not (isIn ?pkg ?truck))
       )
    )

    (:action refuelTruck
```

```
     :parameters (?truck - Truck ?place - Place ?fst - FuelStation)
     :precondition
       (and
         (at ?truck ?place)
         (has ?place ?fst)
         (> (fuelCapacity ?truck) (fuelLevel ?truck))
       )
     :effect
       (and
         (increase (totalFuelCost) (* (- (fuelCapacity ?truck) (fuelLevel
?truck)) (fuelPrice ?fst)))
         (assign (fuelLevel ?truck) (fuelCapacity ?truck))
       )
    )

    (:action fly
     :parameters (?plane - Airplane ?from - Airport ?to - Airport)
     :precondition
       (and
         (parkedAt ?plane ?from)
         (not (= ?from ?to))
         (>= (fuelLevel ?plane) (* (distance ?from ?to) (/ 1 (fuelBurn
?plane))))
       )
     :effect
       (and
         (not (parkedAt ?plane ?from))
         (parkedAt ?plane ?to)
         (decrease (fuelLevel ?plane) (* (distance ?from ?to) (/ 1
(fuelBurn ?plane))))
         (increase (totalFuelUsed) (* (distance ?from ?to) (/ 1 (fuelBurn
?plane))))
       )
    )

    (:action loadAirplane
     :parameters (?plane - Airplane ?pkg - Package ?place - Airport)
     :precondition
       (and
         (parkedAt ?plane ?place)
         (>= (capacity ?plane) (+ (currentLoad ?plane) (weight ?pkg)))
         (isAt ?pkg ?place)
       )
     :effect
       (and
         (increase (currentLoad ?plane) (weight ?pkg))
         (not (isAt ?pkg ?place))
         (isIn ?pkg ?plane)
       )
    )

    (:action unloadAirplane
     :parameters (?plane - Airplane ?pkg - Package ?place - Airport)
     :precondition
       (and
         (parkedAt ?plane ?place)
         (isIn ?pkg ?plane)
       )
     :effect
       (and
```

```
          (decrease (currentLoad ?plane) (weight ?pkg))
          (isAt ?pkg ?place)
          (not (isIn ?pkg ?plane))
        )
    )

    (:action refuelAirplane
     :parameters (?plane - Airplane ?place - Airport ?fst - FuelStation)
     :precondition
       (and
         (parkedAt ?plane ?place)
         (has ?place ?fst)
         (> (fuelCapacity ?plane) (fuelLevel ?plane))
       )
     :effect
       (and
         (increase (totalFuelCost) (* (- (fuelCapacity ?plane) (fuelLevel
?plane)) (fuelPrice ?fst)))
         (assign (fuelLevel ?plane) (fuelCapacity ?plane))
       )
    )

)
```

Arquivo do Problema:

```
(define (problem ExtendedLogisticTenPackages)
    (:domain Extended_Logistic_Domain)
    (:objects
      pkg1 - Package
      pkg2 - Package
      pkg3 - Package
      pkg4 - Package
      pkg5 - Package
      pkg6 - Package
      pkg7 - Package
      pkg8 - Package
      pkg9 - Package
      pkg10 - Package
      belohorizonte - City
      saopaulo - City
      riodejaneiro – City
      airportBH - Airport
      airportSP - Airport
      airportRJ – Airport
      locBH1 - Location
      locSP1 - Location
      locSP2 – Location
      locRJ1 - Location
      locRJ2 - Location
      locBH2 – Location
      truck1 - Truck
      truck2 - Truck
      truck3 - Truck
      plane1 - Airplane
      plane2 - Airplane
```

```
    fst1 - FuelStation
    fst2 - FuelStation
    fst3 - FuelStation
  )
  (:init
    (inCity locBH1 belohorizonte)
    (inCity airportBH belohorizonte)
    (inCity locSP1 saopaulo)
    (inCity airportSP saopaulo)
    (inCity locSP2 saopaulo)
    (inCity airportRJ riodejaneiro)
    (inCity locRJ1 riodejaneiro)
    (inCity locRJ2 riodejaneiro)
    (inCity locBH2 belohorizonte)
    (= (totalFuelCost) 0)
    (= (totalFuelUsed) 0)
    (parkedAt plane1 airportSP)
    (parkedAt plane2 airportBH)
    (at truck1 locSP2)
    (at truck2 locBH2)
    (at truck3 locRJ1)
    (isAt pkg1 locSP2)
    (isAt pkg2 locSP1)
    (isAt pkg3 locSP1)
    (has locSP1 fst3)
    (has locBH2 fst1)
    (isAt pkg4 locBH1)
    (isAt pkg5 locBH1)
    (isAt pkg6 locBH1)
    (has airportRJ fst2)
    (isAt pkg9 locRJ2)
    (isAt pkg10 locRJ2)
    (isAt pkg7 locRJ1)
    (isAt pkg8 locRJ1)
    (= (weight pkg4) 150)
    (= (weight pkg5) 150)
    (= (weight pkg6) 150)
    (= (weight pkg7) 200)
    (= (fuelBurn truck1) 5.0)
    (= (fuelLevel truck1) 10.0)
    (= (fuelCapacity truck1) 90)
    (= (currentLoad truck1) 0)
    (= (capacity truck1) 500.0)
    (= (distance airportSP airportRJ) 492.0)
    (= (distance airportRJ airportSP) 492.0)
    (= (distance airportSP airportBH) 586.0)
    (= (distance airportBH airportSP) 586.0)
    (= (distance airportRJ airportBH) 434.0)
    (= (distance airportBH airportRJ) 434.0)
    (= (distance locSP1 locSP2) 25.0)
    (= (distance locSP2 locSP1) 25.0)
    (= (distance locSP1 airportSP) 30.0)
    (= (distance airportSP locSP1) 30.0)
    (= (distance locSP2 airportSP) 40.0)
    (= (distance airportSP locSP2) 40.0)
    (= (distance locRJ1 locRJ2) 45.0)
    (= (distance locRJ2 locRJ1) 45.0)
    (= (distance locRJ1 airportRJ) 35.0)
    (= (distance airportRJ locRJ1) 35.0)
    (= (distance locRJ2 airportRJ) 20.0)
```

```
      (= (distance airportRJ locRJ2) 20.0)
      (= (distance locBH1 locBH2) 55.0)
      (= (distance locBH2 locBH1) 55.0)
      (= (distance locBH1 airportBH) 40.0)
      (= (distance airportBH locBH1) 40.0)
      (= (distance locBH2 airportBH) 48.0)
      (= (distance airportBH locBH2) 48.0)
      (= (weight pkg1) 200)
      (= (weight pkg2) 200)
      (= (weight pkg3) 300)
      (= (fuelBurn truck2) 4.0)
      (= (fuelLevel truck2) 25.0)
      (= (fuelCapacity truck2) 85)
      (= (currentLoad truck2) 0)
      (= (capacity truck2) 400.0)
      (= (fuelBurn truck3) 5.0)
      (= (fuelLevel truck3) 70.0)
      (= (fuelCapacity truck3) 80.0)
      (= (currentLoad truck3) 0)
      (= (capacity truck3) 500.0)
      (= (fuelBurn plane1) 12.0)
      (= (fuelLevel plane1) 150.0)
      (= (fuelCapacity plane1) 100.0)
      (= (currentLoad plane1) 0)
      (= (capacity plane1) 800.0)
      (= (fuelBurn plane2) 14.0)
      (= (fuelLevel plane2) 60.0)
      (= (fuelCapacity plane2) 110.0)
      (= (currentLoad plane2) 0)
      (= (capacity plane2) 600.0)
      (= (weight pkg8) 100)
      (= (weight pkg9) 100)
      (= (weight pkg10) 300)
      (= (fuelPrice fst1) 1.8)
      (= (fuelPrice fst2) 1.7)
      (= (fuelPrice fst3) 2.0)
    )
    (:goal
      (and
        (isAt pkg1 locBH2)
        (isAt pkg3 locBH2)
        (isAt pkg8 airportBH)
        (isAt pkg6 locSP2)
        (isAt pkg7 locSP2)
        (isAt pkg9 locSP2)
        (isAt pkg10 airportSP)
        (isAt pkg5 airportRJ)
        (isAt pkg2 locRJ2)
        (isAt pkg4 locRJ2)
      )
    )
    (:metric minimize (+ (totalFuelUsed) (totalFuelCost)))
)
```

Montagem Seqüencial de Carros em Linhas de Montagem

Arquivo do Domínio:

```
(define (domain CarSequencing)
(:requirements :adl :conditional-effects  :fluents)
(:types SpecialFeature
        AssemblyLine
        Color
        Vehicle
        SprayGun
        Assembler
        Transporter - object)
(:predicates      (worksOn  ?abl - Assembler ?ass - AssemblyLine)
            (currentColor  ?Spr - SprayGun ?Col - Color)
            (belongsTo  ?Spr - SprayGun ?Ass - AssemblyLine)
            (cleaned  ?Spr - SprayGun)
            (paintColor  ?Veh - Vehicle ?Col - Color)
            (date  ?Veh - Vehicle)
            (painted  ?Veh - Vehicle)
            (assembled  ?Veh - Vehicle)
            (at  ?Veh - Vehicle  ?Ass - AssemblyLine)
            (has  ?Veh - Vehicle  ?Spe - SpecialFeature)
            (availableAt  ?Spe - SpecialFeature  ?Ass - AssemblyLine)
)
(:functions (paintBatchLimit  ?Spr - SprayGun)
            (batchCounter  ?Spr - SprayGun)
            (lastPaintedSeqRank ?Spg - SprayGun)
            (lastAssembledSeqRank ?Abl)
            (seqRank  ?Veh - Vehicle)
            (currentSeqRank  ?Ass - AssemblyLine)
            (featureCounter ?Ass - AssemblyLine ?Spe - SpecialFeature)
            (n ?Spe - SpecialFeature)
            (p ?Spe - SpecialFeature)
            (afterAssembledPosition ?Veh - Vehicle)
            (numberPaintColorChanges)
            (numberViolations)
)

(:action putVehicle
  :parameters (?tra - Transporter ?veh - Vehicle ?ayl - AssemblyLine)
  :precondition
      (and
            (forall (?spe - SpecialFeature) (or (not(has ?veh ?spe)) (and
(has ?veh ?spe) (availableAt ?spe ?ayl))))
            (= (seqRank ?veh) 0)
            (not(painted ?veh))
            (not(assembled ?veh))
      )
  :effect
      (and
            (assign (seqRank ?veh) (currentSeqRank ?ayl))
            (increase (currentSeqRank ?ayl) 1)
            (at ?veh ?ayl)
      )
)

(:action assemble
  :parameters ( ?abl - Assembler ?veh - Vehicle ?ayl - AssemblyLine)
```

```
:precondition
    (and
          (worksOn ?abl ?ayl)
          (= (seqRank ?veh) (+ (lastAssembledSeqRank ?abl) 1))
          (at ?veh ?ayl)
          (painted ?veh)
          (not(assembled ?veh))
    )
:effect
    (and

          ;; counting number of violations - Optimization aspect
          (forall (?spe - SpecialFeature)
                (when
                      (and
                            (has ?veh ?spe)
                            (availableAt ?spe ?ayl)
                            (exists (?v - Vehicle)
                               (and (has ?v ?spe)
                                 (= (afterAssembledPosition ?v) (p ?spe))
                                 (at ?v ?ayl)
                                )
                            )
                            (> (featureCounter ?ayl ?spe) (n ?spe))
                      )
                      (increase (numberViolations) 1)
                )

          )

          (forall (?spe - SpecialFeature)
                (when
                      (and
                            (has ?veh ?spe)
                            (availableAt ?spe ?ayl)
                            (exists (?v - Vehicle)
                               (and (not(has ?v ?spe))
                                 (= (afterAssembledPosition ?v) (p ?spe))
                                 (at ?v ?ayl)
                                )
                            )
                            (> (+ (featureCounter ?ayl ?spe) 1) (n ?spe))
                      )
                      (increase (numberViolations) 1)
                )

          )

          (forall (?spe - SpecialFeature)
                (when
                      (and
                            (has ?veh ?spe)
                            (availableAt ?spe ?ayl)
                            (exists (?v - Vehicle)
                               (and (not(has ?v ?spe))
                                 (= (afterAssembledPosition ?v) (p ?spe))
                                 (at ?v ?ayl))
                            )
                      )
                      (increase (featureCounter ?ayl ?spe) 1)
```

```
                )
            )

            (forall (?spe - SpecialFeature)
                (when
                    (and
                        (not(has ?veh ?spe))
                        (availableAt ?spe ?ayl)
                        (exists (?v - Vehicle)
                           (and (has ?v ?spe)
                             (= (afterAssembledPosition ?v) (p ?spe))
                             (at ?v ?ayl))
                        )
                        (> (featureCounter ?ayl ?spe) 0)
                    )
                    (decrease (featureCounter ?ayl ?spe) 1)
                )

            )


            (forall (?ve - Vehicle)
                (when (and (at ?ve ?ayl) (assembled ?ve))
                    (increase (afterAssembledPosition ?ve) 1)
                )
            )

            (assign (afterAssembledPosition ?veh) 1)
            ;; End of Ordering Optimization


            (assembled ?veh)
            (assign (lastAssembledSeqRank ?abl) (seqRank ?veh))

        )
)

(:action wash
  :parameters ( ?spg - SprayGun)
  :precondition
      (and
            (not(cleaned ?spg))
            (> (batchCounter ?spg) 0)
      )
  :effect
      (and
            (cleaned ?spg)
            (assign (batchCounter ?spg) 0)
      )
)

(:action changeColor
  :parameters ( ?spg - SprayGun ?from - Color ?to - Color)
  :precondition
      (and
            (not(= ?from ?to))
            (currentColor ?spg ?from)
            (cleaned ?spg)
            (= (batchCounter ?spg) 0)
      )
```

```
  :effect
      (and
            (currentColor ?spg ?to)
            (not(currentColor ?spg ?from))

            ;; Painting Optimization
            (increase (numberPaintColorChanges) 1)
      )
)

(:action paint
  :parameters ( ?spg - SprayGun ?veh - Vehicle ?col - Color
                  ?ayl - AssemblyLine ?abl - Assembler)
  :precondition
      (and
            ;;for better looking of the resulting plan
            (forall (?v - Vehicle) (> (seqRank ?v) 0))

            (belongsTo ?spg ?ayl)
            (at ?veh ?ayl)
            (currentColor ?spg ?col)
            (paintColor ?veh ?col)
            (< (batchCounter ?spg) (paintBatchLimit ?spg))
            (= (seqRank ?veh) (+ (lastPaintedSeqRank ?spg) 1))
            (not(painted ?veh))
            (not(assembled ?veh))
            (worksOn ?abl ?ayl)
            (= (lastPaintedSeqRank ?spg) (lastAssembledSeqRank ?abl))

      )
  :effect
      (and
            (increase (batchCounter ?spg) 1)
            (assign (lastPaintedSeqRank ?spg) (seqRank ?veh))
            (not(cleaned ?spg))
            (painted ?veh)
      )
)
)
```

Arquivo do Problema:

```
(define (problem EightCarsThreeSpf)
(:domain CarSequencing)
(:objects
     vehicle1 vehicle2 vehicle3 vehicle4 vehicle5 vehicle6 vehicle7
vehicle8
     vehicleLD1 vehicleLD2 vehicleLD3 vehicleLD4 vehicleLD5 - Vehicle
     s1 - SprayGun
     a1 - Assembler
     red blue - Color
     t1 - Transporter
     ayl1 - AssemblyLine
     spf1 spf2 spf3 - specialFeature
)

(:init
```

```
;; Vehicles of day D

(= (seqRank vehicle1) 0)
  (paintColor vehicle1 red)
(= (afterAssembledPosition vehicle1) 0)
(has vehicle1 spf1)
(has vehicle1 spf2)

(= (seqRank vehicle2) 0)
  (paintColor vehicle2 red)
(= (afterAssembledPosition vehicle2) 0)
(has vehicle2 spf1)
(has vehicle2 spf2)
(has vehicle2 spf3)

(= (seqRank vehicle3) 0)
  (paintColor vehicle3 red)
(= (afterAssembledPosition vehicle3) 0)
(has vehicle3 spf1)

(= (seqRank vehicle4) 0)
  (paintColor vehicle4 red)
(= (afterAssembledPosition vehicle4) 0)


(= (seqRank vehicle5) 0)
  (paintColor vehicle5 red)
(= (afterAssembledPosition vehicle5) 0)
(has vehicle5 spf2)

(= (seqRank vehicle6) 0)
  (paintColor vehicle6 red)
(= (afterAssembledPosition vehicle6) 0)
(has vehicle6 spf3)

(= (seqRank vehicle7) 0)
  (paintColor vehicle7 blue)
(= (afterAssembledPosition vehicle7) 0)


(= (seqRank vehicle8) 0)
  (paintColor vehicle8 red)
(= (afterAssembledPosition vehicle8) 0)
(has vehicle8 spf2)

;; Spray Guns

(belongsTo s1 ayl1)
(currentColor s1 red)
(cleaned s1)
(= (paintBatchLimit s1) 10)
(= (batchCounter s1) 0)
(= (lastPaintedSeqRank s1) 0)


;; Assemblers

(worksOn a1 ayl1)
(= (lastAssembledSeqRank a1) 0)
```

```
;; Assembly Lines

(= (currentSeqRank ayl1) 1)


;; Special Features

(availableAt spf1 ayl1)
(= (featureCounter ayl1 spf1) 1)
(= (n spf1) 1)
(= (p spf1) 3)

(availableAt spf2 ayl1)
(= (featureCounter ayl1 spf2) 1)
(= (n spf2) 1)
(= (p spf2) 2)

(availableAt spf3 ayl1)
(= (featureCounter ayl1 spf3) 1)
(= (n spf3) 1)
(= (p spf3) 5)

;; Optimization functions

(= (numberPaintColorChanges) 0)
(= (numberViolations) 0)


;; Vehicles of day D-1 -> Last Day

(= (seqRank vehicleLD1) 100)
  (paintColor vehicleLD1 red)
(= (afterAssembledPosition vehicleLD1) 1)
(painted vehicleLD1)
(assembled vehicleLD1)
(at vehicleLD1 ayl1)
(has vehicleLD1 spf2)

(= (seqRank vehicleLD2) 99)
  (paintColor vehicleLD2 red)
(= (afterAssembledPosition vehicleLD2) 2)
(painted vehicleLD2)
(assembled vehicleLD2)
(at vehicleLD2 ayl1)
(has vehicleLD2 spf1)

(= (seqRank vehicleLD3) 98)
  (paintColor vehicleLD3 red)
(= (afterAssembledPosition vehicleLD3) 3)
(painted vehicleLD3)
(assembled vehicleLD3)
(at vehicleLD3 ayl1)
(has vehicleLD3 spf3)

(= (seqRank vehicleLD4) 97)
  (paintColor vehicleLD4 red)
(= (afterAssembledPosition vehicleLD4) 4)
(painted vehicleLD4)
(assembled vehicleLD4)
```

```
      (at vehicleLD4 ayl1)

      (= (seqRank vehicleLD5) 96)
        (paintColor vehicleLD5 red)
      (= (afterAssembledPosition vehicleLD5) 5)
      (painted vehicleLD5)
      (assembled vehicleLD5)
      (at vehicleLD5 ayl1)
 )

(:goal
      (and
            (assembled vehicle1)
            (painted vehicle1)
            (assembled vehicle2)
            (painted vehicle2)
            (assembled vehicle3)
            (painted vehicle3)
            (assembled vehicle4)
            (painted vehicle4)
            (assembled vehicle5)
            (painted vehicle5)
            (assembled vehicle6)
            (painted vehicle6)
            (assembled vehicle7)
            (painted vehicle7)
            (assembled vehicle8)
            (painted vehicle8)
      )
)

(:metric minimize (+ (numberViolations) (numberPaintColorChanges)))
)
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo