Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Conversão de Modelo Orientado a Objetivos para Planejador Automatizado Baseado em Redes de Tarefas Hierárquicas

Allisson Matheus de Rezende Barros

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Célia Ghedini Ralha

Brasília
2017

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Conversão de Modelo Orientado a Objetivos para Planejador Automatizado Baseado em Redes de Tarefas Hierárquicas

Allisson Matheus de Rezende Barros

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Célia Ghedini Ralha (Orientadora)
CIC/UnB

Prof.a Dr.a Genaina Nunes Rodrigues
CIC/Unb

Dr.a Vanessa Tavares Nunes
PPGinf/CIC/Unb

Prof. Dr. Rodrigo Bonifacio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 14 de dezembro de 2017

Dedicatória

Dedico esse trabalho à minha família, em especial a meu pai Adão, minha mãe Neile e minha irmã Ana Paula, pelo apoio incondicional em todas as etapas da minha vida e por me proporcionarem oportunidades que me trouxeram até aqui. Foram eles que me deram princípios que eu guardo e guardarei para a vida toda, além de sabedoria para atingir meus objetivos e um amor incondicional que não há como expressar.

Agradecimentos

A minhas orientadoras, Prof.a Dr.a Célia Ghedini e Dr.a Vanessa Tavares, por acreditarem no meu projeto e pelo cuidado que tiveram em me ajudar nessa trajetória. A meus amigos que me acompanharam durante minha graduação, os quais me auxiliaram em vários momentos e de diversas formas, especialmente Caio Batista, Mateus Denucci, Matheus Moretti e Luiz Fernando Vieira. A meus familiares que sempre me apoiaram e motivaram, em especial ao meu tio Dr. Evander Rezende, pelo exemplo de superação e dedicação aos estudos e a minha tia Eva Barros, por sempre acreditar em mim.

Resumo

O desenvolvimento de software é imprescindível à sociedade e permeado de vários contextos. Entre eles, destaca-se o contexto da Engenharia de Requisitos (ER), como a etapa inicial no projeto de software. Para tanto, abordagens, conceitos e ferramentas são empregados, de forma a garantir a qualidade do software a ser desenvolvido. Em outro contexto, insere-se a Inteligência Artificial, como um caminho alternativo à definição e solução de problemas. Dessa forma, é possível conjugar conhecimentos das duas áreas, a fim de criar modelos e ferramentas capazes de agregar conhecimento e qualidade ao processo de especificação de software. Este trabalho apresenta uma proposta de solução para a conversão entres os elementos desses domínios. Essa conversão visa oferecer uma nova forma de avaliação e análise dos modelos de requisitos através do Planejamento Automatizado. Utilizando como ferramenta de modelagem o framework Goal Oriented Dependability Analysis (GODA), que implementa a análise de requisitos orientados à objetivos e como planejador o Pyhop, que implementa a abordagem Hierarchical Task Networks (HTN) como formalização de domínios e problemas de planejamento, esse trabalho é ilustrado por um protótipo implementado utilizando as linguagens de programação Java e Python a fim de integrar as duas ferramentas citadas, representando na prática o modelo de conversão proposto. Para atestar a adequação do modelo e a implementação do protótipo, foram concebidos quatro experimentos, sendo um deles aplicado a um cenário real.

Palavras-chave: Engenharia de Requisitos, Planejamento Automatizado, Modelo de Objetivos Contextual e de Execução, Redes de Tarefas Hierárquicas, TROPOS, Taom4e, GODA, Pyhop

Abstract

The development of software is indispensable to society and filled with many different aspects. Between those, the context of Software Engineering stands out, as the initial phase on designing software. Therefore, aiming to achieve quality, different approaches, concepts and tools are adopted. In another context, Artificial Intelligence is employed as another form of achieving solutions to defined problems. Thus, it is possible to bind knowledge form both areas of study, with the intent to create models and tools capable of increasing quality on the process of software specification. This research aims at proposing a solution to translate elements between this two domains. This form of conversion is displayed as a new approach on evaluation and analysis of software requirements through Automated Planning. By adopting, as a modeling tool, the Goal Oriented Dependability Analysis (GODA) framework, which implements goal oriented requirements analysis and, as a planner, Pyhop, which implements Hierarchical Task Networks (HTN) as a formalization of planning problems and domains, this work is illustrated with a prototype, developed in Java and Python programming languages, as a composition of the tools mentioned and as a practical representation of the concepts proposed. Lastly, as a way of attesting the adequacy of the model and the correct implementation of the prototype, four different experiments were conceived, including a real life scenery.

Keywords: Requirements Engineering, Automated Planning, Contextual Runtime Goal Model, Hierarquical Task Networks, TROPOS, Taom4e, GODA, Pyhop

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Engenharia de *Software* (ES) é uma grande área da Ciência da Computação que se faz presente em diversos tipos de aplicação e domínios. Dentro dessa área se encontra a Engenharia de Requisitos (ER), cujo objeto de estudo é a etapa de criação que precede o desenvolvimento do *software*. Com o crescimento dos projetos e a evolução das tecnologias, esse processo de especificação de um produto de *software* se tornou cada vez mais complexo e variado, de forma que há várias maneiras de abordá-lo. Entre elas está a Engenharia de Requisitos Orientada a Objetivos, onde o foco do processo é satisfazer as necessidades e os propósitos de um indivíduo ou organização.

Por outro lado, a evolução dos dispositivos computacionais trouxe a necessidade de um estudo sobre inteligência que aplicasse o uso de raciocínio a máquinas e *software*, de forma que suas ações fossem guiadas por recursos inteligentes. A área de Inteligência Artificial (IA), apesar de nova em relação a outras ciências, possui muitas vertentes de estudo e entre elas está o Planejamento Automatizado. Planejar consiste em escolher e organizar ações antecipando seus resultados (ou seus efeitos) [6]. E então, o Planejamento Automatizado é uma área da Inteligência Artificial que estuda esse processo de deliberação de forma computacional.

Assim, seria interessante criar uma abordagem capaz de aproveitar características do Planejamento Automatizado que sirva como uma nova forma de avaliar os modelos orientados a objetivos .Dito isso, como desenvolver um modelo de conversão automática de requisitos de *software* orientados a objetivos para uma linguagem de Planejamento Automatizado?

Para responder a essa pergunta foi criado um modelo de conversão que se baseia na formalização aplicada em duas ferramentas, para modelagem orientada a objetivos, o uso do Goal Oriented Dependability Analysis (GODA) e, para Planejamento Automatizado, o Pyhop (versão desenvolvida em Python do Simple Hierarchical Ordered Planner (SHOP)). O GODA é um framework que utiliza o Tropos e aplica regras e contextos para análise

de dependabilidade em tempo de execução. Pyhop é um planejador que implementa a abordagem Hierarchical Task Networks (HTN) para planejar sobre domínios e problemas.

1.1 Problema

A Engenharia de Requisitos, apesar de ser uma grande área de estudo e aplicação, ainda concentra muito conhecimento em poucas ferramentas e especializações e, portanto, novas abordagens são interessantes à evolução do estudo nessa área. Além disso, o levantamento de requisitos é um processo racional que pode ser muito complexo dependendo do sistema a ser especificado e realizar esse procosse de forma correta e eficiente é um dos grandes desafios da áres. Nesse espaço, seria interessante agregar conhecimentos de outras áreas, como Planejamento Automatizado, de forma a automatizar os conhecimentos de ER e oferecer outras possibilidades de ferramentas e modelos para levantamento de requisitos de *software*.

1.2 Objetivos

O objetivo geral desse trabalho é propor uma solução para converter modelos orientados a objetivos para um modelo de Planejamento Automatizado baseado em redes de tarefas hierárquicas (HTN). Os objetivos específicos da pesquisa são:

1. Definir a equivalência entre elementos de modelagem orientada a Objetivos e elementos de Planejamento Automatizado baseado em redes de tarefas hierárquicas.
2. Desenvolver um protótipo que realiza essa conversão
3. Avaliar o protótipo utilizando um caso de uso aplicado a tráfego de informações sobre transporte público.

1.3 Metodologia

Esse trabalho foi desenvolvido em seis etapas: (i) estudo de conceitos, algoritmos e ferramentas de Planejamento Automatizado; (ii) estudo do modelo orientado a objetivos GODA e da ferramenta Taom4e; (iii) estudo para escolha de ferramentas e linguagens mais adequadas para implementar um protótipo de conversão do modelo GODA para linguagem de Planejamento Automatizado; (iv) desenvolvimento de um protótipo para conversão automática; (v) desenvolvimento de experimentos para ilustrar a utilização do

protótipo desenvolvido; (vi) análise dos resultados e conclusões com pesquisa exploratória da literatura, a fim de definir a abordagem utilizada para modelagem orientada a Objetivos, bem como a abordagem utilizada para Planejamento Automatizado.

1.4 Apresentação do Manuscrito

No Capítulo 2 são apresentados os conceitos fundamentais para o melhor entendimento do que foi explorado nesse trabalho, bem como alguns trabalhos semelhantes que foram analisados. No Capítulo 3, a definição completa da proposta de solução. No Capítulo 4, os experimentos realizados. No Capítulo 5, uma discussão sobre os resultados obtidos e conclusões sobre o que foi desenvolvido.

Capítulo 2

Fundamentação Teórica

Neste capítulo são introduzidos os conceitos fundamentais estudados durante o desenvolvimento do trabalho, de forma a contextualizar as áreas de estudo envolvidas. Na Seção 2.1, o conceito de Modelagem Orientada à Objetivos é introduzido juntamente com Análise de Dependabilidade e de *Runtime*. Alguns exemplos de aplicação desse são apresentados, bem como ferramentas e linguagens mais utilizadas. Na Seção 2.2 é apresentada a teoria fundamental de Planejamento, no contexto da Inteligência Artificial. Os conceitos teóricos, exemplos de problemas clássicos, algoritmos e linguagens/ferramentas de planejamento são explicados também nessa seção e, por fim, são apresentados outros trabalhos relacionados à este.

2.1 Modelagem de Processos Orientada a Objetivos

Modelos Orientados a Objetivos pertencem à área da Engenharia de *Software* responsável pelos requisitos de especificação de um sistema. Essa área é conhecida como Engenharia de Requisitos. Segundo [7], o interesse de ER é a elicitação, análise, especificação e validação dos requisitos de *software*, bem como a gerência desses requisitos durante todo o ciclo de vida do produto de *software*. E, por sua vez, os requisitos de um *software* expressam as necessidades e restrições colocadas em um produto de *software*, os quais contribuem para a solução de algum problema do mundo real.

Ademais, os requisitos ainda podem ser divididos em três grupos: requisitos funcionais, não-funcionais e de domínio [8]. Como expressado pelo termo, requisitos funcionais são aqueles que representam funcionalidades ou serviços a serem executados pelo sistema. Muitas vezes, de forma bem explícita e definida, com entrada e saída para requisitos a nível de sistema ou de forma mais genérica para requisitos de usuário.

Em contrapartida, os requisitos não-funcionais estão mais ligados à restrições que devem ser impostas sobre o sistema e não em relação as funcionalidades que devem ser

oferecidas. Um bom exemplo é o tempo de resposta, geralmente um requisito comum em vários tipos de sistemas e domínios. Além disso, esse tipo de requisito também pode se referir a regulamentações que o sistema deve seguir ou uso de dispositivos de E/S específicos.

Além desses, requisitos de domínio são aqueles que são específicos ao domínio de aplicação, por vezes fazendo parte das necessidades apenas dos usuários desse sistema. Esse tipo de requisito envolve os outros tipos apresentados, de forma que tanto um requisito funcional como um não-funcional podem ser considerados também como requisitos de domínio. Exemplificando, um sistema de ponto eletrônico pode possuir um requisito como "Os sensores biométricos devem ser do modelo X", que é ao mesmo tempo um requisito de domínio (uma vez que apenas o uso de sensores biométricos caracteriza um domínio de sistemas de ponto) e também não-funcional (já que restringe o uso de sensores do modelo X ao invés de usar um sensor biométrico qualquer).

2.1.1 Objetivos como Requisitos de *Software*

Outra abordagem às etapas de especificação de um sistema de *software* é a utilização de objetivos como requisitos. Diferentemente dos requisitos comuns em ES, os objetivos são formas mais abstratas de representar quais características do *software* são interessantes aos *stakeholders*, ou seja, como essas abstrações representam as necessidades e expectativas a serem supridas pelo sistema [9]. Além disso, a ideia do uso de objetivos na especificação de sistemas é justamente capturar a intencionalidade por trás dos requisitos de um sistema de *software* [10].

Ainda sobre o porquê do uso de objetivos na análise de requisitos, em [11], a autora observa que o assunto de ER é inerentemente amplo, interdisciplinar e aberto. Trata-se de tradução de observações informais do mundo real para linguagens de especificação matemática. Nesse ponto, o uso de objetivos como requisitos de *software* é interessante para aprimorar essa "tradução de observações"em constructos e formalismos matemáticos.

Todavia, os conceitos de objetivos discutidos pelos autores sempre são relacionados com sua utilidade e significado aplicados ao framework ou ferramenta criada por tais autores. Segundo [10], é difícil discernir uma noção uniforme de objetivo em ER, o que o autor acredita ser causado pelas mesmas razões apontadas por [11] no parágrafo anterior.

Portanto, é interessante avaliar as ferramentas e modelos criados com a aplicação de objetivos para compreender, através do conceitos definidos pelos seus autores, como os objetivos são utilizados na especificação de sistemas de *software* e o que de fato representam.

2.1.2 Modelagem de Objetivos

A aplicação dos objetivos na ER é feita através dos modelos. Como discutido, anteriormente, os modelos trazem uma ilustração do que são os objetivos e como podem ser usados. Os modelos, em geral, trazem cobertura para quatro fases do levantamento de requisitos: (i) requisitos iniciais, (ii) requisitos finais, (iii) *design* de arquitetura e (iv) *design* detalhado.

seguem várias abstrações diferentes e com focos diferentes, como por exemplo, a metodologia *Knowledge Acquisition in autOmated Specification* (KAOS), aonde os objetivos são o conceito central, ou então o modelo Tropos, que utiliza os objetivos de outra forma, dando mais importância ao agentes que querem atingir seus objetivos. Ou ainda, um modelo com uma preocupação com contexto, que será abordado na Seção 2.1.3.

Nesse trabalho, o modelo base utilizado foi estendido da modelagem Tropos, que será apresentada juntamente com i* e KAOS, para melhor entendimento da inserção desses modelos na ER orientada a objetivos e sua relevância para a área. Na sequência, uma breve explicação sobre o uso de modelos de objetivos centrados em contextos e do framework GODA, também utilizado no desenvolvimento deste trabalho.

i*

A modelagem i* foi criada com o intuito de incluir fatores sociais na fase de elaboração dos requisitos de um sistema de *software* [12]. Na Seção 2.1.2 foi levantado o aspecto abstrato do levantamento de requisitos no sentido de que não é trivial conceber formalização para observações do mundo real. Dessa forma, é comum que se queira diminuir a incerteza dessas observações, principalmente abstraindo os aspectos humanos. Porém, segundo [12], i* reconhece a primazia de atores considerando a forma como os objetivos de cada ator são alcançados. Assim, outros fatores são também considerados, como por exemplo as relações entre pessoas e os atores e quais dessas relações podem contribuir para os interesses dos atores. Essa metodologia é aplicada a etapa de requisitos iniciais da elaboração do *software*.

KAOS

A metodologia KAOS (*Knowledge Acquisition in autOmated Specification*) possui uma abordagem mais voltada para formalismos, especialmente durante as fases de *early* e *late requirements*, ou seja, na fase de levantamento de requisitos de *software*. Nessa metodologia, os objetivos são representados de forma hierárquica e possuem relações AND/OR para definir a hierarquia. Eles também são divididos em duas categorias, objetivos fun-

cionais, referentes a serviços, e objetivos não-funcionais que representam a qualidade dos serviços.

Outra característica é o uso de objetos, que podem ser entidades, relacionamentos ou eventos que evoluem e se modificam durante as transições de estados. Essas transições são feitas por meio de Operações, que possuem pré e pós condições (de forma semelhante a algumas linguagens de Planejamento Automatizado discutidas a Seção 2.2.2). Por fim, uma funcionalidade importante da metodologia é a capacidade de criar metas que se referem aos objetivos e com isso os modelos podem ser ajustados de forma a otimizar o levantamento de requisitos. Essa metodologia é aplicada a etapa de requisitos iniciais e também à etapa de requisitos finais da elaboração do *software*.

Tropos

Tropos é uma metodologia voltada a agentes e que também é um aperfeiçoamento do i*. O foco aqui é utilizar uma abordagem de desenvolvimento de *software* orientado a requisitos e isso é feito durante quatro etapas: *early* e *late* requirements, design da arquitetura e design detalhado. Além disso, a proposta também inclui uma interface com plataforma de programação orientadas a agentes e existe uma ferramenta (*Tool for Agent Oriented Modeling (Taom4e)*[1]) que implementa a modelagem utilizando Tropos.

A metodologia possui cinco conceitos básicos: atores, que são papéis, entidades ou agentes (humanos ou de *software*); objetivos, que são os interesses estratégicos de um ator (classificados em *soft* e *hard* goals); tarefas, que são ações que podem ser executadas para satisfazer um objetivo; recursos, que são entidades físicas ou informacionais; e dependências, que são relações entre atores que impõem restrições entre eles para que um objetivo seja satisfeito ou uma tarefa executada.

As relações entre objetos são realizadas de formas diferentes, dependendo do objeto. Entre objetivos ou entre tarefas, as relação são decomposições do tipo AND/OR. Também há a relação *meio-fim*, de forma que uma tarefa (meio) pode ser usada para alcançar um objetivo (fim). Existem ainda as relações de contribuição, onde tarefas ou objetivos contribuem para satisfazer um *soft-goal* (essas relações também possuem informações sobre contribuição positiva ou negativa e intensidade da contribuição).

Essa metodologia é aplicada às quatro etapas de requisitos de levantamento de requisitos, de forma que é possível modelar as duas primeiras, como no KAOS, mas também as etapas de *design* e agregar mais informações aos modelos.

[1] http://selab.fbk.eu/taom/

2.1.3 *Contextual and Runtime Goal Model*

O uso de contextos em ER pode ser uma abordagem rica. Como discutido por [9], contextos podem exercer grande influência sobre os interesses e os objetivos de um ator, seja na forma como esses objetivos são alcançados ou na qualidade deles. Segundo os autores, a definição de contexto varia conforme a sua aplicação a um domínio, sendo difícil criar uma definição única, porém, ele apresenta contextos como estado parcial do mundo que é relevante para os objetivos de um ator. Por outro lado, em [13], os autores introduzem o conceito de *runtime* na modelagem de objetivos, uma vez que, segundo eles, durante a execução, o comportamento do sistema se caracteriza de forma diferente de acordo com os eventos que ocorrem e com as instâncias dos objetivos utilizadas naquela execução. Assim, para eles é interessante que exista uma forma de monitorar a execução do sistema para diagnosticar possíveis incoerências entre os requisitos e o funcionamento real do sistema de *software*.

É interessante notar que os dois conceitos apresentados são pertinentes ao levantamento de requisitos e podem incrementar esse processo. Sobre esse ponto de vista, surgiu uma abordagem que ao mesmo tempo considera fatores de contexto, utilizando especificações de contexto e fatores de *runtime* como as instâncias de objetivos e tarefas, seguindo regras *Runtime Goal Model* estendidas de [13]. Essa abordagem é o GODA [5].

GODA

O framework GODA é proposto como ferramenta para agregar análises de contexto e *runtime*. Como o contexto tem influência sobre a satisfação de objetivos, o framework possibilita especificar a interação entre eles e estimar a dependabilidade das estratégias para os objetivos em diferentes contextos. Ao mesmo tempo, já com foco sobre *runtime*, o GODA provê resultados que indicam se um sistema está ou não conseguindo satisfazer os objetivos do ator. Na Tabela 2.1, estão listadas as regras RGM do GODA, onde um nó *n* é decomposto em *n1* e *n2*, de acordo com a semântica de cada regra.

O processo de modelagem do GODA funciona basicamente criando um modelo de objetivos usando metodologias convencionais e aplicando a esse modelo informações de contexto e *runtime*, tornando-o um modelo CRGM (*Contextual and Runtime Goal Model*). A partir daí, o modelo é convertido para DTMC (*Discrete-Time Markov Chains*), cujas propriedades de dependabilidade são então renderizadas como fórmula PCTL (*Probabilistic Computation Tree Logic*), e então é realizada a verificação do sistema. No Capítulo 3, é apresentada uma descrição mais detalhada de cada regra e um exemplo para ilustrar todo o processo de modelagem com o framework GODA.

Expressão	**Significado**
AND (n1;n2)	Satisfação em sequência de n1 e n2.
AND ($n1\#n2$)	Satisfação paralela de n1 e n2.
OR (n1;n2)	Satisfação em sequência de n1 ou n2, ou de ambos.
OR ($n1\#n2$)	Satisfação paralela de n1 ou n2, ou de ambos.
$n + k$	n deve ser satisfeito k vezes, com k>0.
$n\#k$	Satisfação paralela de k instâncias de n, com k>0.
$n@k$	Máximo k-1 tentativas de satisfazer n, com k>0.
opt(n)	A satisfação de n é opcional.
try(n)?n1:n2	Se n for satisfeito, n1 deve ser satisfeito; Senão, n2.
$n1\|n2$	Satisfação alternativa de n1 ou n2, nunca de ambos.
skip	Sem ação. Útil para expressões ternárias condicionais.

Tabela 2.1: Regras RGM presentes no GODA, onde n, n1 e n2 representam objetivos ou tarefas em um modelo de objetivos [5]

2.2 Planejamento Automatizado

Como discutido por [6], planejar é deliberar sobre ações executadas com um objetivo predefinido. Essa deliberação consiste em escolher e organizar ações que vão levar ao objetivo da melhor forma possível, considerando o resultado que se espera de cada ação. Inserido como área de estudo em Inteligência Artificial, o Planejamento Automatizado estuda essas deliberações utilizando recursos computacionais.

O Planejamento Automatizado é baseado em premissas sobre os domínios e os problemas. Essas premissas caracterizam o planejamento clássico. Para todos os problemas, os estados pertencem a um conjunto finito S, ou seja, após formalizado um problema, não há como incluir ou remover estados e também não há como avaliar problemas com quantidade infinita de estados. Cada estado também deve ser observável, o que significa que há conhecimento sobre qualquer estado do ambiente.

Outro elemento importante na formalização de um problema é a ação. Uma ação é uma modificação direta que gera a transição entre estados. As ações seguem a premissa de serem determinísticas, de forma que sempre um mesmo efeito é gerado por uma ação, dado que as condições iniciais da ação também são as mesmas. As ações são os únicos elementos capazes de causar mudanças no ambiente e isso garante que a solução do problema não sofra interferências externas.

Além disso, os problemas são resolvidos quando um objetivo é alcançado. Os objetivos devem ser bem definidos e restritos para garantir que os planejadores encontrem as soluções. Uma solução é uma sequência de ações que são escolhidas pelo planejador. Essa sequência é formada pelo planejador que cria as soluções a partir de algoritmos de busca e heurísticas. Porém, os planejadores só dispõe das informações definidas antes de sua execução, de forma que a busca por soluções é *offline*.

Outra premissa importante é a duração das ações. Em planejamento clássico, a duração é abstraída e todas as ações são consideradas como instantâneas, causando sempre uma mudança de estado.

Mas por que automatizar o planejamento? Considere o exemplo do aspirador de pó, como ilustrado na Figura 2.1. Imagine que você está em uma casa com dois cômodos, os quais estão sujos. Você então pega um aspirador de pó para limpar os cômodos. Aqui, temos a definição do problema: *limpar os cômodos*. Além disso, sabemos que eles estão sujos, ou seja, também sabemos o *Estado Inicial* do problema, que é ambos os cômodos estão sujos e o aspirador está no cômodo à esquerda, como mostrado na Figura 2.1.

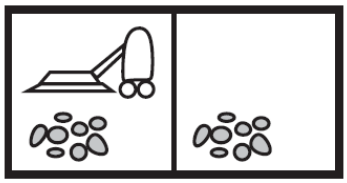

Figura 2.1: Representação do problema no seu estado inicial [1].

Agora, precisamos definir uma sequência de ações (através de um planejador) que vai nos levar ao objetivo, outra definição importante e que, nesse caso, é ambos os cômodos estão limpos. Vamos considerar que nosso aspirador pode realizar três ações: aspirar, mover-se à esquerda, mover-se à direita. Agora que temos o problema bem definido, com seu estado inicial, seu objetivo e suas ações, podemos criar um plano para que o objetivo seja cumprido utilizando um *Planejador*.

O planejador recebe como entrada todas as definições citadas acima e retorna como saída uma sequência de ações que é o *Plano*. Ele baseia as escolhas das ações e a ordem em que elas aparecem no plano em técnicas, algoritmos e heurísticas, tentando sempre encontrar um plano que atinge o objetivo definido e que seja o melhor possível. Além disso, é possível que o planejador também leve em consideração outros fatores na sua decisão, como por exemplo tempo gasto na execução de uma determinada ação ou eficácia em obter o resultado esperado de uma ação. Imagine que o aspirador, algumas vezes, não consiga remover toda a sujeira de um cômodo, ou seja, a ação de aspirar não necessariamente implica que o cômodo vai estar limpo e isso pode afetar o efeito que o planejador espera quando uma ação de aspirar é incluída no plano. A Figura 2.2 ilustra o processo de resolução de um problema.

Essas questões que envolvem propriedades dos problemas são fundamentais na área de Planejamento Automatizado. O exemplo do Problema do Aspirador de Pó possui propriedades do ambiente importante, a saber, ele é observável (o agente, que é o aspirador, consegue saber em qual estado ele se encontra durante a execução do plano),

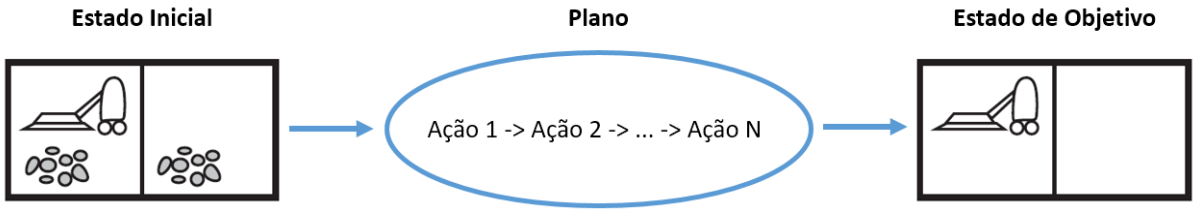

Figura 2.2: Transição do estado inicial para o estado de objetivo.

determinístico (todas as ações tem um único resultado), estático (todas as mudanças no ambiente só são causadas pelo agente, sem interferências externas) e discreto (a quantidade de ações possíveis é finita). Essas propriedades são o que caracterizam problemas de Planejamento Clássico, que é uma subárea de planejamento, assim como os problemas de Planejamento Não-Clássico, onde essas propriedades também estão presentes, mas são relaxadas de forma a se adequar às características do problema. Estes são problemas de mundo real e podem ser não-observáveis, dependentes de tempo, possuir ambiente dinâmico, conter ações cujos efeitos não foram previstos. Em [1], os autores discutem mais detalhadamente as diferenças entre os problemas clássicos e não-clássicos, com uma abordagem de relaxamento das propriedades citadas, nos Capítulos 10 e 11.

Consideradas essas propriedades dos problemas, deve-se então compreender como será encontrada a solução, ou seja, quais algoritmos e técnicas serão empregados. Para tanto, considere novamente o exemplo do Aspirador de Pó, para ilustrar a diferença entre algoritmos de busca e de planejamento (aqueles aplicados por planejadores). Na Figura 2.3, apresentamos uma representação das três ações mencionadas anteriormente (**S** - aspirar, **L** - mover-se à esquerda e **R** - mover-se à direita) e dos estados atingidos quando essas ações são executadas.

Note que essa representação é semelhante a um grafo e, portanto, seria interessante aplicar um algoritmo de busca de caminhos em grafos e assim, teríamos uma solução que seria um caminho do estado inicial até um dos dois possíveis estados de objetivo. Por outro lado, essa abordagem só é interessante quando o problema é escalável, como é o caso. Aqui, como só há dois cômodos, a quantidade de estados é pequena, se comparada por exemplo ao mesmo ambiente, porém com 10 cômodos. Ademais, esse problema ainda é determinístico, sempre que a ação aspirar é executada, o cômodo fica limpo, mas, se relaxamos essa suposição de forma que o aspirador não consiga aspirar toda a sujeira em 30% das vezes que a ação é executada, temos então uma situação nova, onde não é possível representar algumas situação, como por exemplo laço onde o aspirador sempre deixa sujeira no cômodo e tem que repetir a ação de aspirar indefinidamente. Assim, um algoritmo de busca nunca encontrará a solução, porque nessa situação, a representação

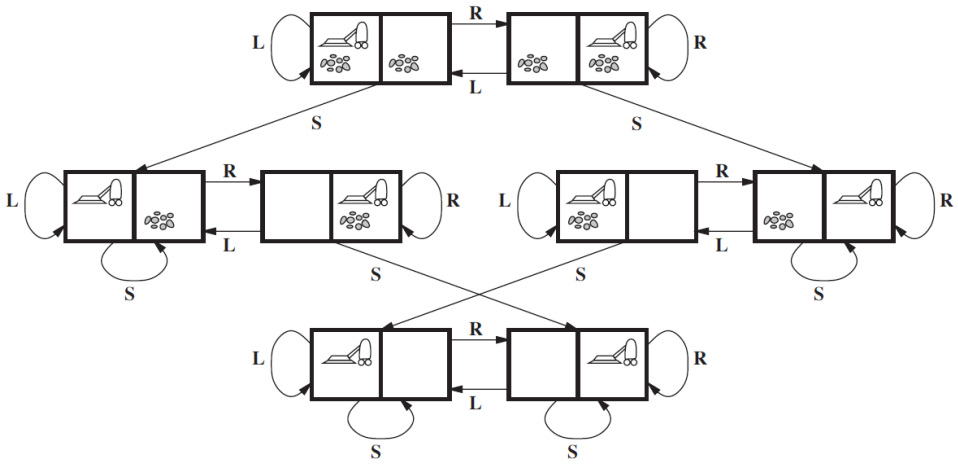

Figura 2.3: Diagrama com todos os possíveis estados e transições [1].

do problema impede o uso de desse tipo de algoritmo.

Por outro lado, uma abordagem que utilize algoritmos de planejamento não está sujeita a falhar pela presença de laços, uma vez que a forma de representar o problema é diferente. Nos algoritmos de busca, o plano nada mais é do que uma sequência de ações que levam do estado inicial ao estado de objetivo. Essa sequência é escolhida com base na busca realizada em uma representação em estrutura de dados do ambiente do problema. Já nos algoritmos de planejamento, os estados não são elementos de uma estrutura de dados, mas em sentenças lógicas que descrevem cada estado. As ações são representadas com precondições e efeitos, não apenas como transições entre os estados, e por fim, os objetivos também são sentenças lógicas. Dessa forma, o plano é definido a partir das descrições dos estados, seguindo as restrições das ações escolhidas, bem como seus efeitos.

2.2.1 Representação de Problemas de Planejamento

A representação de Problemas de Planejamento se dá pela formalização das três componentes principais do sistema: os estados, os objetivos e as ações. Para tanto a forma mais comum de representação é através da Lógica de Primeira Ordem (LPO). Nela, todos os componentes do problema são proposições formadas por átomos e conectivos lógicos, guiados por um sistema de regras de derivação. Segundo [1], a formalização via LPO possui quatro elementos principais: termo, sentença atômica, conectivos lógicos e quantificadores.

Termos são expressões lógicas que se referem a objetos, podendo assumir papéis de constante (objeto conhecido do domínio), variável (objeto desconhecido do domínio) ou função (que pode ser aplicada a objetos do domínio). Sentenças atômicas (ou átomos)

são fatos do problema. Um predicado seguindo de uma lista de termos, entre parêntese, é uma sentença atômica. Seguindo o exemplo do Aspirador de Pó, *CômodoEsq* é termo e *Limpo(CômodoEsq)* é uma sentença atômica com o predicado *Limpo* e que indica que *CômodoEsq* está limpo. Conectivos lógicos são símbolos de operadores lógicos usados para compor sentenças atômicas mais completas, que possuem mais de um predicado, como por exemplo, $Limpo(CômodoEsq) \wedge Limpo(CômodDir)$, que implica que ambos os cômodos estão limpos. E além destes, os quantificadores são símbolos utilizados para atribuir universalidade $\forall$ e existencialidade $\exists$ aos objetos.

Mais uma vez voltando ao Problema do Aspirador de Pó, podemos então, utilizando LPO, definir a representação do problema. Apresentamos então um ambiente com dois cômodos, um agente que é o Aspirador de Pó e o estado inicial onde ambos os cômodos estão sujos, conforme Figura 2.4.

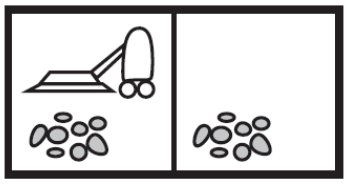

Figura 2.4: Representação do ambiente no seu estado inicial [1].

Sabemos também que o agente pode executar três ações, mas para incrementar a ilustração, vamos considerar que, quando o agente está no cômodo da esquerda, ele não pode mover-se à esquerda e, da mesma forma, quando ele está no cômodo da direita, não pode mover-se à direita. Então, para a ação mover-se à direita existe a precondição *Posição(CômodoEsq)*, uma vez que o aspirador só pode se mover para um cômodo vazio. Nesse exemplo, *Posição* é um predicado que indica em qual cômodo o aspirador está e *CômodoEsq* é um termo que representa o objeto Cômodo do problema. Agora, se a precondição é satisfeita, então nosso agente executará a ação e esta resultará em uma mudança no ambiente, ou seja, a ação produz um efeito e esse efeito pode ser representado pela sentença $Posição(CômodoDir) \wedge \neg Posição(CômodoEsq)$, indicando que de fato o aspirador se moveu para a direita. Assim, toda vez que a ação *Mover(CômodoEsq, CômodoDir)* for executada, o agente vai se mover à direita (note que as ações também são representadas como predicados em sentenças atômicas).

Criando definições para as outras ações do exemplo e para o estado de objetivo, temos então um problema definido e devidamente formalizado. Porém, a maioria dos planejadores utiliza uma representação ainda mais completa, as linguagens de planejamento. Na Seção 2.2.2, são apresentadas algumas das linguagens importantes e exemplos de como elas extrapolam a LPO e incluem novos componentes para a representação de problemas.

2.2.2 Linguagens de Planejamento

Conforme Seção 2.2.1, os problemas podem ser definidos quando se representa os estados, as ações e os objetivos. No contexto das linguagens de planejamento, essa divisão é feita de outra forma: Domínio e Problema. O Domínio é onde o ambiente é definido, ou seja, suas características, ações e predicados, que servem para explicitar as propriedades dos objetos. O Problema é onde são definidos os estados (inicial e objetivos), os objetos e o domínio ao qual está associado. Aqui, a definição dos estados é feita por sentenças como as descritas na seção 2.2.1, utilizando LPO e os predicados definidos no Domínio.

STRIPS

Stanford Research Institute Problem Solver (STRIPS) é ao mesmo tempo uma linguagem e um planejador, criados com um propósito específico de criar planos de ação para robôs. Apresentada em 1971 por [14], tanto a linguagem como o planejador ainda são referências e utilizados por outras linguagens, dadas sua relevância para a área. Essa relevância é efeito da forma como as ações são representadas e por ser possível aplicar estratégias de dividir e conquistar aos problemas definidos com essa linguagem.

A formalização se dá utilizando LPO (com algumas restrições) e dividindo em duas partes, domínio e problema, como discutido anteriormente. Em STRIPS, cada ação tem quatro informações: parâmetros, pré-condições, o que a ação adiciona na lista, o que a ação remove da lista (sendo os dois últimos a forma escolhida para representar o efeito causado pela ação, ou seja, o que é adicionado ou removido do ambiente). Formalmente, um problema de planejamento em STRIPS é uma tripla $P = (\Sigma, s_i, g)$ onde:

- $\Sigma = (S, A, \gamma)$ representa o domínio, em que:
  - $S = \{s_1, s_2, ...\}$ é o conjunto de estados possíveis;
  - $A = \{a_1, a_2, ...\}$ é o conjunto de ações possíveis;
  - $\gamma : S \times A \rightarrow S$ é a função de transição entre estados.
- $s_i \in S$ é o estado inicial
- g é um conjunto de predicados que descrevem o objetivo

Para facilitar o entendimento, é apresentado a seguir o Problema do Mundo dos Blocos. Considere que no problema existe uma mesa e três blocos sobre a mesa, que podem estar empilhados ou não. O objetivo aqui é mover os blocos para empilhá-los em uma certa ordem. A Figura 2.5 ilustra o exemplo apresentando um possível estado inicial e o objetivo, que é empilhar os blocos em ordem alfabética. Aqui, temos quatro objetos, Mesa, Bloco

A, Bloco B e Bloco C e eles estão dispostos de forma que B e A estão sobre a mesa e C está sobre A. Ao executar o plano, espera-se que A esteja sobre B, B esteja sobre C e C esteja sobre a mesa. Utilizando a sintaxe da linguagem STRIPS, uma forma de codificar o domínio desse exemplo é apresentada no Código 2.1.

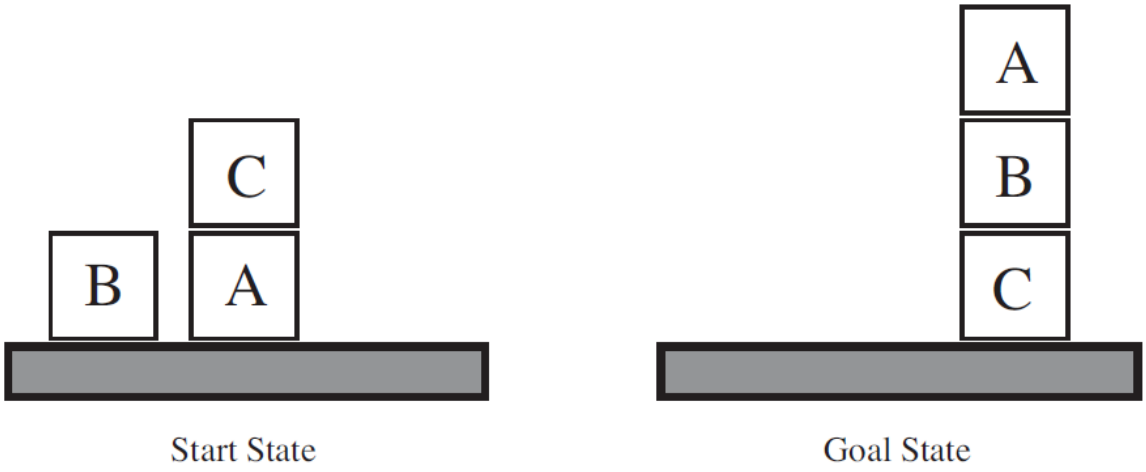

Figura 2.5: Ilustração do estado inicial e do objetivo do Mundo dos Blocos [1].

```
1  operator pickup(Obj)
2  pre: clear(Obj), onTable(Obj)
3  post: holding(Obj), ~clear(Obj), ~onTable(Obj)
4
5  operator putdown(Obj)
6  pre: holding(Obj)
7  post: clear(Obj), onTable(Obj), ~holding(Obj)
8
9  operator stack(Obj, UnderObj)
10 pre: clear(UnderObj), holding(Obj)
11 post: clear(Obj), on(Obj, UnderObj), ~clear(UnderObj), ~
       holding(Obj)
12
13 operator unstack(Obj, UnderObj)
14 pre: on(Obj, UnderObj), clear(Obj)
15 post: holding(Obj), clear(UnderObj), ~on(Obj, UnderObj), ~
       clear(Obj)
```

Código 2.1: Formalização STRIPS do domínio do Mundo dos Blocos [15]

Note que para essa versão do problema, existe quatro ações *pickup*, *putdown*, *stack* e *unstack* que faz com que um bloco se mova de uma mesa para outra, empilham dois blocos que estejam na mesma mesa e desempilham dois blocos empilhados, respectivamente. E, para completar a formalização do problema, o Código 2.2 do problema em STRIPS, com a definição dos objetos e dos estados inicial e final:

```
start (
 onTable(a),
 onTable(b),
 on(c, a),
 clear(b),
 clear(c)
)
goal(
 on(a, b),
 on(b, c),
 onTable(c)
)
```

Código 2.2: Formalização STRIPS do Problema do Mundo dos Blocos [15]

Tendo a definição do problema na linguagem, podemos também exibir a descrição formal, como segue:

- $\Sigma : STRIPSPlanningDomainBLOCKSWORLD$
- s_i : um estado do problema $(i = 0, 1, ..., n)$
  - $s_0 = \{onTable(a), onTable(b), on(c, a), clear(b), clear(c)\}$
- $g \in L$
  - $g = \{on(a, b), on(b, c), onTable(c)\}$

Assim, podemos utilizar um planejador para criar um plano que, partindo do estado inicial s_0, leve o agente ao estado de objetivo g, ou seja, que faz com que os blocos fiquem empilhados da forma definida acima. Um possível plano é apresentado na sequência de ações:

1. unstack(c, a)
2. putdown(c)
3. pickup(b)

4. stack(b, c)
5. pickup(a)
6. stack(a, b)

PDDL

A linguagem Planning Domain Definition Language (PDDL) veio como uma forma de viabilizar a IPC*(International Planning Competition)* realizada em 1998, durante a conferência internacional AIPS (*Artificial Intelligence Planning Systems*). A ideia dos idealizadores da linguagem era possibilitar aos participantes da competição uma padronização quanto à formalização dos problemas que ainda não existia até então. Nas edições seguintes, a linguagem continuou a ser utilizada e novas versões foram sendo criadas para se adequar aos novos desafios da área de planejamento. Além disso, PDDL se tornou uma das principais linguagens utilizadas em planejamento, como discutido por [16].

Como mostrado por [17], a linguagem também se baseia na definição de um domínio e de um problema. Porém, PDDL possui características presentes nas definições que conferem à linguagem maior expressividade. Essas características foram sendo acrescentadas a cada versão nova da linguagem, ou seja, acrescentado elementos novos, como funções, constantes, métricas para os planos, ações com duração em unidades de tempo, preferências, entre outros. E por se tornar cada vez mais robusta, a formalização em PDDL é mais abrangente e aplicável a problemas reais, em se tratando de Planejamento Automatizado. No Código 2.3 abaixo, o mesmo exemplo do Mundo dos Blocos é apresentado, mas agora utilizando PDDL:

```
1 (define (domain blocksworld)
2 (:requirements :strips :equality)
3 (:predicates (clear ?x)
4              (onTable ?x)
5              (holding ?x)
6              (on ?x ?y))
7
8 (:action pickup
9   :parameters (?ob)
10  :precondition (and (clear ?ob) (onTable ?ob))
11  :effect (and (holding ?ob) (not (clear ?ob)) (not (onTable
      ?ob))))
12 (:action putdown
13  :parameters  (?ob)
```

```
14   :precondition (and (holding ?ob))
15   :effect (and (clear ?ob) (onTable ?ob)
16                (not (holding ?ob))))
17 (:action stack
18   :parameters  (?ob ?underob)
19   :precondition (and  (clear ?underob) (holding ?ob))
20   :effect (and (clear ?ob) (on ?ob ?underob)
21                (not (clear ?underob)) (not (holding ?ob))))
22 (:action unstack
23   :parameters  (?ob ?underob)
24   :precondition (and (on ?ob ?underob) (clear ?ob))
25   :effect (and (holding ?ob) (clear ?underob)
26                (not (on ?ob ?underob)) (not (clear ?ob)))))
```

Código 2.3: Formalização em PDDL do Domínio do Mundo dos Blocos [15]

Note que no Código 2.3, as quatro ações estão presentes, porém as expressões são mais completas e alguns novos constructos são utilizados, como a definição de predicados. Já no Código 2.4, é apresentado o arquivo de problema.

```
1 (define (problem pb4)
2    (:domain blocksworld)
3    (:objects a b c d)
4    (:init  (onTable a) (on b a) (on c b) (on d c) (clear d))
5    (:goal (and (on b a) (on c b) (on a d))))
```

Código 2.4: Formalização em PDDL do Problema do Mundo dos Blocos [15]

A mais recente versão da linguagem é o PDDL3.1. A partir do PDDL3, três novas funcionalidades foram introduzidas: (i) restrições de trajetória de estados; (ii) preferências; e (iii) objetos fluentes. As restrições de trajetória são expressões lógicas que definem a aceitação dos estados pertencentes ao plano. Preferências também são restrições, mas que possuem uma característica facultativa, de forma que sua satisfação não é necessária (preferências, em geral, são relacionadas a métricas que avaliam a qualidade dos planos gerados). Por fim, objetos fluentes são uma extensão incorporada das versões anteriores que já aceitavam tipagem de objetos, porém apenas tipos numéricos e, agora na versão 3.1, qualquer tipo de objeto é aceito.

2.2.3 Abordagem de Planejamento HTN

Outra formalização para problemas de planejamento é o Hierarchical Task Networks (HTN). No modelo HTN, a abordagem de definição do ambiente é um pouco diferente,

bem como a forma de busca por planos. Em PDDL (e também em STRIPS), o domínio possui a definição das ações que o agente é capaz de executar. Essas ações devem ser expressões compostas por predicados definidos também no domínio e todas as ações são primitivas, ou seja, não é possível compor ações ou derivá-las em expressões menores. Esse é o ponto principal da abordagem HTN, a capacidade de utilizar ações abstratas. Além disso, o objetivo nessa abordagem não é criar uma sequência de ações que leva a um estado onde todas as condições definidas como objetivo são satisfeitas, mas sim de completar *task networks*, que podem incluir mais do que apenas satisfazer condições.

Task networks, segundo [2], são coleções de tarefas que precisam ser realizadas, juntamente com restrições na ordem de execução das tarefas, na forma como as variáveis são instanciadas e quais literais devem ser verdadeiros antes ou depois que cada tarefa é realizada. Através dessas redes é que o planejamento é criado, mais especificamente, com o uso de *métodos*. Os métodos recebem como entrada uma *task network* d e uma tarefa não-primitiva α que será executada. Então, para executar α , o método realiza todas as tarefas da rede d, se atentando também para as restrições que d contém. Para ilustrar os conceitos, considere que um agente queira realizar a tarefa de ir à São Paulo. Considere que esse mesmo agente esteja em Brasília e queira fazer o trajeto de carro. Para representar então esse problema, basta criar um predicado Ir(A, B), onde A e B são cidades. Então, o agente deve realizar Ir(Brasília, São Paulo) para resolver o problema. Porém, Ir é uma tarefa não-primitiva, uma vez que há várias tarefas menores que compõe essa tarefa, como por exemplo abastecer o carro, separar dinheiro para o pedágio, dirigir até São Paulo e outras. Então, é preciso criar um método que resolva essa tarefa Ir(Brasília, São Paulo), como por exemplo na Figura 2.6.

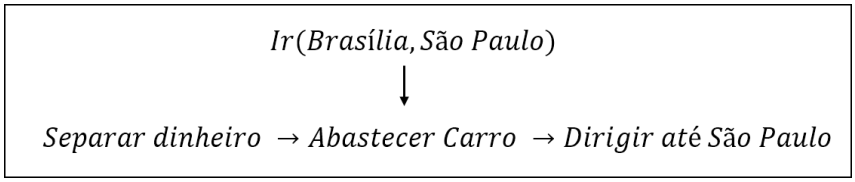

Figura 2.6: Definição do método Ir(A, B).

Como HTN não possui uma sintaxe definida e padronizada, para efeito de ilustração e também para começar a introduzir o planejador utilizado nesse trabalho, será apresentada a definição utilizada no Pyhop (uma explicação mais detalhada do planejador e seu funcionamento é apresentada na Seção 2.2.5). Segundo a definição dos autores, o Pyhop utiliza três arquivos para formalizar os problemas em HTN: um arquivo de definição de operadores, um arquivo de definição de métodos e um arquivo de definição do problema.

O Código 2.5[2] apresenta o arquivo de operadores para o problema do Mundo dos Blocos.

```
1 def pickup(state,b):
2     if state.pos[b] == 'table' and state.clear[b] == True and
          state.holding == False:
3         state.pos[b] = 'hand'
4         state.clear[b] = False
5         state.holding = b
6         return state
7     else: return False
```

Código 2.5: Declaração de operando *pickup* para o Problema do Mundo dos Blocos [18]

Aqui é possível notar que a própria sintaxe da linguagem Python é utilizada para definir a formalização da abordagem HTN, dando muito mais liberdade e maior capacidade de escrita para quem utiliza esse planejador. Ainda dentro da ilustração, o Código 2.6 é referente à definição dos métodos, responsáveis por decompor as tarefas não-primitivas, também utilizando a linguagem Python para criar uma representação HTN.

```
1 def moveb_m(state,goal):
2     for b1 in all_blocks(state):
3         s = status(b1,state,goal)
4         if s == 'move-to-table':
5             return [('move_one',b1,'table'),('move_blocks',
                  goal)]
6         elif s == 'move-to-block':
7             return [('move_one',b1,goal.pos[b1]), ('
                  move_blocks',goal)]
8         else:
9             continue
10    b1 = pyhop.find_if(lambda x: status(x,state,goal) == '
         waiting', all_blocks(state))
11    if b1 != None:
12        return [('move_one',b1,'table'), ('move_blocks',goal)
             ]
13    return []
```

Código 2.6: Declaração do método *move_b* para o Problema do Mundo dos Blocos [18]

Por último, a definição do problema em si, que apesar de seguir a sintaxe do Python e o conceito de redes de tarefas do HTN, possui semelhanças com PDDL e STRIPS, no sentido

[2]Extraído de: https://bitbucket.org/dananau/pyhop.

de que esse arquivo é o que contém as definições de estado de inicial e de objetivos. Porém, como discutido anteriormente, o estado de objetivo não é necessariamente um conjunto de expressões como em PDDL e STRIPS, mas sim uma rede de tarefas a serem decompostas, que na definição do Pyhpo são instâncias de um objeto do tipo Goal, como apresentado no 2.7

```
1 state1 = State('state1')
2 state1.pos={'a':'b', 'b':'table', 'c':'table'}
3 state1.clear={'c':True, 'b':False,'a':True}
4 state1.holding=False
5
6 goal1a = Goal('goal1a')
7 goal1a.pos={'c':'b', 'b':'a', 'a':'table'}
8 goal1a.clear={'c':True, 'b':False, 'a':False}
9 goal1a.holding=False
10
11 print_goal(goal1b)
12
13 pyhop(state1,[('move_blocks', goal1a)], verbose=1)
```

Código 2.7: Declaração do problema para o Mundo dos Blocos [18]

2.2.4 Algoritmos de Planejamento

O objetivo dessa seção é apresentar quais são de fato os mecanismos utilizados em planejamento para encontrar planos. Os algoritmos apresentados aqui foram escolhidos pela sua relevância para a proposta e para a área de planejamento, e divididos em: (i) algoritmos de busca desinformada (aqueles que não possuem informações adicionais sobre os estados além daquelas informadas na definição do problema); (ii) algoritmos de busca informada (que possuem conhecimentos relativos ao problema que vão além da definição do problema) [1].

Busca em Largura e Profundidade

Os algoritmos de Busca em Largura e Busca em Profundidade são semelhantes em relação a seu uso em estruturas de dados como grafos e árvores. A Busca em Largura é realizada expandindo sempre todos os filhos de um nó e em seguida, todos os sucessores de cada filho, um por vez. Dessa forma o algoritmo sempre procura pelo nó mais *raso*, ou seja, com menor altura. Assim, os nós visitados vão sendo armazenados em uma fila FIFO (*First In First Out*) e expansão de nós continua até que todos tenham sido expandidos.

Já na Busca em Profundidade, a expansão sempre ocorre buscando pelo nó mais *profundo* dentro da estrutura, ou seja, com maior altura. Dessa forma, sempre que um nó é expandido, ele é marcado e então a busca continua pelo próximo nó de maior altura. Semelhante à Busca em Largura, uma fila também é utilizada para armazenar os nós visitados, porém aqui a ordem é LIFO (*Last In First Out*), para garantir que seja possível voltar depois que o nó mais profundo for visitado.

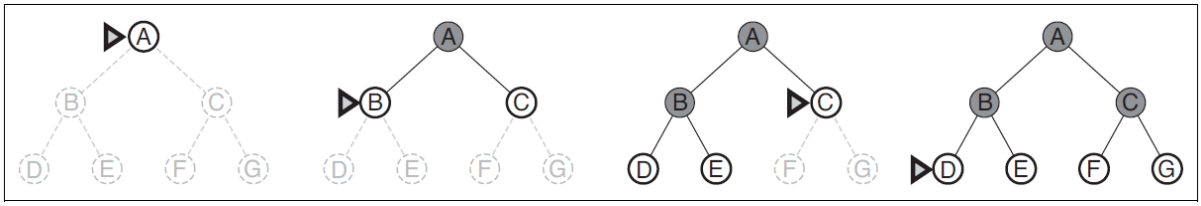

Figura 2.7: Busca em Largura em uma árvore binária. O nó a ser expandido em seguida é apontado pelo indicador [1].

Apesar da facilidade em aplicar esses algoritmos é muito comum utilizar estruturas como árvores e grafos na definição de problemas e de implementá-los, mas existem algumas considerações a serem feitas em relação a eficiência desses algoritmos. Em [1], os autores mostram que a Busca em Largura possui complexidade de tempo $O(b^{d+1})$ e complexidade de espaço $O(b^d)$, onde b representa a quantidade de nós em cada nível e d é a profundidade do nó que contém a solução. Essa complexidade pode levar a valores muito altos, especialmente de espaço, uma vez que para uma estrutura não muito grande, com $b = 10$ níveis e $d = 10$, considerando que cada nó ocupe 1 kilobytes de espaço, então o algoritmo vai precisar de 10 terabytes de espaço para expandir todos os nós, o que já se torna inviável para computadores pessoais, por exemplo.

A Busca em Profundidade possui complexidade de tempo semelhante $O(b^m)$, onde m é a altura máxima de um nó. Porém, nos casos em que a solução está em um nó mais raso, então $m > d$, o que pode aumentar significativamente a quantidade de tempo necessária para a execução, devido ao fato do algoritmo sempre buscar os nós mais profundos, expandindo muitos nós sem necessidade até chegar na solução, que está em um nível mais raso).

Por outro lado, a complexidade de espaço desse algoritmo é $O(bm)$ e então, seguindo o mesmo cálculo e com as mesmas condições, o algoritmo de Busca em Profundidade usaria apenas 100 Kilobytes de memória, ou seja, 10^{10} vezes menos espaço, o que é viável a qualquer computador moderno, e muito mais interessante para usar em aplicações cuja definição demanda estruturas grandes. Em [1], os autores discutem que essa característica fez com que o algoritmo fosse largamente utilizado em várias áreas de Inteligência Artificial.

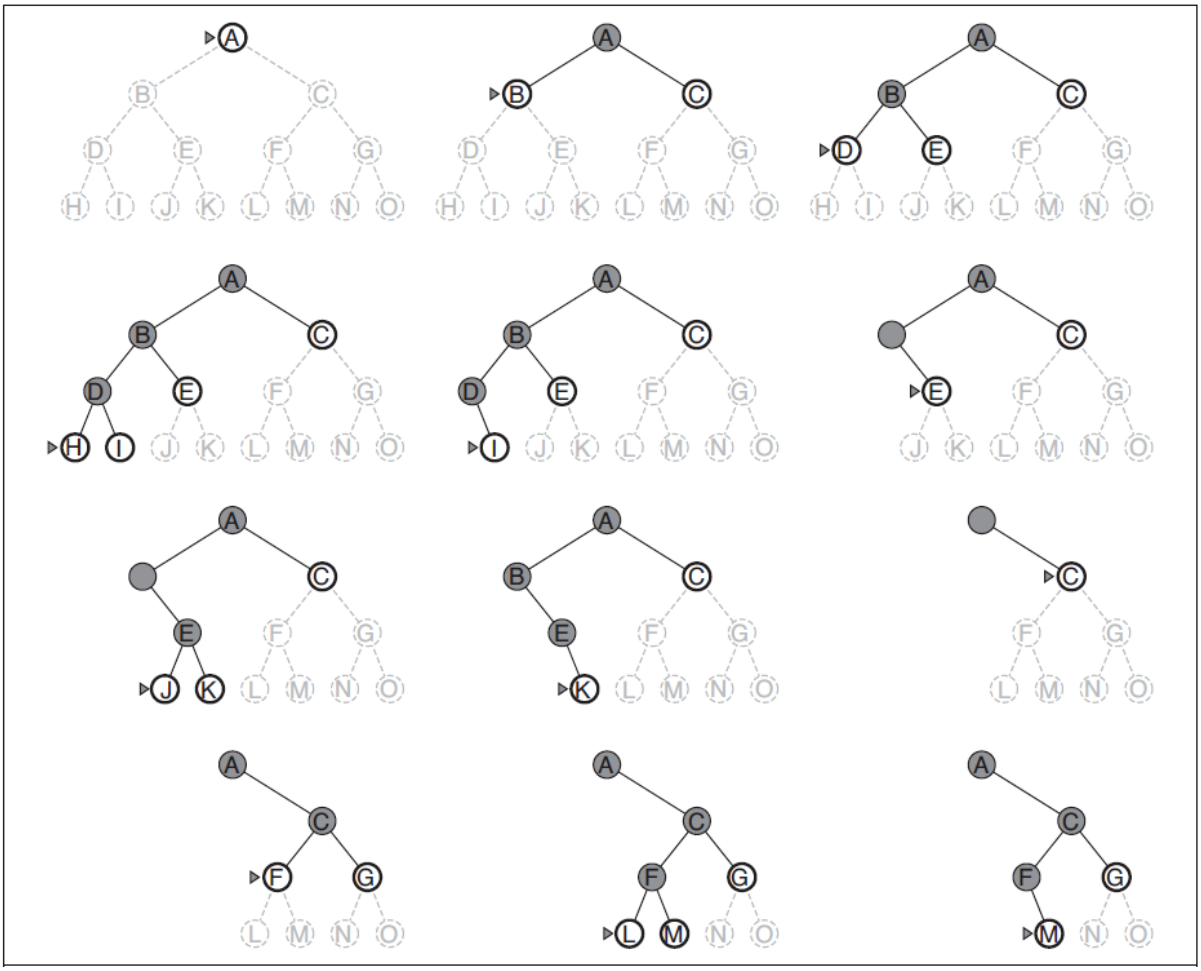

Figura 2.8: Busca em Profundidade em uma árvore binária. O nó a ser expandido em seguida é apontado pelo indicador. (Fonte [1]).

Greedy Best-First

Esse algoritmo já não é como os anteriores. Aqui há informações adicionais baseadas em heurísticas que são utilizadas para escolher os nós mais próximos do objetivo, partindo da premissa que esse formato de busca vai levar à solução mais rapidamente, por isso esse é um algoritmo guloso. Essa abordagem pode ser muito eficiente para algumas aplicações e em muitas vezes melhor que os algoritmos de Busca em Largura ou em Profundidade.

Porém, o *Greedy Best-First* possui uma característica de não-completude, ou seja, é possível que durante a expansão o algoritmo chegue a um nó folha que não possui transições para outros nós e a solução não seja encontrada. Isso ocorre quando esse nó *sem-saída* está próximo à solução (ou então, a heurística utilizada para a expansão indica que esse nó deve ser expandido em seguida) mas não faz parte do caminho que leva da raiz até o nó solução. Além disso, os nós adjacentes que fazem parte do caminho e deveriam

ter sido expandidos no lugar deste não estão tão próximos (ou sua heurística indica que eles não devem ser expandidos). Dessa forma, o algoritmo entra em *loop* infinito, já que ele fica alternando entre o nó "sem-saída"e o nó anterior sem nunca expandir outros nós.

Em termos de complexidade, tanto para complexidade de tempo como para complexidade de espaço, o *Greedy Best-First* possui $O(b^m)$. Essa complexidade pode ser alta dada a escala do problema em que se deseja aplicar o algoritmo. Porém, a heurística utilizada, bem como as particularidades do problema (Problemas formalizados utilizando grafos com alta conectividade podem minimizar a não-completude do algoritmo, fazendo com que uma solução seja encontrada em grande parte das execuções), podem diminuir o custo e tornar o algoritmo bastante eficiente.

A*

O algoritmo A* (lê-se *A-star* ou A-estrela) também é do tipo *Best-First*, ou seja, ele também busca pela melhor opção primeiro, mas não de forma "gulosa"e sim utilizando um combinação de custo e heurística. Suponha que $g(n)$ seja o custo para chegar a um determinado nó e $h(n)$ o custo para chegar de um determinado nó até a solução. Assim, a expansão pode ser feita buscando por nós que minimizem esse dois tipos de custo, por exemplo.

Esse algoritmo é melhor que a abordagem gulosa, para casos onde existem nós próximos à solução, mas que não levam à solução ("sem-saída"). Como aqui existem pelo menos dois fatores para avaliar se um nó deve ser expandido ou não, então o risco de chegar a um nó que não pertence ao caminho é bem menor e, o mais importante, é possível voltar para um nó anterior e considerar as outras opções, diferentemente do algoritmo guloso que nessas situações entra em *laço* infinito e já não consegue encontrar a solução.

Contudo, o algoritmo também possui desvantagens, especialmente relacionadas à complexidade de espaço. Como a expansão sempre procura pelo melhor estado seguinte para ser visitado, em uma situação com vários nós solução é possível que o algoritmo acabe por escolher as de menor custo, já que também são soluções, mesmo que isso signifique não encontrar as soluções ótimas. Assim, por vezes se torna impraticável utilizar o A* para encontrar uma solução ótima de um problema [1].

Algoritmo HTN

Como explicado anteriormente, a abordagem HTN utiliza *task networks* para encontrar a solução de um determinado problema. Essas podem conter ações primitivas bem como ações não-primitivas. Para as redes cujas ações são todas primitivas, o algoritmo então as seleciona de forma que todas as restrições sejam satisfeitas, semelhante a um agendamento

de tarefas, ou seja, o planejador cria apenas uma ordenação das ações para que se chegue à solução.

Contudo, o caso geral para problemas formalizados em HTN é aquele cujas redes de tarefas possuem ações não-primitivas. E aqui, o algoritmo deve se atentar ao fato de que esse tipo de ações não podem ser executadas diretamente, mas antes devem ser decompostas através do uso dos métodos até que se tenha uma rede contendo apenas ações primitivas, que podem então ser executadas.

Segundo [2], vários sistemas diferentes foram desenvolvidos para planejamento com HTN usando heurísticas e outros tantos modelos foram apresentados em periódicos como tentativas de formalização da abordagem HTN. Ainda segundo os autores, a essência dessas tentativas pode ser vista no pseudocódigo da Figura 2.9.

1. Input a planning problem **P**.
2. If **P** contains only primitive tasks, then
   resolve the conflicts in **P** and return the result.
   If the conflicts cannot be resolved, return failure.
3. Choose a non-primitive task t in **P**.
4. Choose an expansion for t.
5. Replace t with the expansion.
6. Use critics to find the interactions among the tasks in **P**,
   and suggest ways to handle them.
7. Apply one of the ways suggested in step 6.
8. Go to step 2.

Figura 2.9: Procedimento básico para planejamento em HTN [2].

No entanto, as vantagens de se poder expressar melhor os problemas também trazem complexidade. Nos experimentos realizados por [3], até mesmo os problemas que não possuem tarefas não-primitivas, mas com uso de variáveis e com a condição de que todas as *task networks* sejam ordenadas pertencem à classe dos problemas NP-completos, que apresentam casos de solução computacional ótima (para determinados domínios, uma vez que não existe verificabilidade para classes de problemas NP-completos). E mais ainda, alguns problemas não podem ser resolvidos, como mostrado na Figura 2.10.

Complexity of HTN Planning

Restrictions on non-primitive tasks	Must every HTN be totally ordered?	Are variables allowed?	
		no	yes
none	no	Undecidable	Undecidable[β]
	yes	in EXPTIME; PSPACE-hard	in DEXPTIME; EXPSPACE-hard
"regularity" [α]	doesn't matter	PSPACE-complete	EXPSPACE-complete
no non-primitive tasks	no	NP-complete	NP-complete
	yes	Polynomial time	NP-complete

[α] At most one non-primitive task, which must follow all primitive tasks.
[β] Even if the planning domain is fixed in advance.

Figura 2.10: Complexidade de problemas de Planejamento em HTN (Fonte [3]).

2.2.5 Planejadores

Essa seção apresenta alguns planejadores disponíveis na literatura, incluindo o planejador utilizado durante o desenvolvimento deste trabalho. As características, linguagens ou abordagens que implementam, com intuito de ilustrar o conjunto de ferramentas já existentes para Planejamento Automatizado.

JavaGP/ EmPlan

Esses dois planejadores são baseados no algoritmo Graphplan[3] que utiliza grafos como estruturas principais para definição do problema e para busca por soluções. Ambos são orientados a resolver problemas clássicos da literatura de planejamento. O JavaGP[4] é um planejador desenvolvido na linguagem Java[5] que implementa alguns recursos da linguagem PDDL e utiliza o gerador de analisador sintático JavaCC[6]. Seu foco no entanto é a implementação da linguagem STRIPS e o suporte à linguagem PDDL foi adicionado posteriormente. Já o planejador EmPlan[7] foi desenvolvido em C++ e utiliza Lex e Yacc[8] como analisadores. O EmPlan também utiliza o algoritmo Graphplan e também objetiva o uso de STRIPS.

O JavaGP foi considerado nas fases iniciais de definição da proposta deste trabalho e utilizado para avaliar a adequação do modelo proposto ao domínio de planejamento. Porém, por questões discutidas na Seção 3.1, esse planejador foi substituído, assim como a escolha da linguagem de planejamento também foi alterada.

[3] https://www.cs.cmu.edu/~avrim/graphplan.html
[4] https://github.com/pucrs-automated-planning/javagp
[5] https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html
[6] https://javacc.org/
[7] http://emplan.sourceforge.net/
[8] http://dinosaur.compilertools.net/

SGPLan

O SGPlan é um planejador que utiliza a abordagem HTN. Ele funciona de forma que problemas grandes sejam particionados em problemas menores, com seus próprios objetivos e, em seguida, combinados para atingir o objetivo principal. Essa característica foi implementada partindo da premissa de que subproblemas possuem menos restrições e menor complexidade, quando comparados com o problema maior. Assim, o planejador consegue ser mais eficiente e aproveitar a estrutura de redes de tarefas do HTN.

Esse planejador utiliza o planejador *Metric-FF*[9] para resolver os subproblemas e combina os resultados para encontrar a solução do problema original. Além de usar a abordagem HTN, o planejador também implementa recursos da linguagem PDDL e inclusive faz a conversão entre as duas linguagens. Assim como o JavaGP, esse planejador também foi considerado na elaboração deste trabalho, porém não foi adequado à proposta.

SHOP/Pyhop

O Simple Hierarchical Ordered Planner (SHOP)[10] é um sistema de Planejamento Automatizado independente de domínio. Ele é baseado em decomposição ordenada de tarefas, que é uma forma de planejamento da abordagem HTN, ou seja, o planejamento coloca as tarefas na mesma ordem em que serão executadas. Esse planejador também foi utilizado na IPC 2002, onde resolveu quase 1000 problemas e recebeu quatro prêmios[11].

A versão mais recente do SHOP é o SHOP2, escritas em Lisp[12] e também foi criada uma versão em Java, o JSHOP2 que utiliza uma nova forma de compilação dos planos e pode ser otimizado para uma execução mais rápida que a versão em Lisp.

Existe ainda uma versão desenvolvida em Python, o Pyhop. Segundo os desenvolvedores, o algoritmo de planejamento de Pyhop é como o do SHOP, mas com várias diferenças que devem facilitar a integração com programas de computador comuns[13]. No Pyhop, a abordagem HTN é utilizada, mas como uma abstração da sintaxe e dos componentes da linguagem Python, o que facilita a definição de métodos e operadores, uma vez que basta utilizar funções e variáveis comuns que estão presentes em Python, além do fato de que a linguagem é mais portável para integração com outros *softwares*, como especificado pelos próprios desenvolvedores. O uso dessa portabilidade foi testada nesse trabalho com a integração entre o GODA desenvolvido sobre Java e o Pyhop).

Na definição do problema, os estados são representados por objetos e variáveis de Python, ao invés de proposições lógicas, como em PDDL, por exemplo. Os operadores

[9] https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html
[10] https://www.cs.umd.edu/projects/shop/index.html
[11] https://www.cs.umd.edu/projects/shop/description.html
[12] http://lisp-lang.org/
[13] https://bitbucket.org/dananau/pyhop

e métodos do HTN são abstraídos para funções em Python, cuja entrada é um objeto de estado. Assim, é possível criar várias funcionalidades e incorporá-las aos métodos e operadores, uma vez que não há restrições de expressividade dada a capacidade de escrita de Python. Esse foi um dos motivos que levaram à escolha do Pyhop como ferramenta para implementação da proposta deste trabalho. Maiores informações sobre a ferramenta são apresentadas na Seção 3.1.

HTNPlan-P

O HTNPlan-P [19] é um planejador para HTN cujo foco é a utilização de preferências. Baseado no SHOP2, o HTNPlan-P estende a implementação do SHOP2 e acrescenta suas definições de preferência. Partindo do PDDL3, os autores inserem essa funcionalidade no HTN utilizando uma linguagem específica, criada também pelos autores, que define o uso de preferências.

O planejador utiliza um algoritmo de poda-e-ramificação, especializado para a a aplicação na linguagem citada. O algoritmo também é permeado de heurísticas que melhoram a sua performance, outro fator priorizado pelos autores. Em comparações com o SGPlan, HTPlan-P e outros planejadores apresentados na *International Planning Competition* (IPC) de 2006, os resultados mostram uma performance semelhante ou superior aos outros algoritmos avaliados, no quesito de qualidade de plano gerado.

2.3 Trabalhos Correlatos

2.3.1 Análise em Dependabilidade e Automação em BPMN

Segundo [4], a modelagem orientada a objetivos é muito utilizada para especificação de *software*. Porém, nas organizações, a abordagem para modelagem de processos de negócio, incluindo aqueles voltados à Engenharia de *Software*, mais comumente utilizada é o Business Process Model (BPM). Esse trabalho apresenta uma forma de compor a modelagem de sistemas por objetivos e a perspectiva de processos de negócio.

Os autores propõe então, uma conversão entre o CRGM e o Business Process Model Notation (BPMN)[14], criando uma equivalência entre os elementos presentes no CRGM e os elementos BMPN. Essa equivalência é feita de acordo com as regras RGM Tabela **??**. Em seguida, as regras do GODA são convertidas em indicações de fluxo de execução no BMPN, como por exemplo a regra $AND(n_1 \# n_2)$ que especifica a execução das regras n_1 e n_2, de forma paralela. Segundo o modelo apresentado, n_1 e n_2 são convertidos em atividades e o fluxo de execução é determinado por um *paralell gateway* (caminho paralelo)

[14] http://www.bpmn.org/

para indicar que as atividades devem ser realizadas ao mesmo tempo, como mostrado na Figura 2.11.

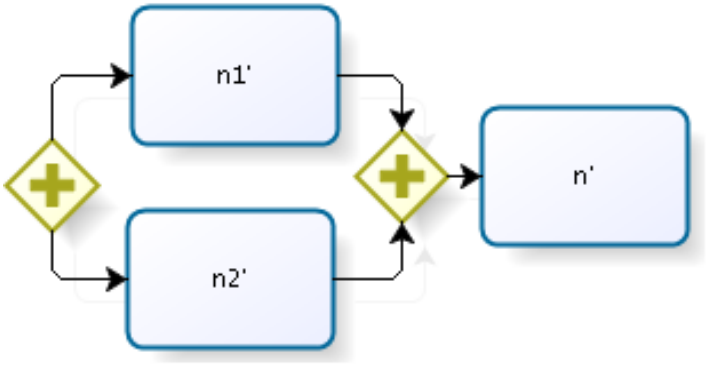

Figura 2.11: Processo BMPN para a regra **AND(n1#n2)** [4].

Juntamente como o modelo de conversão, os autores apresentam uma ferramenta que realiza esse papel. Essa ferramenta baseia-se no plugin Taom4e com o GODA e gera a conversão em formato de texto, seguindo a notação BMPN 2.0[15].

Assim como o trabalho apresentado nesse manuscrito, os autores propõe uma nova forma de enxergar processos, utilizando uma ferramenta mais utilizada para tanto, que é a modelagem BMPN e, assim, facilitando o entendimento por parte dos *stakeholders* dos modelos Tropos.

2.3.2 Adaptação Dinâmica de Processos

Em [20], a autora apresenta uma forma de fornecer mecanismos para adaptação dinâmica de processos em sistemas de informação orientados por processos. O trabalho traz uma abordagem que é baseado em contextos e Planejamento Automatizado.

A tese do trabalho está na obtenção de informações durante a execução de um processo que caracterizam uma necessidade de mudança no processo e que são apontadas pelos contextos. Considerando então essa influência dos contextos, a autora propõe uma abordagem para conceber a adaptação dinâmica de processos como uma tarefa de Planejamento Automatizado. Assim, e baseada na teoria de sistemas de informação orientados a contextos, essa abordagem mantém o alinhamento dos processos com os objetivos, de forma que a caracterização gerada pelos contextos serve de guia para a tarefa de re-planejar os caminhos dentro dos processos.

Para tanto, é utilizado o GCAdapt, um framework para gerenciamento de contextos na adaptação dinâmica de processos. A função do GCAdapt é supervisionar o processo, com o

[15]http://www.omg.org/spec/BPMN/2.0/About-BPMN/

intuito de descobrir novas situações que demandem um comportamento diferente, ou seja, um re-planejamento do processo, sempre baseado no conhecimento provido pelo contexto. Então, quando uma situação precisa ser re-planejada, entra em ação o planejador SAPA[16], que executa a tarefa de re-planejamento e devolve para o GCAdapt um novo plano de execução do processo. Esse procedimento é repetido sempre que uma situação nova com necessidade de mudança é detectada pelo GCAdapt.

Portanto, a autora propõe uma forma de utilizar o Planejamento Automatizado como ferramenta para melhorar os processos das organizações, da mesma forma que nesse manuscrito é apresentado um modelo que aproveita o Planejamento Automatizado para criar uma nova ótica sobre os processos modelos por objetivos.

[16]http://rakaposhi.eas.asu.edu/sapa.html/

Capítulo 3

Proposta

Neste capítulo é apresentada a proposta e a implementação da solução de conversão de um modelo orientado a objetivos para uma ferramenta de planejamento automático baseado em HTN.

3.1 Histórico da Proposta

Em se tratando de um trabalho envolvido na área de Planejamento Automatizado, a primeira preocupação foi encontrar uma opção de linguagem ou framework que viabilizasse a implementação da proposta. Como o foco do trabalho é uma conversão de modelos, também existiu uma preocupação na adequação da linguagem escolhida de forma que as relações entre conceitos dos dois domínios não prejudicassem o valor semântico deles. Para tanto, a linguagem PDDL foi escolhida, por uma quantidade de fatores, mas especialmente pela praticidade em encontrar planejadores mais completos e pela robustez da linguagem, que oferece vários recursos. Os planejadores avaliados foram JavaGP/EmPlan, SGPlan, HTNPlan-P, e SHOP/Pyhop.

Em seguida, foi feito um estudo sobre as opções de planejadores que implementam os recursos oferecidos pela linguagem PDDL e que possuíam relevância para o trabalho. Como até então não havia sido feita a escolha do framework GODA como o modelo a ser convertido, os recursos escolhidos como necessários foram selecionados de forma arbritária e com menos peso, enquanto a usabilidade dos planejadores avaliados e a quantidade desses recursos implementados por eles foram fatores mais decisivos para a escolha.

Ainda durante a fase de escolha de um planejador, foi feita a escolha do GODA como modelo a ser convertido. Essa escolha foi motivada por duas razões: (1) pela característica do modelo em avaliar contextos e *runtime* de um sistema de software e (2) pelo potencial de avanço que o uso de um novo domínio para análise poderia trazer. Como discutido na Seção 2.1.3, o framework GODA, durante a fase de análise de dependências, avalia

a execução dos sistemas para compreender a adequação deste aos requisitos e objetivos levantados durante a fase de especificação. Já no domínio de planejamento, os planejadores apresentam soluções (planos) para problemas, sempre buscando pela melhor delas, o que é feito através de algoritmos de planejamento que executam as ações do domínio para chegar ao objetivo. Aqui também há uma preocupação com a definição do domínio e do problema, que afeta completamente a execução do planejador e também a sua saída, característica que foi interfaceada com a qualidade do GODA em também gerar resultados que possam influenciar na definição dos modelos.

PDDL e JavaGP

Assim, a primeira tentativa de conjugar o modelo GODA com a linguagem PDDL foi criada. Nela, cada elemento do modelo GODA (objetivo ou tarefa) era tratado como objeto do problema; as regras de decomposição foram convertidas para ações do domínio; e foram criados dois predicados (*completed* e *not-completed*) para indicar se uma tarefa foi executada ou se um objetivo foi atingido. O objetivo do problema então foi definido como a aplicação do predicado *completed* ao objetivo principal do modelo, de forma que o planejador deveria procurar por um plano que, executando os outros objetivos e tarefas, chegasse ao objetivo no topo do modelo. Já o estado inicial, foi definido com a aplicação do predicado *not-completed* a todos os objetos, significando que nenhuma tarefa havia sido executada e nenhum objetivo havia sido satisfeito.

Apesar da proposta aparentar promissora, logo foi constatado que esse não era o caminho a ser seguido. Considerando a Figura 3.1, tem-se um objetivo principal G0 que se decompõe em G1 e G2 por meio de uma regra AND-Sequencial. A partir daí, outros objetivos e tarefas compõe o modelo com suas respectivas regras. A descrição desse modelo em PDDL, seguindo a proposta explicada no parágrafo anterior é apresentada no Código 3.1.

```
1 (define (problem godaProblem)
2  (:domain godaDomain)
3  (:objects
4   G0 - goal
5   G1 - goal
6   G2 - goal
7   G3 - goal
8   G4 - goal
9   T1 - task
10  T1_1 - task
11  T1_2 - task)
```

```
12   (:init
13   (notcompleted T1)
14   (notcompleted T1_1)
15   (notcompleted T1_2)
16   (notcompleted G0)
17   (notcompleted G1)
18   (notcompleted G2)
19   (notcompleted G3)
20   (notcompleted G4))
21   (:goal
22    (completed G0))
23 )
```

Código 3.1: Descrição do modelo GODA genérico em PDDL

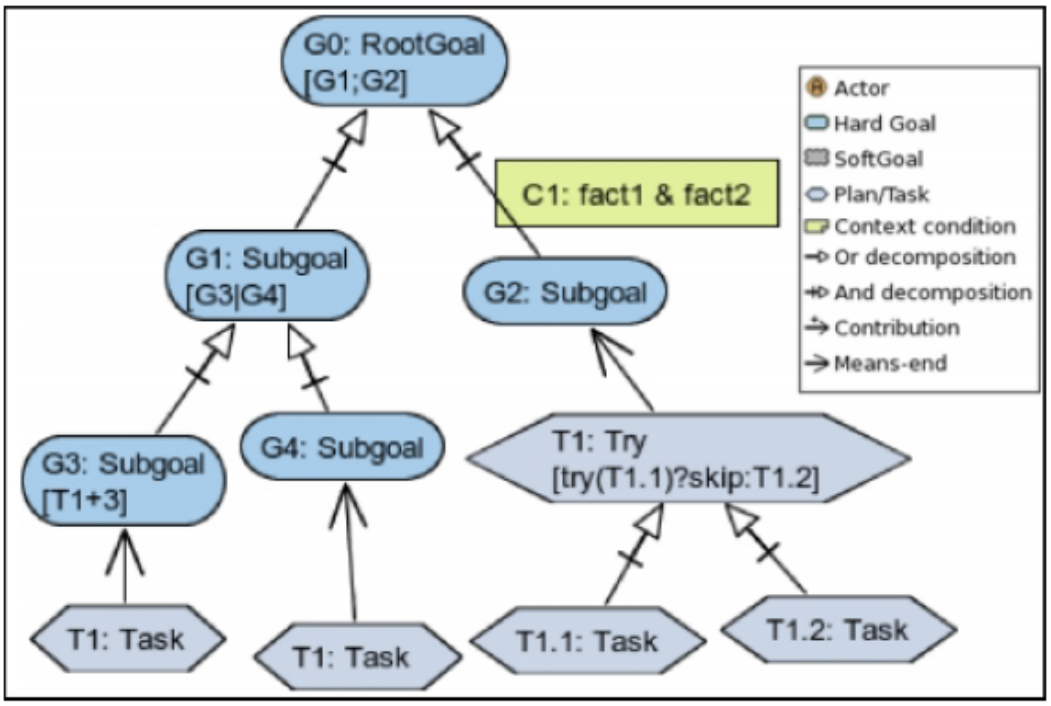

Figura 3.1: Modelo GODA de ilustração do uso dos elementos [5].

Para testar a abordagem e verificar se a definição em PDDL estava adequada, foi usado o planejador JavaGP, apresentado na Seção 2.2.5. Porém, percebeu-se que os planos escolhidos pelo JavaGP, apesar de garantirem a execução de tarefas e objetivos e de satisfazer o objetivo principal, possuía uma característica não desejável de escolher um único caminho até o objetivo, sem considerar todas as decomposições, ou seja, sem de fato

respeitar as regras do GODA. No exemplo apresentado, o planejador por vezes escolheu plano que executava as tarefas T1.1: Task, T1.2: Task, T1: Try e então prosseguia para o objetivo G2: Subgoal e dele concluia que o objetivo GO: RootGoal podia ser satisfeito, apesar de não terem sido considerados os elementos do outro lado, ou seja, G1, G3, G4 e a tarefa T1: Task. Dessa forma, a proposta, como idealizada, nunca iria atingir todos os elementos, independente do planejador.

HTN e Pyhop

Então, percebeu-se a complexidade de adequar o modelo GODA à linguagem PDDL. Sendo uma linguagem do contexto de planejamento, não foi encontrada uma forma expressiva de representar as regras de *runtime* do GODA, uma vez que em PDDL não há como forçar situações que são necessárias ao cumprimento das regras. Por esse motivo, a linguagem PDDL foi substituída pela abordagem HTN. Nela, foi possível utilizar as redes de tarefas para que todos os elementos do modelo fossem avaliados, e não apenas os elementos de um caminho direto até o objetivo principal, já que naturalmente as redes possuem uma estruturação hierárquica, que também é muito comum ao modelo GODA. Mais especificamente a composição de elementos pelo Tropos, que geralmente tem características de hierarquia, ou seja, as tarefas em níveis inferiores, logo acima os objetivos e no topo o objetivo principal.

Na sequência, foi investigado um planejador em HTN que fosse adequado à proposta. De início, foram avaliados o SGPlan e HTNPlan-P. O SGPlan, como citado no Capítulo 2.2.5, possui grande versatilidade em receber como entrada definições em HTN e ao mesmo tempo PDDL. Mas essa mesma característica foi o que gerou dificuldade à elaboração da proposta. Essa interface entre as duas linguagens acaba por congelar algumas formalizações à sintaxe PDDL que já havia sido testada e considerada inadequada para este trabalho. Além do mais, o foco do planejador é a satisfação de restrições e preferências, recursos das versões mais atuais de PDDL e que não agregam nenhuma vantagem à proposta. Então, uma tentativa foi feita com o HTNPlan-P. A premissa desse planejador é garantir maior eficiência em relação ao SGPlan e ao HPlan-P [19]. Esse planejador também é direcionado ao recurso de preferências do PDDL3 e estende esse recurso para a abordagem HTN. Novamente, a formalização utilizada não pareceu interessante ao projeto e também não foi possível experimentar com o HTNPlan-P dado que a fonte para *download* do planejador não está mais acessível.

Por fim, a outra tentativa foi realizada com o Pyhop. Como esse planejador é baseado no SHOP e a implementação tem um foco mais voltado ao HTN e a decomposição ordenada das redes de tarefas, o Pyhop se mostrou mais adequado à proposta deste trabalho. E não apenas isso, mas a versatilidade das abstrações do Pyhop que são todas

implementadas puramente em Python, trouxe um elemento de maior liberdade em poder formalizar a conversão das regras do GODA para o HTN. Portanto, ficou definido que a proposta de conversão seria do modelo GODA, estendida do modelo Tropos, para HTN, implementado na forma utilizada pelo Pyhop.

3.2 Modelo Conceitual e Implementacional

Esta seção apresenta a descrição completa da proposta de conversão e também a descrição da implementação do protótipo, bem como do seu funcionamento. Primeiro, apresenta-se a equivalência entre os elementos do GODA/Tropos e os elementos do HTN. Em seguida, uma explicação do operadores HTN, da conversão das Regras GODA e por fim a definição dos estados de inicial e de objetivo em função dos elementos Tropos

3.2.1 Conversão GODA-Pyhop

Como citado anteriormente, a proposta de conversão consiste em transformar as regras de *runtime* do GODA, os elementos do modelo Tropos e as decomposições entre elementos do Tropos em definições de estado inicial e de objetivo (através de objetos implementados no Pyhop) e métodos e operadores do HTN. A Tabela 3.1 apresenta as relações criadas no modelo de conversão.

Elementos Tropos/GODA	**Elementos HTN**
Elementos Tropos	Objetos HTN
Regras de decomposição Tropos	Métodos HTN
Regras GODA	Métodos HTN

Tabela 3.1: Equivalência entre elementos do Tropos/GODA e elementos do HTN

Operadores

Os operadores são formas de aplicar fatos sobre os objetos do problema. O Código 3.2 do arquivo *goda_operators.py* contém a descrição dos dois operadores criados, *completed* e *not-completed*. Quando aplicados a um objeto que representa um elemento Tropos adicionam o significado de que uma tarefa foi executada (ou não) ou um objetivo foi satisfeito (ou não).

```
1 def completed(state, plan):
2     if state.objects[plan] == False:
3         state.objects[plan] = True
4         return state
```

```
5       else:
6           return False
7   def not_completed(state, plan):
8       if state.objects[plan] == True:
9           state.objects[plan] = False
10          return state
11      else:
12          return False
```

Código 3.2: Trecho do arquivo de implementação dos operadores

Aqui cada função em Python implementa um dos operadores, que na verdade são bem simples. As funções verificam se o plano ou objetivo recebido como parâmetro já foi executado, se não, ele é marcado como executado (para o operador *completed*) ou se foi, marcado como não executado (para o operador *not-completed*).

Regras GODA e métodos HTN

Em seguida, foram definias as conversões para as regras GODA mostradas na Tabela **??**. Cada regra foi implementada como uma função, onde a entrada é o estado atual (como regulamentado pela implementação Pyhop discutido na Seção 2.2.5), seguido de um número de argumentos ($n \geq 1$) que representam os elementos avaliados na regra, sejam tarefas ou objetivos. A saída de cada um depende de qual decomposição será feita, porém todos retornam uma tupla com o nome do operando a ser aplicado e o elemento a quem esse operando deve ser aplicado.

Regras AND Sequencial e AND Paralela

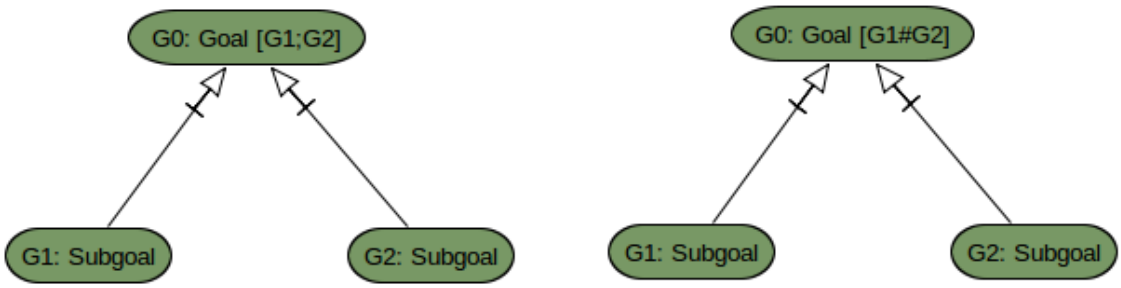

Figura 3.2: Regras de decomposição AND Sequencial e AND Paralela.

A regra AND Sequencial, conforme apresentado na Figura 3.2, satisfaz/executa n quando n_1 e n_2 são satisfeitos/executados em ordem, uma depois da outra. O método apresentado no Código 3.3 implementa essa regra.

```
1  def and_seq(state, *plans):
2   count = 0
3   for i in range(1, len(plans)):
4    plan = plans[i]
5    if state.objects[plan] == True:
6     count += 1
7   if count == len(plans)-1:
8    return [('completed', plans[0])]
9   else:
10   return [('not_completed', plans[0])]
```

Código 3.3: Função que implementa um método para a regra AND-Sequencial

Nesse método, todos os objetos recebidos como entrada são verificados. Um contador é utilizado para cada método completado. Ao final, se o contador possui o mesmo valor que a quantidade de elementos na entrada, então todos já foram completados, o que leva a uma saída positiva e a tarefa é decomposta. Caso contrário, a tarefa não é decomposta.

A regra AND Paralela, é idêntica à regra AND Sequencial. A restrição que faz com que as regras sejam iguais é discutida na Seção 3.2.3.

Regras OR Sequencial e OR Paralela

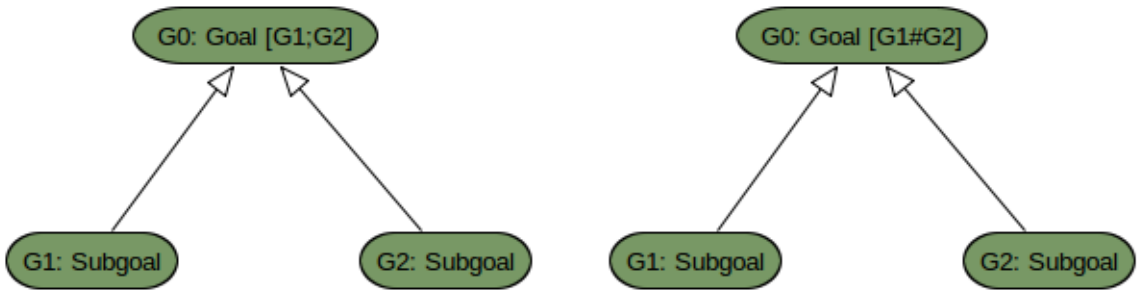

Figura 3.3: Regras de decomposição OR Sequencial e OR Paralela.

A regra OR Sequencial satisfaz/executa n quando um dos elementos, ou n_1 ou n_2 é satisfeito/executado, ou ambos, em ordem, um depois do outro. O método apresentado no Código 3.4 implementa essa regra.

```
1  def or_seq(state, *plans):
2   count = 0
3   for i in range(1, len(plans)):
4    plan = plans[i]
5    if state.objects[plan] == True:
6     count += 1
7   if count != 0:
8    return [('completed', plans[0])]
9   else:
10   return [('not_completed', plans[0])]
```

Código 3.4: Função que implementa um método para a regra OR-Sequencial

Nesse método, todos os objetos recebidos como entrada são verificados. Um contador é iterado para cada método completado. Ao final, se o contador possui valor diferente de zero, então pelo menos um elemento foi completado, o que leva a uma saída positiva e a tarefa é decomposta. Caso contrário, a tarefa não é decomposta.

A regra OR Paralela, é idêntica à regra OR Sequencial. A restrição que faz com que as regras sejam iguais é discutida na Seção 3.2.3.

Regras TRY e SKIP

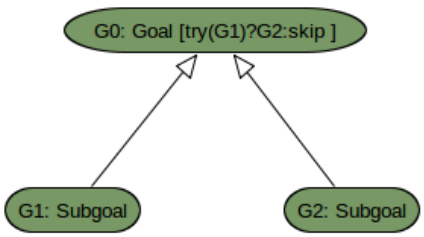

Figura 3.4: Regras de decomposição TRY e SKIP.

A regra TRY confere se n_1 foi satisfeito/executado. Em caso positivo, n_2 é satisfeito/executado, caso contrário, n_3 satisfeito/executado. Caso n_2 ou n_3 sejam *skip*, então esse elemento não é satisfeito/executado, porém n ainda assim é satisfeito/executado. O método apresentado no Código 3.5 implementa essa regra.

```
1 def try_op(state, plan, plan1, plan2, plan3):
2  if state.objects[plan1] == True:
3   if plan2 == 'skip':
4    return [('skip', plan2)]
5   else:
6    return [('completed', plan1)]
7  elif state.objects[plan] == False:
8   if plan2 == 'skip':
9    return [('skip', plan2)]
10  else:
11   return [('completed', plan2)]
12 else:
13  return [('not_completed', plan)]
```

Código 3.5: Função que implementa um método para a regra TRY e SKIP

Nesse método, o objeto *plan1* é verificado. Caso tenha sido satisfeito/executado, então é vertificado o objeto *plan2*. Se for um objeto, então *plan1* é satisfeito/executado, se ele for um *skip*, então o operador *skip* é executado e da mesma forma é avaliado o *plan3*.

Regra XOR

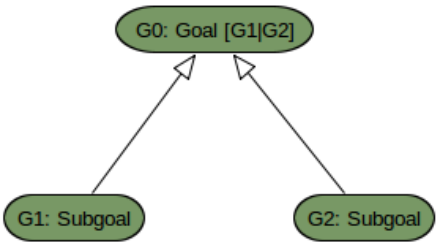

Figura 3.5: Regra de decomposição XOR.

A regra XOR satisfaz/executa n quando um dos elementos, ou n_1 ou n_2 é satisfeito/executado, mas nunca os dois. O método apresentado no Código 3.6 implementa essa regra.

```
1 def xor(state, plan, plan1, plan2):
2  if state.objects[plan1] == True:
3   if state.objects[plan2] == True:
4    return [('not_completed', plan)]
```

```
5   else
6    state.objects[plan2] = False
7    return [('completed', plan)]
8  elif state.objects[plan2] == True:
9   state.objects[plan1] = False
10  return [('completed', plan)]
11 else:
12  return [('not_completed', plan)]
```

Código 3.6: Função que implementa um método para a regra XOR

Nesse método, o objeto *plan1* é avaliado. Se foi executado/satisfeito, então *plan2* é avaliado. Se não foi executado, então a regra é decomposta e *plan* é executado/satisfeito. Caso contrário, a regra não é decomposta. Se *plan1* não foi executado/satisfeito, então *plan2* é avaliado. Em caso positivo, a regra é decomposta e *plan* é executado/satisfeito, caso contrário, a regra não é decomposta.

Regra MEANS-END

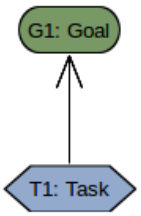

Figura 3.6: Regra de decomposição MEANS-END.

Apesar de não ser uma regra GODA, a decomposição MEANS-END também foi convertida para um método HTN, de forma a representar essa relação entre elementos. O método apresentado no Código 3.7 implementa essa regra.

```
1 def means_end(state, plan, plan1):
2  if state.objects[plan1] == True:
3   return [('completed', plan)]
4  else:
5   return [('not_completed', plan)]
```

Código 3.7: Função que implementa um método para MEANS-END

Nesse método, o objeto *plan1* é avaliado. Se foi executado/satisfeito, então *plan* também é executado/satisfeito. Caso contrário, a decomposição MEANS-END não é feita.

Definição do Problema em Pyhop

As regras GODA e os elementos presentes no modelo Tropos são representados na definição do problema. Nesse caso, todas as tarefas e objetivos são representados como variáveis pertencentes a um estado. Já as regras GODA são representadas como uma rede de tarefas a ser decomposta pelo planejador.

Para definir o estado inicial, todas as variáveis (tarefas e objetivos) são inicialmente não-completadas, salvo aquelas que não possuem decomposições e nem regras, que são inicializadas pelo usuário do protótipo. O *estado de objetivo* é na verdade uma rede de tarefas que contém todas as regras GODA utilizadas na modelagem, ordenadas de forma *bottom-up*, ou seja, as regras são aplicadas às folhas da árvore sintática primeiro (essa estrutura é explicada melhor na Seção 3.2.2) e vão sendo aplicadas aos outros nós até que chegue à regra do objetivo principal.

A implementação do problema é apresentada no Código 3.8.

```
1 state = State('Actor 1')
2 state.objects = { 'T2: Task':True, 'G3: Subgoal':False, \
3 'T3: Task':True, 'G4: Subgoal':False, 'G1: Subgoal':False, \
4 'T1.1: Task':True, 'T1.2: Task':True, 'T1: Task ':False, \
5 'G2: Subgoal':False, 'G0: RootGoal':False}
6
7 pyhop(state, [('k_times', 'G3: Subgoal', 'T2: Task', '3'), \
8 ('means_end', 'G4: Subgoal', 'T3: task'), \
9 ('xor', 'G1: Subgoal', 'G3: Subgoal', 'G4: Subgoal'), \
10 ('try_op', 'T1: Task', 'T1.1: Task', 'skip', 'T1.2: Task'), \
11 ('means_end', 'G2: Subgoal', 'T1: Task'), \
12 ('and_seq', 'G0: RootGoal', 'G1: Subgoal', 'G2: Subgoal'), \
13 ], verbose=1)
```

Código 3.8: Implementação de problema para o modelo da Figura 3.1

No Código 3.8, um objeto da class *State* é instanciado rotulado com o nome do Ator. Em seguida, os objetivos e tarefas são adicionados como atributos desse objeto, em uma estrutura de dicionário, onde cada atributo também reflete o estado de uma tarefa ou objetivo, ou seja, se eles foram executados/satisfeitos (valor *True*) ou se não (valor *False*). Loga abaixo, a declaração do estado de objetivo, representado por uma rede de tarefas,

onde cada elemento é uma tupla com o nome da regra a ser decomposta, o objeto ao qual a regra será aplicada e os elementos que fazem parte da decomposição.

Agora, com as definições de operadores, métodos e problema, o planejador pode ser executado para encontrar uma solução. Nas Seções 3.2.2 e 3.2.3, são apresentados o funcionamento do protótipo e sua utilização, bem como as funcionalidades e restrições implementadas.

3.2.2 Arquitetura e Fluxo de Execução

O protótipo foi divido em quatro módulos: módulo de *parsing*, módulo de interface (que engloba os módulos de conversão e a definição do problema) e módulo de planejamento (Pyhop). A Figura 3.7 apresenta uma ilustração em alto nível dos módulos do protótipo.

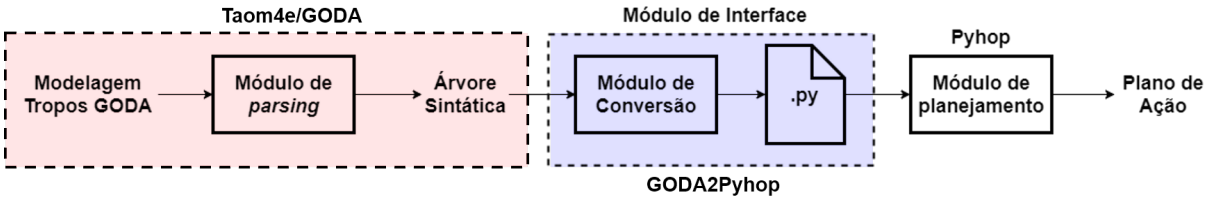

Figura 3.7: Representação em alto nível do funcionamento do protótipo.

Módulo de *Parsing*

A primeira consideração ao definir a arquitetura foi avaliar como seria feito o *parsing* dos dados do modelo gráfico dentro do Taom4e/GODA. Porém, o GODA já possui um módulo de *parsing* para o uso nas ferramentas de análise de *runtime*. Dessa forma, o protótipo utiliza a saída do módulo de *parsing*, implementado no GODA, que é uma árvore de análise sintática com todas as regras representadas como expressões para cada elemento do modelo Tropos. Na Figura 3.8 é possível ver uma ilustração dessa estrutura. Essa estrutura então é utilizada como entrada para o módulo de conversão, que percorre a árvore e converte as regras GODA e os elementos Tropos para métodos e objetos no Pyhop.

Módulo de Conversão

Esse módulo recebe a árvore de análise sintática como entrada e começa a percorrê-la buscando por todos os nós e armazenando-os em uma lista. A árvore é percorrida sempre das folhas para a raiz e em pré-ordem, ou seja, os filhos mais à esquerda de um nó são visitados primeiro. Durante essa passagem, as folhas da árvore são marcadas para depois

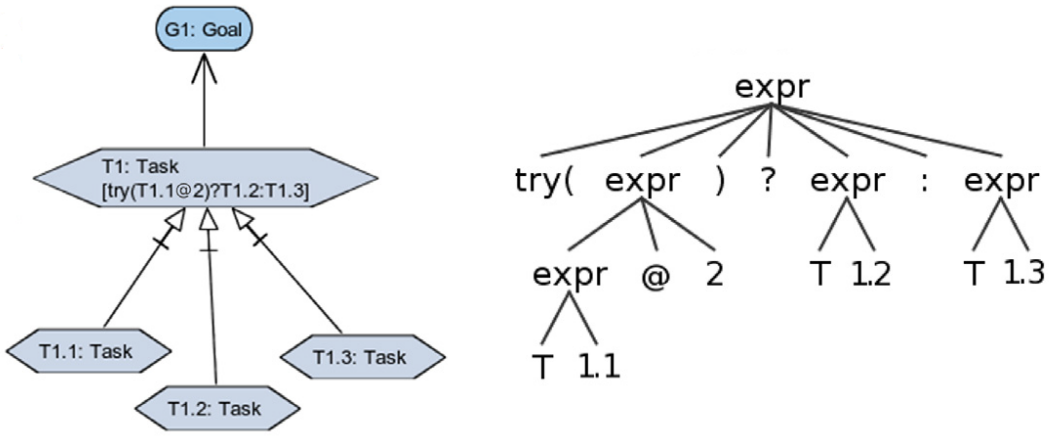

Figura 3.8: Árvore gerada a partir da decomposição de regras GODA [5].

serem utilizadas pelo módulo de interface. Terminado o preenchimento da lista de objetos, a execução continua com uma nova passagem pela árvore, mas desse vez buscando pelas regras GODA associadas a cada nó.

Assim que um nó que contém regra GODA é encontrado, uma chamada a uma função específica é realizada e essa regra é convertida em uma tupla que contém o nome do método que implementa a regra em Pyhop, o nome do elemento que contém a regra e os argumentos que são os elementos da decomposição dessa regra (como ilustrado na Seção 3.2.1). Essas tuplas são armazenadas em uma outra lista e então essas duas listas são enviadas como entrada para o módulo de interface.

Módulo de Interface

Esse módulo é responsável pela conexão entre o GODA e o PYhop. A entrada para esse módulo são as duas listas geradas pelo módulo de conversão. A lista de objetos é percorrida primeiro e todos os objetos marcados como folhas da árvore são apresentados ao usuário para que seja determinado se eles devem ser inicializados ou não, ou seja, se essas tarefas ou objetivos já serão considerados executados/satisfeitos na definição do problema em Pyhop. Os demais objetos não passam por essa verificação dado que já foram considerados como não completados.

Em seguida, é apresentada uma janela (Figura 3.11) onde o usuário pode escolher o diretório onde será escrito o arquivo do problema, as tarefas e objetivos que ele deseja marcar como completados e uma opção de abrir o arquivo gerado em um editor de texto para revisão antes de executar o Pyhop. Ao confirmar essas informações, uma nova janela é apresentada confirmando que o processo de conversão e de criação do arquivos foram

realizados e é solicitado ao usuário decidir se o Pyhop deve ser executado em seguida com o arquivo criado ou se a execução protótipo deve ser finalizada.

Com o arquivo criado, o próximo módulo entra em ação, seja automaticamente ou manualmente (dependendo da escolha do usuário). Esse módulo recebe o arquivo de problema gerado pelo módulo de interface e também os arquivos de métodos e de operandos (já presentes no diretório principal do pyhop). Assim, o planejador é acionado e apresenta o resultado na tela.

Módulo Pyhop

Esse módulo é um pouco diferente em comparação ao anteriores. Todos os outros módulos são vinculados ao GODA e implementados como uma funcionalidade de extensão do plugin. Já o módulo de planejamento está fora do GODA, apesar do diretório do Pyhop estar presente no mesmo diretório do projeto Eclipse. No diretório do Pyhop, ficam os arquivos Python necessários para a execução do planejador bem como os arquivos usados na formalização da proposta.

Basicamente o módulo consiste na especificação dos métodos, relacionados com as regras GODA e dos operadores, que modificam o objetos em cada estado. Quando executado, o módulo apresenta no terminal os dados dos arquivos de definição. Após a execução, o resultado obtido é mostrado na janela do terminal e então encerra-se a execução do protótipo.

3.2.3 Restrições

Nesta seção são apresentadas as restrições do modelo conceitual. O modelo foi baseado no planejamento clássico, como discutido na 2.2. Portanto, o sistema deve ser monoagente (apenas um agente é capaz de executar as ações) e seguir as demais premissas apresentadas. Assim, o modelo possui duas restrições em relação ao uso das regras GODA. As regras que definem objetos que são executados/satisfeitos em ordem sequencial ou paralela são todas tratadas sem levar em conta a semântica da ordem de execução. Isso é válido para as regras AND e OR. Essa restrição se deve ao fato de que no planejamento clássico, as ações são sempre executadas de forma sequencial, limitando o uso das regras AND-Paralela e OR-Paralela que são consideradas como sequenciais.

Outra restrição do modelo são as regras GODA que envolvem repetição. Novamente, no planejamento clássico, não há como representar ações que se repetem, dado que o planejador é que escolhe a ordem de execução das ações, e forçar uma repetição interfere com a função do planejador que é encontrar o melhor plano. Portanto, as regras $n{+}k$, $n\#k$ e $n@k$, apresentadas na Tabela 2.1, não foram implementadas com essa semântica.

Ao invés disso, quando o módulo Pyhop é acionado para que o usuário execute o planejador, fica a critério do próprio usuário escolher entre duas opções para o resultado dessas regras: (1) independente dos elementos presentes na regra, ela sempre será decomposta e o elemento completado; ou (2) independente dos elementos presentes na regra, ela nunca será decomposta e o elemento não-completado. Além dessas, a regra *opt(n)* também foi implementada da mesma forma, ficando a critério do usuário escolher se ela será executada ou não.

3.3 Utilização do Protótipo

Nessa seção é apresentada uma explicação do uso do protótipo, começando pela preparação do ambiente de execução, em seguida a modelagem feita no Taom4e, aplicação das regras GODA e por fim a execução do planejador.

3.3.1 Preparação do Ambiente

Os primeiros requisitos são a instalação do Java 8 (JRE versão 1.8.0[1]) e do Python 3 (versão 3.6.3[2]). Para rodar o GODA corretamente é necessário instalar o JRE 1.8 e para executar o Pyhop pode ser usado o Python 2 ou Python 3. Para os experimentos deste trabalho foi usado o Python 3.6.3. Para executar o protótipo é necessário preparar o ambiente Eclipse, instalando o plugin Taom4e e em seguida importando o projeto GODA[3]. O diretório contendo o Pyhop e os demais arquivos para executar o planejador encontra-se dentro do projeto do GODA, portanto não é preciso se preocupar com isso. Depois de preparado o ambiente, basta abrir o Eclipse e executar o projeto importado em *Run As -> Eclipse Application*.

3.3.2 Modelagem utilizando Taom4e

Com o plugin GODA-Pyhop aberto, a próxima etapa é modelar o processo usando os elementos e as composições. Primeiro deve ser incluído um ator, definido seu nome e atribuindo valor *true* à propriedade *isSystem* (essa propriedade inclui o ator na análise do *parser*). Em seguida, os outros elementos podem ser adicionados e também suas relações, utilizando a paleta de objetos na lateral esquerda. Também é preciso atribuir o nome correto aos elementos seguindo a descrição do GODA. Um passo-a-passo da utilização da ferramenta é apresentado no Anexo IX.

[1] http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

[2] https://www.python.org/downloads/release/python-363/

[3] Um passo-a-passo detalhado dessa preparação pode ser consultado em: https://github.com/lesunb/CRGMToPRISM/

3.3.3 Aplicando as regras de *runtime* do GODA

Após a modelagem Tropos, agora devem ser adicionadas as regras do GODA. Para tanto, basta selecionar um elemento e alterar seu atributo *name* colocando a regra depois do identificador. Na Figura 3.9 é apresentado um exemplo de regra AND Sequencial entre dois objetivos e o atributo com descrição e a regra especificados.

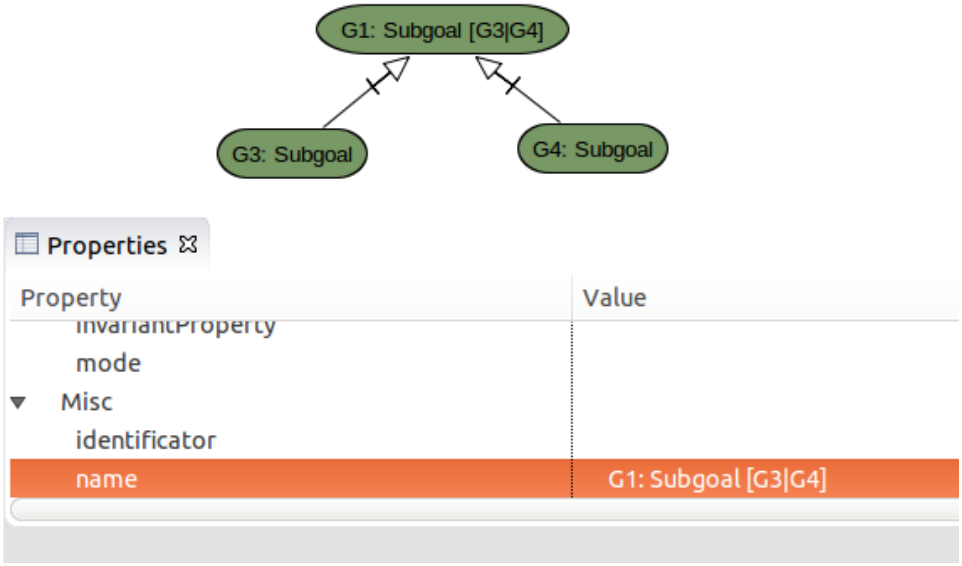

Figura 3.9: Aplicação da regra AND Sequential.

3.3.4 Execução do protótipo

Após aplicadas todas as regras, o próximo passo é executar o módulo de conversão para o Pyhop. Para tanto, basta selecionar o ator e selecionar no menu do Eclipse a opção *GODA -> Generate Pyhop*, ou então clicar no botão indicado na Figura 3.10.

Figura 3.10: Menu para execução do módulo de conversão.

Em seguida, a janela do módulo de interface é apresentada para que o usuário escolha três configurações como mostrado na Figura 3.11:

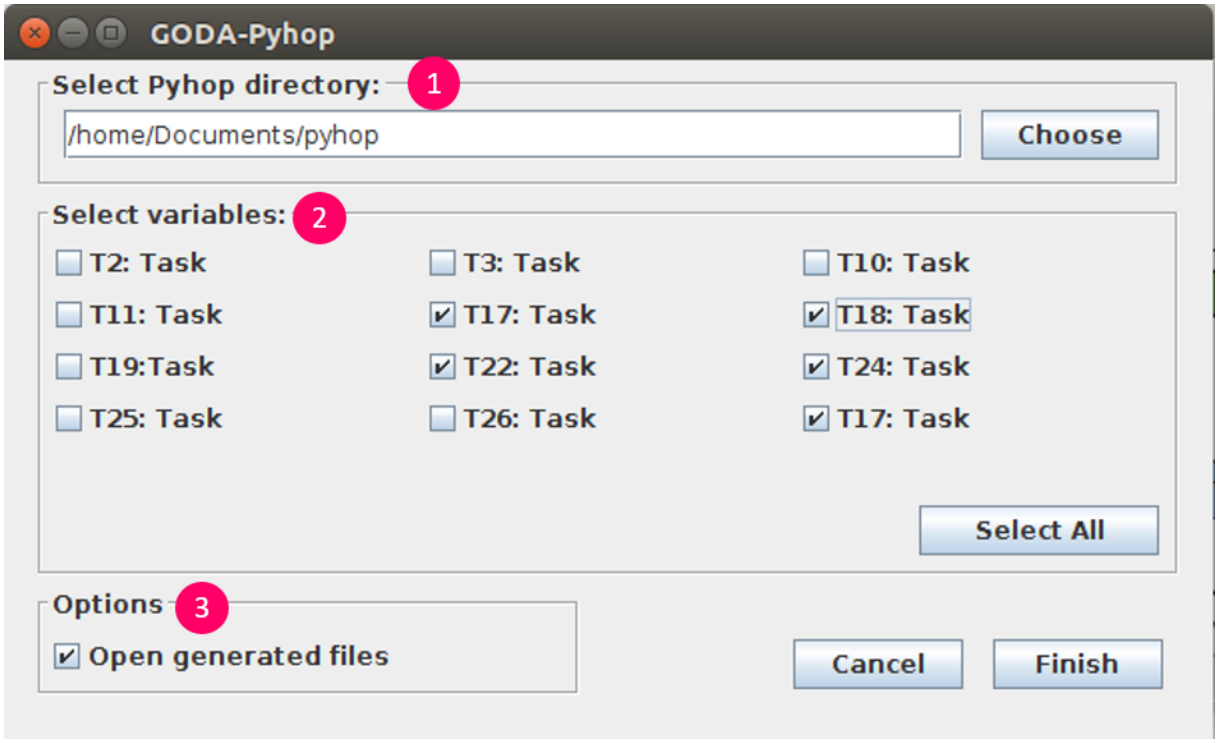

Figura 3.11: Janela de interface GODA-Pyhop.

1. Selecionar diretório: aqui é solicitado do usuário o diretório onde deve ser criado o arquivo de problema. Essa etapa é obrigatória para finalizar a execução do programa.
2. Selecionar variáveis: aqui o usuário deve marcar os elementos que ele deseja inicializar como completados.
3. Opções: aqui o usuário pode escolher se quer que o arquivo criado seja aberto após a execução para edição.

Ao terminar de selecionar todas as opções, o usuário deve clicar em *Finish*.

3.3.5 Execução do Pyhop

Após terminada a conversão, o usuário deve navegar até o diretório escolhido pelo terminal. O diretório deve ser o que contém os arquivos de definição de operadores e de métodos. Então, o usuário deve executar o comando:

```
python goda-pyhop.py
```

3.4 Apresentação da Saída

A saída do protótipo é dividida em três partes: (i) apresentação dos operadores, das tarefas e dos métodos; (ii) apresentação da definição do estado inicial; e (iii) apresentação do resultado. Na Figura 3.12, é possível perceber cada uma das partes.

Figura 3.12: Saída do Pyhop apresentada no terminal.

Aqui, para cada linha do estado inicial (onde estão as tarefas a serem decompostas (ii)), uma linha é apresentada no resultado contendo o que foi feito na decomposição dessa tarefa (iii), ou seja, se aquela tarefa ou objetivo foi executada/satisfeito.

Capítulo 4

Experimentos

Esse capítulo apresenta os experimentos realizados para validar a proposta e o protótipo. Os resultados obtidos são incluídos e discutidos. Todos os exemplos foram executados de acordo com o modelo de conversão apresentado no Seção 3.2.1 e utilizando o protótipo da forma especificada na Seção 3.3. O ambiente utilizado durante os experimentos inclui:

- Hardware
  - Sistema Operacional Ubuntu 14.04 LTS 32bits (rodando em uma máquina virtual VMWare Workstation 12)
  - Processador Intel Core i3-3217U 1.80GHz x 2 cores
  - Vídeo Intel HD Graphics 4000
  - Memória 2Gib DDR3 1600Mhz
- Software
  - Eclipse Mars 2 versão 4.5.2[1]
  - Plugin Taom4e[2] versão 0.6.3.1
  - GODA[3] extension for Taom4e
  - Pyhop[4] HTN Planner versão 1.2.2
  - Python versão 3.6.3
  - Java SE Runtime Environment versão 1.8.0

[1]http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/mars2
[2]http://selab.fbk.eu/taom/
[3]https://github.com/lesunb/CRGMToPRISM/tree/GODA_BPM
[4]https://bitbucket.org/dananau/pyhop

4.1 Ilustrações

Os primeiros experimentos foram feitos com modelos genéricos, um deles extraído de [5] e dois criados para testar a corretude da implementação.

4.1.1 Modelo CRGM Genérico

Esse experimento foi o primeiro a ser realizado. O objetivo aqui era validar inicialmente a execução do protótipo e verificar possíveis erros e *bugs*. Por se tratar de um modelo simples, mas que possui todos os elementos do TROPOS considerados, bem como as relações e quatro regras GODA diferentes, esse modelo foi escolhido para validar todos os módulos, durante o desenvolvimento. Na Figura 4.1 é apresentado o modelo pronto dentro do *plugin*, já com todos os elementos e com as regras GODA aplicadas aos objetos.

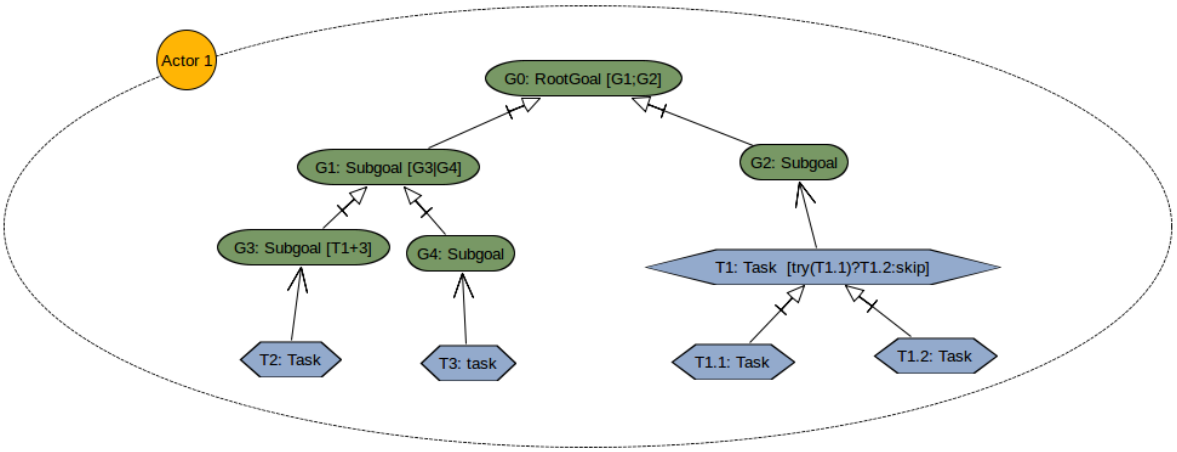

Figura 4.1: Modelagem CRGM para o exemplo do experimento.

O experimento foi realizado primeiramente modelando o processo genérico no GODA, em seguida foi executado o módulo de conversão e após o módulo de planejamento. Na etapa de conversão, foram selecionadas as tarefas *T2: Task* e *T1.1: Task* como tarefas já executadas. Essa escolha foi baseada na intenção de testar a implementação da regra *Try* em *T1: Task* e o funcionamento da regra *n+k* em *G3*. Especialmente a segunda regra mencionada, uma vez que essa é uma das regras não implementadas normalmente e que fica a cargo do usuário escolher se serão executadas ou não (como discutido na Seção 3.2.3). Em seguida, foi feita a conversão e o módulo de planejamento executado. O resultado apresentado pelo Pyhop está descrito na Figura 4.3.

Os resultados mostram o resultado esperado. O teste visava escolher tarefas que levassem à satisfação do objetivo principal e, quando selecionadas, causaram esse efeito.

```
OPERATORS: not_completed, skip, completed

TASK:          METHODS:
opt            opt
try_op         try_op
xor            xor
k_tries        k_tries
means_end      means_end
k_times        k_times
and_par        and_par
k_times_par    k_times_par
and_seq        and_seq
or_seq         or_seq
or_par         or_par
tasks =
        ('k_times', 'G3:_Subgoal', 'T2:_Task', '3')
        ('means_end', 'G4:_Subgoal', 'T3:_Task')
        ('xor', 'G1:_Subgoal', 'G3:_Subgoal', 'G4:_Subgoal')
        ('try_op', 'T1:_Task', 'T1.1:_Task', 'T1.2:_Task', 'skip')
        ('means_end', 'G2:_Subgoal', 'T1:_Task')
        ('and_seq', 'G0:_RootGoal', 'G1:_Subgoal', 'G2:_Subgoal')

result =
        ('completed', 'G3:_Subgoal')
        ('not_completed', 'G4:_Subgoal')
        ('completed', 'G1:_Subgoal')
        ('completed', 'T1.2:_Task')
        ('completed', 'T1:_Task')
        ('completed', 'G2:_Subgoal')
        ('completed', 'G0:_RootGoal')
```

Figura 4.2: Resultado apresentado pelo planejador para o primeiro teste do modelo genérico.

Em seguida, foi executado um caso onde todas as tarefas foram selecionadas como executadas pelo usuário. E o resultado é apresentado na Figura 4.3

Os resultados aqui também mostram o resultado esperado. O teste visava escolher todas as tarefas, o que poderia causar a não-satisfação do objetivo principal, dado que as decomposições anteriores envolviam uma regra XOR. No resultado apresentado na tela, podemos ver que o objetivo de fato não pôde ser alcançado com essa escolha de tarefas.

No Anexo I e no Anexo II o código gerado pelo módulo de conversão, para cada teste, pode ser consultado e avaliado para melhor entendimento do experimento.

```
OPERATORS: not_completed, skip, completed

TASK:          METHODS:
opt            opt
try_op         try_op
xor            xor
k_tries        k_tries
means_end      means_end
k_times        k_times
and_par        and_par
k_times_par    k_times_par
and_seq        and_seq
or_seq         or_seq
or_par         or_par
tasks =
        ('k_times', 'G3:_Subgoal', 'T2:_Task', '3')
        ('means_end', 'G4:_Subgoal', 'T3:_Task')
        ('xor', 'G1:_Subgoal', 'G3:_Subgoal', 'G4:_Subgoal')
        ('try_op', 'T1:_Task', 'T1.1:_Task', 'T1.2:_Task', 'skip')
        ('means_end', 'G2:_Subgoal', 'T1:_Task')
        ('and_seq', 'G0:_RootGoal', 'G1:_Subgoal', 'G2:_Subgoal')

result =
        ('completed', 'G3:_Subgoal')
        ('completed', 'G4:_Subgoal')
        ('not_completed', 'G1:_Subgoal')
        ('completed', 'T1.2:_Task')
        ('completed', 'T1:_Task')
        ('completed', 'G2:_Subgoal')
        ('not_completed', 'G0:_RootGoal')
```

Figura 4.3: Resultado apresentado pelo planejador no segundo testedo modelo genérico.

4.1.2 Modelo de Teste Largura

Os dois experimentos realizados em seguida foram pensados para forçar situações específicas, de forma a validar as funcionalidades do protótipo e sua corretude em relação à proposta. O primeiro experimento foi feito com um modelo bem "largo", onde cada elemento é decomposto em mais de dois outros elementos. Esse teste visa garantir o funcionamento do algoritmo de conversão em situações onde as regras possuem muitos operandos. Além disso, o modelo também intercala regras AND com regras OR, especialmente a regra OR do objetivo principal, diferentemente dos outros exemplos que possuem

no objetivo principal uma regra AND. Além disso, esse exemplo também possui uma regra *opt(n)*, que também se enquadra nas regras especiais discutidas na Seção 3.2.3. Essa regra foi adicionada propositalmente para continuar a validação do módulo de planejamento, onde são escolhidos os resultados dessas regras especiais. A Figura 4.4 apresenta o modelo TROPOS já com as regras GODA utilizado para esse fim.

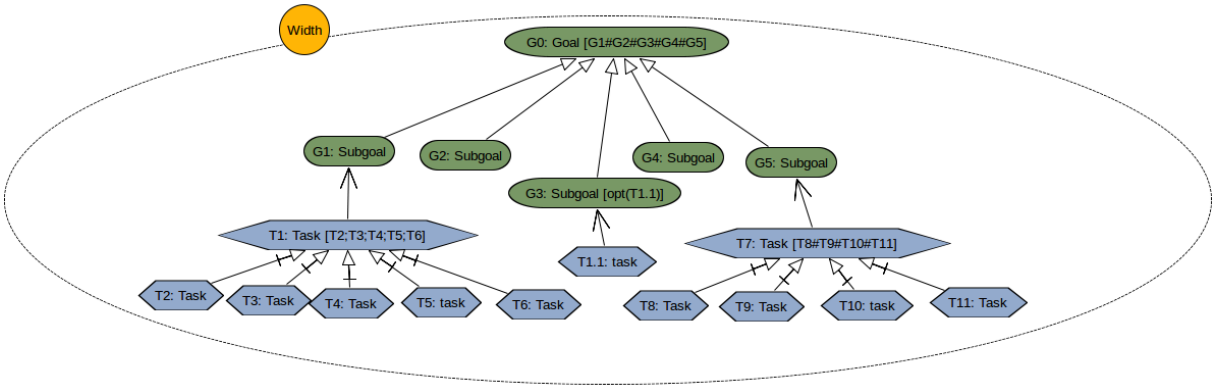

Figura 4.4: Modelagem CRGM para o exemplo do experimento.

Após modelado o exemplo, o módulo de conversão foi acionado e após a conversão, foi apresentada a tela de interface. Dessa vez, no teste realizado, só foram, inicializadas as tarefas que compõe as regras de *T1* e de *T7*, de forma que os objetivos *G2*, *G3* e *G4* não fossem satisfeitos, avaliando melhor o funcionamento da regra OR do objetivo principal. A Figura 4.5 mostra o resultado apresentado.

Aqui, pelo fato de o objetivo principal ser decomposto em uma regra OR, basta que um dos seus elementos filhos seja satisfeito. Como no teste dois desses elementos foram satisfeitos, então era esperado que o objetivo principal também fosse, como apresentado.

Para o segundo teste, foram desabilitadas todas as tarefas e todos os objetivos que foram solicitados do usuário. O intuito era fazer com que o objetivo principal não fosse atingido e, para tanto, era necessário que nenhum elemento filho do objetivo principal fosse satisfeito. A Figura 4.6 mostra o resultado apresentado pelo planejador, confirmando a situação que se desejava alcançar.

No Anexo III e no Anexo IV o código gerado pelo módulo do conversão, para cada teste, pode ser consultado e avaliado para melhor entendimento do experimento.

```
OPERATORS: not_completed, completed

TASK:         METHODS:
opt           opt
try_op        try_op
xor           xor
k_tries       k_tries
means_end     means_end
k_times       k_times
and_par       and_par
k_times_par   k_times_par
and_seq       and_seq
or_seq        or_seq
or_par        or_par

** pyhop, verbose=1: **
   state = Width

tasks =
       ('and_seq', 'T1:_Task', 'T2:_Task', 'T3:_Task', 'T4:_Task', 'T5:_task', 'T6:_Task')
       ('means_end', 'G1:_Subgoal', 'T1:_Task')
       ('opt', 'G3:_Subgoal', 'T1.1:_task')
       ('and_par', 'T7:_Task', 'T8:_Task', 'T9:_Task', 'T10:_task', 'T11:_Task')
       ('means_end', 'G5:_Subgoal', 'T7:_Task')
       ('or_par', 'G0:_RootGoal', 'G1:_Subgoal', 'G2:_Subgoal', 'G3:_Subgoal', 'G4:_Subgoal', 'G5:_Subgoal')

result =
       ('completed', 'T1:_Task')
       ('completed', 'G1:_Subgoal')
       ('completed', 'T1.1:_Task')
       ('completed', 'G3:_Subgoal')
       ('completed', 'T7:_Task')
       ('completed', 'G5:_Subgoal')
       ('completed', 'G0:_RootGoal')
```

Figura 4.5: Resultado apresentado pelo planejador no primeiro teste do modelo Largura.

```
OPERATORS: not_completed, skip, completed

TASK:         METHODS:
opt           opt
try_op        try_op
xor           xor
k_tries       k_tries
means_end     means_end
k_times       k_times
and_par       and_par
k_times_par   k_times_par
and_seq       and_seq
or_seq        or_seq
or_par        or_par
tasks =
       ('and_seq', 'T1:_Task', 'T2:_Task', 'T3:_Task', 'T4:_Task', 'T5:_task', 'T6:_Task')
       ('means_end', 'G1:_Subgoal', 'T1:_Task')
       ('opt', 'G3:_Subgoal', 'T1.1:_task')
       ('and_par', 'T7:_Task', 'T8:_Task', 'T9:_Task', 'T10:_task', 'T11:_Task')
       ('means_end', 'G5:_Subgoal', 'T7:_Task')
       ('or_par', 'G0:_Goal', 'G1:_Subgoal', 'G2:_Subgoal', 'G3:_Subgoal', 'G4:_Subgoal', 'G5:_Subgoal')

result =
       ('not_completed', 'T1:_Task')
       ('not_completed', 'G1:_Subgoal')
       ('not_completed', 'G3:_Subgoal')
       ('not_completed', 'T7:_Task')
       ('not_completed', 'G5:_Subgoal')
       ('not_completed', 'G0:_Goal')
```

Figura 4.6: Resultado apresentado pelo planejador no segundo teste do modelo Largura.

4.1.3 Modelo de Teste Altura

Semelhante ao experimento anterior, esse também visava a validação do algoritmo de conversão, mas em uma situação diferente. Aqui o objetivo era avaliar se a ordem de percorrimento da árvore definida na proposta havia sido implementada corretamente, e por conseguinte, a escrita correta no arquivo de definição do problema. Esse modelo também inclui outra situação distinta, onde uma tarefa é utilizada em duas regras diferentes, nesse caso, a tarefa *T17* pertence tanto à decomposição de *T14* como de *T13*. Além disso, esse modelo novamente utiliza regras especiais que são definidas pelo usuário e são elas $n + k$, $n@k$ e $n\#k$ (essa última regra é representada no modelo por uma restrição de implementação do GODA, uma vez que o símbolo # é utilizado para outras regras, a separação feita pelo módulo de *parsing* substitui esse símbolo por %). A Figura 4.7 mostra o que foi modelado.

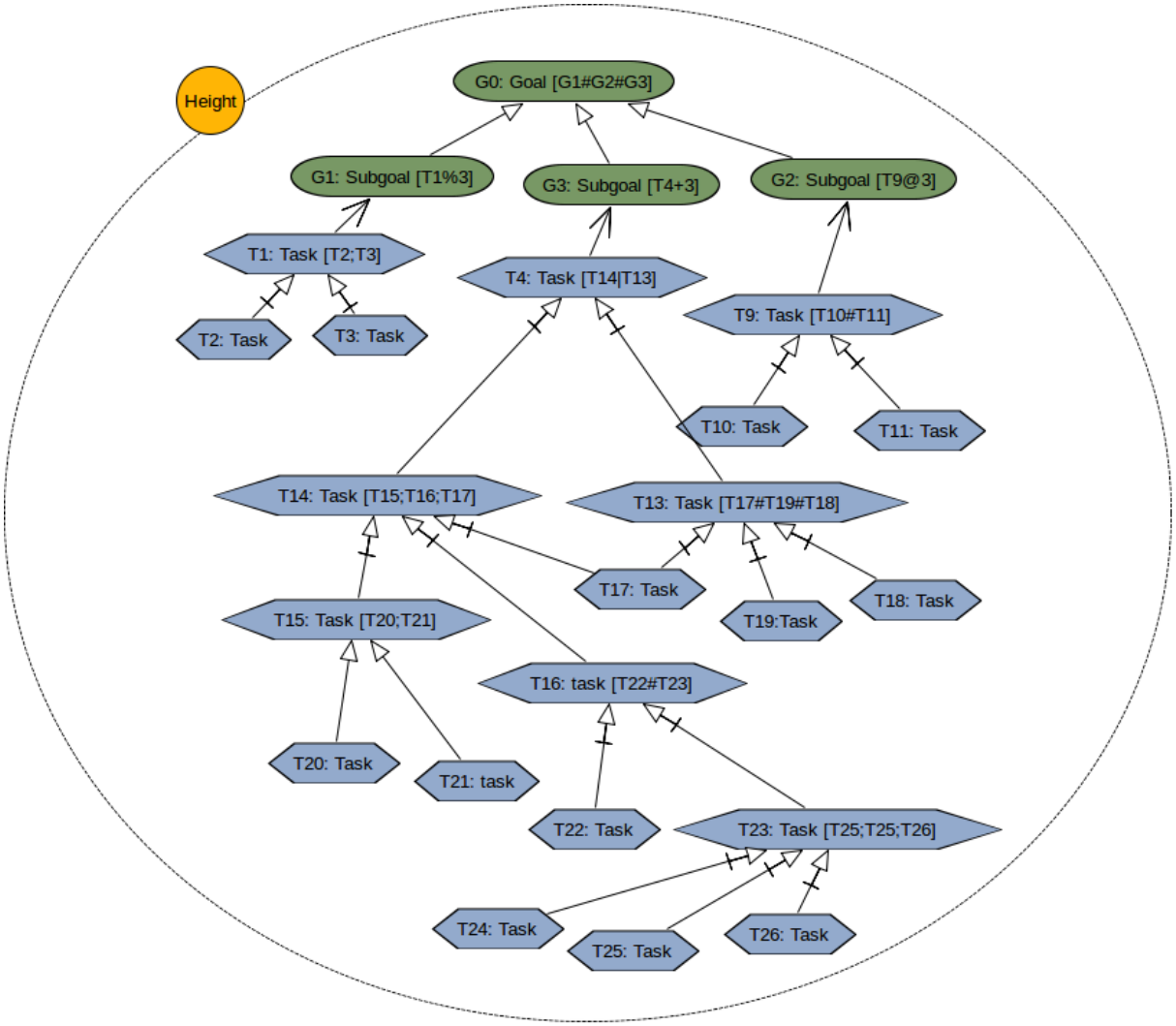

Figura 4.7: Modelagem CRGM para o exemplo do experimento.

Já na fase de conversão, todas as tarefas e objetivos sem decomposições foram escolhidos como executados. Isso se deu primeiro para validar a funcionalidade *Select All* da interface e também para verificar o comportamento do planejador quando essa situação excepcional ocorre. Na Figura 4.8 é apresentada a saída do planejador.

No Anexo V, o código gerado pelo módulo de conversão pode ser consultado e avaliado para melhor entendimento do experimento.

```
OPERATORS: not_completed, completed

TASK:          METHODS:
opt            opt
try_op         try_op
xor            xor
k_tries        k_tries
means_end      means_end
k_times        k_times
and_par        and_par
k_times_par    k_times_par
and_seq        and_seq
or_seq         or_seq
or_par         or_par
tasks =
        ('and_seq', 'T1:_Task', 'T2:_Task', 'T3:_Task')
        ('k_times_par', 'G1:_Subgoal', 'T1:_Task', '3')
        ('and_par', 'T9:_Task', 'T10:_Task', 'T11:_Task')
        ('k_tries', 'G2:_Subgoal', 'T9:_Task', '3')
        ('and_par', 'T13:_Task', 'T17:_Task', 'T18:_Task', 'T19:Task')
        ('and_seq', 'T23:_Task', 'T24:_Task', 'T25:_Task', 'T26:_Task')
        ('and_par', 'T16:_task', 'T22:_Task', 'T23:_Task')
        ('and_seq', 'T14:_Task', 'T15:_Task', 'T16:_task', 'T17:_Task')
        ('xor', 'T4:_Task', 'T13:_Task', 'T14:_Task')
        ('k_times', 'G3:_Subgoal', 'T4:_Task', '3')
        ('or_par', 'G0:_Goal', 'G1:_Subgoal', 'G2:_Subgoal', 'G3:_Subgoal')

result =
        ('completed', 'T1:_Task')
        ('completed', 'G1:_Subgoal')
        ('completed', 'T9:_Task')
        ('completed', 'G2:_Subgoal')
        ('completed', 'T13:_Task')
        ('completed', 'T23:_Task')
        ('completed', 'T16:_Task')
        ('completed', 'T14:_Task')
        ('not_completed', 'T4:_Task')
        ('not_completed', 'G3:_Subgoal')
        ('completed', 'G0:_RootGoal')
```

Figura 4.8: Resultado apresentado pelo planejador para o teste do modelo Altura.

4.2 Caso de Uso Mobee Mobile

Após os experimentos iniciais, foi decidido o uso de um modelo real, para dar mais credibilidade à proposta. O modelo escolhido foi o utilizado pelos autores em [5] e trata-se de um sistema de mobile voltado à transporte público. O Mobee é baseado na premissa

de oferecer informações antecipadas ou em tempo real sobre formas de chegar ao destino utilizando serviços de transporte público. Os usuários podem tanto receber como fornecer essas informações que consistem em rotas e paradas de ônibus. Além disso, os usuários também podem solicitar alterações nas rotas e na paradas através da aplicação.

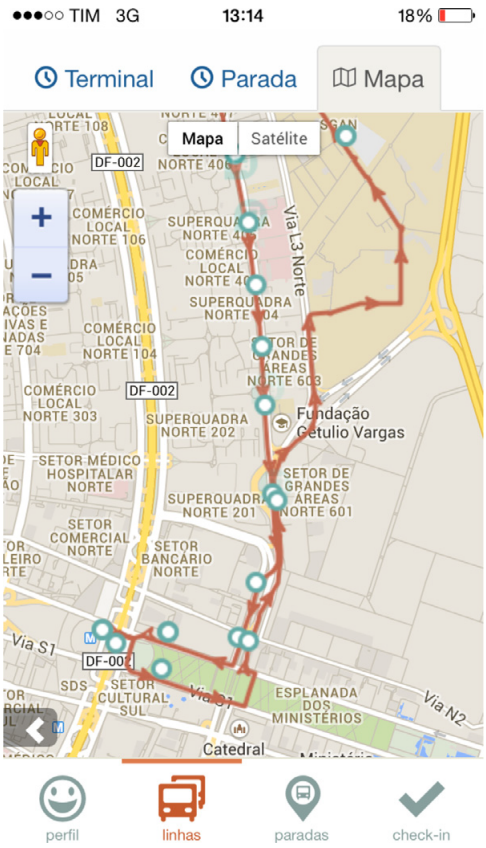

Figura 4.9: GUI Mobee - exemplo [5].

A plataforma foi desenvolvida primeiramente para Web, em 2013, e em seguida para sistemas Android e iOS. Como toda aplicação mobile, o Mobee também depende de contextos relacionados a esse tipo de domínio, por exemplo uso de bateria e disponibilidade de GPS, que variam de acordo com tempo e espaço, ou seja, durante a utilização, pode ser que o usuário entre em uma região onde o sinal de GPS não existe ou ainda que depois de usar a aplicação por muito, a bateria do dispositivo acabe. Por exemplo, na Figura 4.9, a interface do aplicativo é apresentada durante execução, e é possível notar que a bateria do dispositivo está acabando (18%), ou seja, a execução da aplicação pode ser afetada em algum momento por influência desse contexto.

Para a realização do experimento, as mesmas etapas dos experimentos anteriores foram seguidas e, dessa vez, o objetivo era avaliar a ferramenta em um caso de uso real. O modelo

utilizado foi o mesmo apresentado por [5], utilizando as mesmas regras GODA e os mesmos elementos do Tropos. A Figura 4.10 apresenta esse diagrama.

Em seguida, foi executado o módulo de conversão, para seleção das tarefas e objetivos a serem inicializados. Nesse experimento, todos os elementos foram inicializados como executados/satisfeitos, de forma a avaliar se o planejador seria capaz de executar todas as redes de tarefas. Para esse primeiro teste, o resultado apresentado é mostrado na Figura 4.11.

É possível notar que o planejador retornou o resultado esperado. As decomposições que levam ao objetivo principal são todas do tipo AND e dependentes das tarefas escolhidas pelo usuário. O ponto crítico para esse processo, porém, é a tarefa *T3.1: Fetch Geolocation*. Essa tarefa possui uma regra do tipo XOR e, no caso de teste, ambas as tarefas dessa decomposição foram executadas, o que impede que *T3.1: Fetch Geolocation* seja executada. Esse fato é adequado uma vez que *T3.11: Fetch GPS* e *T3.12: Fetch Triagulation* são tarefas que causam o mesmo efeito no ambiente e servem ao mesmo propósito: oferecer informações de posicionamento global. Assim, se ambas forem executadas, é possível que ocorra uma incoerência dos dados recebidos pelas próximas tarefas e, para o bom funcionamento do aplicativo, é importante que o posicionamento seja correto.

Então, um segundo teste foi executado para avaliar uma situação onde não há incoerência entre as informações de geolocalização, ou seja, onde a tarefa *T3.1: Fetch Geolocation* seja executada corretamente. Para esse teste, todas as tarefas foram marcadas como executadas, com exceção de *T3.12: Fetch Triagulation*. O resultado é apresentado na Figura 4.12.

Agora, com o resultado positivo da tarefa *T3.1: Fetch Geolocation*, então todo o processo pôde ser executado e o objetivo principal satisfeito, como era esperado.

No Anexo VI e no Anexo VII, os códigos dos testes gerados pelo módulo de conversão podem ser consultados e avaliados para melhor entendimento do experimento.

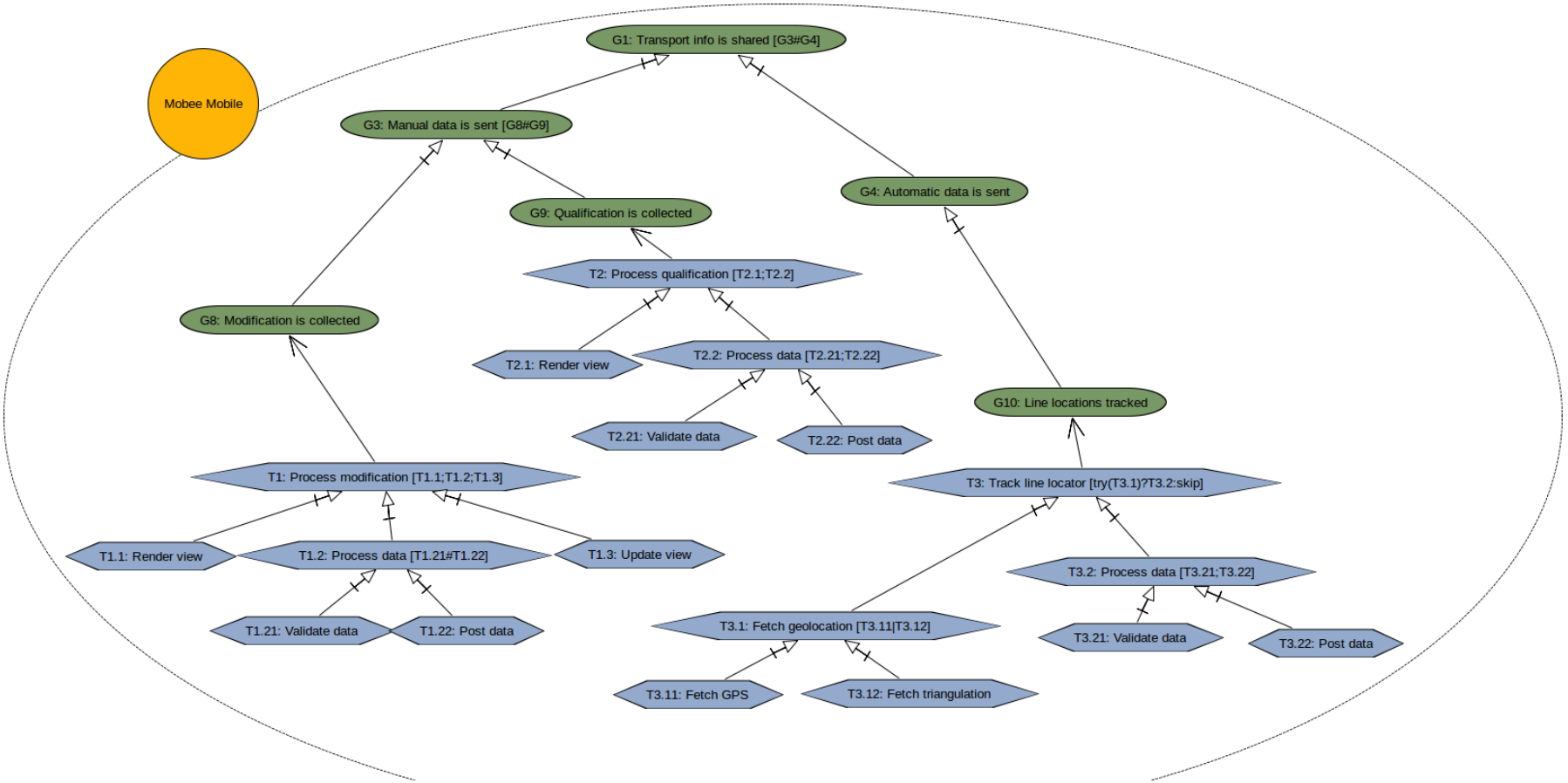

Figura 4.10: Modelagem CRGM para o exemplo do experimento

```
OPERATORS: not_completed, skip, completed

TASK:          METHODS:
opt            opt
try_op         try_op
xor            xor
k_tries        k_tries
means_end      means_end
k_times        k_times
and_par        and_par
k_times_par    k_times_par
and_seq        and_seq
or_seq         or_seq
or_par         or_par
tasks =
        ('and_par', 'T1.2:_Process_data', 'T1.21:_Validate_data', 'T1.22:_Post_data')
        ('and_seq', 'T1:_Process_modification', 'T1.1:_Render_view', 'T1.2:_Process_data', 'T1.3:_Update_view')
        ('means_end', 'G8:_Modification_is_collected', 'T1:_Process_modification')
        ('and_seq', 'T2.2:_Process_data', 'T2.21:_Validate_data', 'T2.22:_Post_data')
        ('and_seq', 'T2:_Process_qualification', 'T2.1:_Render_view', 'T2.2:_Process_data')
        ('means_end', 'G9:_Qualification_is_collected', 'T2:_Process_qualification')
        ('and_par', 'G3:_Manual_data_is_sent', 'G8:_Modification_is_collected', 'G9:_Qualification_is_collected')
        ('xor', 'T3.1:_Fetch_geolocation', 'T3.11:_Fetch_GPS', 'T3.12:_Fetch_triangulation')
        ('and_seq', 'T3.2:_Process_data', 'T3.21:_Validate_data', 'T3.22:_Post_data')
        ('try_op', 'T3:_Track_line_locator', 'T3.1:_Fetch_geolocation', 'T3.2:_Process_data', 'skip')
        ('means_end', 'G10:_Line_locations_tracked', 'T3:_Track_line_locator')
        ('means_end', 'G4:_Automatic_data_is_sent', 'G10:_Line_locations_tracked')
        ('and_par', 'G1:_Transport_info_is_shared', 'G3:_Manual_data_is_sent', 'G4:_Automatic_data_is_sent')

result =
        ('completed', 'T1.2:_Process_data')
        ('completed', 'T1:_Process_modification')
        ('completed', 'G8:_Modification_is_collected')
        ('completed', 'T2.2:_Process_data')
        ('completed', 'T2:_Process_qualification')
        ('completed', 'G9:_Qualification_is_collected')
        ('completed', 'G3:_Manual_data_is_sent')
        ('not_completed', 'T3.1:_Fetch_geolocation')
        ('completed', 'T3.2:_Process_data')
        skip  =>  ('not_completed', 'T3:_Track_line_locator')
        ('not_completed', 'G10:_Line_locations_tracked')
        ('not_completed', 'G4:_Automatic_data_is_sent')
        ('not_completed', 'G1:_Transport_info_is_shared')
```

Figura 4.11: Resultado apresentado pelo para o primeiro teste do modelo *Mobee*.

```
OPERATORS: not_completed, completed

TASK:         METHODS:
opt           opt
try_op        try_op
xor           xor
k_tries       k_tries
means_end     means_end
k_times       k_times
and_par       and_par
k_times_par   k_times_par
and_seq       and_seq
or_seq        or_seq
or_par        or_par
tasks =
        ('and_par', 'T1.2:_Process_data', 'T1.21:_Validate_data', 'T1.22:_Post_data')
        ('and_seq', 'T1:_Process_modification', 'T1.1:_Render_view', 'T1.2:_Process_data', 'T1.3:_Update_view')
        ('means_end', 'G8:_Modification_is_collected', 'T1:_Process_modification')
        ('and_seq', 'T2.2:_Process_data', 'T2.21:_Validate_data', 'T2.22:_Post_data')
        ('and_seq', 'T2:_Process_qualification', 'T2.1:_Render_view', 'T2.2:_Process_data')
        ('means_end', 'G9:_Qualification_is_collected', 'T2:_Process_qualification')
        ('and_par', 'G3:_Manual_data_is_sent', 'G8:_Modification_is_collected', 'G9:_Qualification_is_collected')
        ('xor', 'T3.1:_Fetch_geolocation', 'T3.11:_Fetch_GPS', 'T3.12:_Fetch_triangulation')
        ('and_seq', 'T3.2:_Process_data', 'T3.21:_Validate_data', 'T3.22:_Post_data')
        ('try_op', 'T3:_Track_line_locator', 'T3.1:_Fetch_geolocation', 'T3.2', 'T3.2:_Process_data')
        ('means_end', 'G10:_Line_locations_tracked', 'T3:_Track_line_locator')
        ('means_end', 'G4:_Automatic_data_is_sent', 'G10:_Line_locations_tracked')
        ('and_par', 'G1:_Transport_info_is_shared', 'G3:_Manual_data_is_sent', 'G4:_Automatic_data_is_sent')

result =
        ('completed', 'T1.2:_Process_data')
        ('completed', 'T1:_Process_modification')
        ('completed', 'G8:_Modification_is_collected')
        ('completed', 'T2.2:_Process_data')
        ('completed', 'T2:_Process_qualification')
        ('completed', 'G9:_Qualification_is_collected')
        ('completed', 'G3:_Manual_data_is_sent')
        ('completed', 'T3.1:_Fetch_geolocation')
        ('completed', 'T3.2:_Process_data')
        ('completed', 'T3:_Track_line_locator')
        ('completed', 'G10:_Line_locations_tracked')
        ('completed', 'G4:_Automatic_data_is_sent')
        ('completed', 'G1:_Transport_info_is_shared')
```

Figura 4.12: Resultado apresentado pelo para o segundo teste do modelo *Mobee*.

Capítulo 5

Conclusões

Como discutido no Capítulo 1.1, a proposta de modelo de conversão apresentada nesse trabalho é de grande proveito para a área, dado que oferece uma alternativa à análise de requisitos e dependabilidade de objetivos, durante a etapa de especificação de um produto de software. Ademais, os conceitos estudados e a implementação do protótipo trouxeram conhecimentos sobre outras áreas, como tradução de software, desenvolvimento de interfaces gráficas e integração entre projetos de diferentes linguagens de programação. Conhecimentos esses que serão de grande valia para a continuidade do projeto e o desenvolvimento de trabalhos futuros.

As dificuldades encontradas durante a pesquisa foram superadas graças à facilidade de uso tanto do framework GODA como do planejador Pyhop e, especialmente, pela característica de extensão que o planejador possui, tornando o trabalho de formalização do modelo e de seus componentes mais simples quando comparado a outros planejadores HTN.

O protótipo desenvolvido apresentou resultados bons e condizentes com o esperado do modelo. Dessa forma, apesar de ser uma ferramenta de apoio, o protótipo pode ser utilizado em um amplo espectro de problemas, desde que observadas as restrições do modelo. Além disso, a implementação foi toda feita em Java e Python, duas linguagens de programação amplamente utilizadas e facilmente portáveis para integração em outros sistemas.

Todavia, como discutido na Seção 3.2.3, o modelo ainda não abrange todas os elementos presentes no GODA, característica que será investigada posteriormente. Além disso, outros tópicos abordados nesse trabalho, porém não contemplados, ficam como possíveis trabalhos futuros e são eles:

- Integrar o planejador Pyhop dentro do plugin do GODA, dado que o código é compacto, livre e existem pacotes em Java que suportam a execução de programas codificados em outra linguagens, como Python.

- Aplicar o conceito de contextos, que é pertinente à modelagem CRGM.
- Implementar o uso de regras de repetição, criando uma equivalência entres as regras no GODA e os métodos e operadores em HTN.
- Incluir suporte ao uso de algoritmos de planejamento além daquele implementado pelo Pyhop.
- Adicionar suporte à exportação dos resultados na formalização proposta pela linguagem PDDL.

Referências

[1] Russel, Stuart e Peter Norvig: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3ª edição, 2010. ix, 10, 11, 12, 13, 15, 21, 22, 23, 24

[2] Kutluhan Erol, James Hendler, Dana S. Nau: *Semantics for hierarchical task-network planning*. Relatório Técnico, Institute for Systems Research, University of Maryland, College Park, MD20742, março 1995. ix, 19, 25

[3] Kutluhan Erol, James Hendler, Dana S. Nau: *Htn planning: Complexity and expressivity**. AAAI-94 Proceedings, 1994. ix, 25, 26

[4] Santos, Genaína Nunes Rodrigues Yan Victor dos: *Análise em dependabilidade orientada a objetivos para variabilidade e automação em bpmn*. Em *Proc. 23o Congresso de Iniciacao Cientifica da UnB e 14o Congresso de Iniciacao Cientifica do DF*. Universidade de Brasilia (UnB), 2017. ix, 28, 29

[5] Danilo Filgueira Mendonça, Genaína Nunes Rodrigues, Raian Ali Vander Alves Luciano Baresi: *Goda: A goal-oriented requirements engineering framework for runtime dependability analysis*. Information and Software Technology, 80:245–264, 2016. ix, x, xi, 8, 9, 33, 43, 50, 57, 58, 59

[6] Malik Ghallab, Dana Nau, Paolo Traverso: *Automated Planning: Theory & Practice*. Cambridge University Press, 2016. 1, 9

[7] Pierre Bourque, Richard E. (Dick) Farley: *SWEBOK - Guide to the Software Engineering Body of Knowledge V3.0*. IEEE Computer Society, 2004. 4

[8] Sommerville, Ian: *Engenharia de Software*. Pearson, 2007. 4

[9] Raian Ali, Fabiano Dalpiaz, Paolo Giorgini: *A goal-based framework for contextual requirements modeling and analysis*. Requirements Eng, 15:439–458, 2010. 5, 8

[10] Eric Yu, John Mylopoulos: *Why goal-oriented requirements engineering*. Proceedings of REFSQ'98, páginas 15–22. 5

[11] Zave, Pamela: *Classification of research efforts in requirements engineering*. ACM Computing Surveys, 29(4):315–321, 1997. 5

[12] Yu, Eric S.: *Social modeling and i**. Conceptual Modeling: Foundantions and Applications - Essays in Honor of John Mylopoulos, 5600, 2009. 6

[13] Fabiano Dalpiaz, Alexander Borgida, Jennifer Horkoff John Mylopoulos: *Runtime goal models: Keynote*. Proceedings - International Conference on Research Challenges in Information Science, páginas 1–11, maio 2013. 8

[14] Fikes, Richard E. e Nils J. Nilsson: *Strips: A new approach to the application of theorem proving to problem solving*. Artificial Intelligence, 2(3):189–208, 1971, ISSN 0004-3702. `http://www.sciencedirect.com/science/article/pii/0004370271900105`. 14

[15] Meneguzzi, Felipe: *Graphplan implementations*, 2010. `http://emplan.sourceforge.net/`. 15, 16, 18

[16] Alfonso E. Gerevini, Patrik Haslum, Derek Long Alessandro Saetti Yannis Dimopoulos: *Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners*. Artificial Intelligence, 173:619–668, 2009. 17

[17] Malik Ghallab, Adele Howe, Craig Knoblock Drew McDermott Ashwin Ram Manuela Veloso Daniel Weld David Wilkins: *Pddl - the planning domain denition language version 1.2*. AIPS-98, 1998. 17

[18] *Pyhop*. 20, 21

[19] Sohrabi S, Baier JA, McIlraith S: *Htn planning with preferences*. Proceedings of the 21st international joint conference on artificial intelligence (IJCAI'09), páginas 1790–1797, 2009. 28, 34

[20] Nunes, Vanessa Tavares: *Dynamic Process Adaptation: Planning in a Context-Aware Approach*. Tese de Doutoramento, UFRJ/COPPE, Rio de Janeiro, 2014. 29

Anexo I

Código para o Experimento do Modelo Genérico

```
from __future__ import print_function
from pyhop import *

import goda_operators
import goda_methods

print('')
print_operators()

print('')
print_methods()

state = State('Actor 1')
state.objects = {\
'T2:_Task':True,\
'G3:_Subgoal':False,\
'T3:_task':False,\
'G4:_Subgoal':False,\
'G1:_Subgoal':False,\
'T1.1:_Task':True,\
'T1.2:_Task':False,\
'T1:_Task_':False,\
'G2:_Subgoal':False,\
'G0: RootGoal':False}
```

```
25
26 pyhop(state, [\
27 ('k_times', 'G3:_Subgoal', 'T2:_Task', '3'),\
28 ('means_end', 'G4:_Subgoal', 'T3:_task'),\
29 ('xor', 'G1:_Subgoal', 'G3:_Subgoal', 'G4:_Subgoal'),\
30 ('try_op', 'T1:_Task_', 'T1.1:_Task', 'T1.2', 'T1.2:_Task'),\
31 ('means_end', 'G2:_Subgoal', 'T1:_Task_'),\
32 ('and_seq', 'G0:_RootGoal', 'G1:_Subgoal', 'G2:_Subgoal'),\
33 ], verbose=1)
```

Anexo II

Código para o 1º Experimento do Modelo *Width*

```
from __future__ import print_function
from pyhop import *

import goda_operators
import goda_methods

print('')
print_operators()

print('')
print_methods()

state = State('Actor 1')
state.objects = {\
'T2:_Task':True,\
'G3:_Subgoal':False,\
'T3:_task':True,\
'G4:_Subgoal':False,\
'G1:_Subgoal':False,\
'T1.1:_Task':True,\
'T1.2:_Task':True,\
'T1:_Task_':False,\
'G2:_Subgoal':False,\
'G0: RootGoal':False}
```

```
25
26 pyhop(state, [\
27 ('k_times', 'G3:_Subgoal', 'T2:_Task', '3'),\
28 ('means_end', 'G4:_Subgoal', 'T3:_task'),\
29 ('xor', 'G1:_Subgoal', 'G3:_Subgoal', 'G4:_Subgoal'),\
30 ('try_op', 'T1:_Task_', 'T1.1:_Task', 'T1.2', 'T1.2:_Task'),\
31 ('means_end', 'G2:_Subgoal', 'T1:_Task_'),\
32 ('and_seq', 'G0:_RootGoal', 'G1:_Subgoal', 'G2:_Subgoal'),\
33 ], verbose=1)
```

Anexo III

Código para o 2º Experimento do Modelo *Width*

```
from __future__ import print_function
from pyhop import *

import goda_operators
import goda_methods

print('')
print_operators()

print('')
print_methods()

state = State('Width')
state.objects = {\
'T2:_Task':True,\
'T3:_Task':True,\
'T4:_Task':True,\
'T5:_task':True,\
'T6:_Task':True,\
'T1:_Task':False,\
'G1:_Subgoal':False,\
'G2:_Subgoal':False,\
'T1.1:_task':False,\
'G3:_Subgoal':False,\
```

```
'G4:_Subgoal':False,\
'T8:_Task':True,\
'T9:_Task':True,\
'T10:_task':True,\
'T11:_Task':True,\
'T7:_Task':False,\
'G5:_Subgoal':False,\
'G0: Goal':False}

pyhop(state, [\
('and_seq', 'T1:_Task', 'T2:_Task', 'T3:_Task', 'T4:_Task', '
   T5:_task', 'T6:_Task'),\
('means_end', 'G1:_Subgoal', 'T1:_Task'),\
('opt', 'G3:_Subgoal', 'T1.1:_task'),\
('and_par', 'T7:_Task', 'T8:_Task', 'T9:_Task', 'T10:_task',
   'T11:_Task'),\
('means_end', 'G5:_Subgoal', 'T7:_Task'),\
('or_par', 'G0:_Goal', 'G1:_Subgoal', 'G2:_Subgoal', 'G3:
   _Subgoal', 'G4:_Subgoal', 'G5:_Subgoal'),\
], verbose=1)
```

Anexo IV

Código para o 1º Experimento do Modelo *Height*

```
from __future__ import print_function
from pyhop import *

import goda_operators
import goda_methods

print('')
print_operators()

print('')
print_methods()

state = State('Width')
state.objects = {\
'T2:_Task':False,\
'T3:_Task':False,\
'T4:_Task':False,\
'T5:_task':False,\
'T6:_Task':False,\
'T1:_Task':False,\
'G1:_Subgoal':False,\
'G2:_Subgoal':False,\
'T1.1:_task':False,\
'G3:_Subgoal':False,\
```

```
'G4:_Subgoal':False,\
'T8:_Task':False,\
'T9:_Task':False,\
'T10:_task':False,\
'T11:_Task':False,\
'T7:_Task':False,\
'G5:_Subgoal':False,\
'G0: Goal':False}

pyhop(state, [\
('and_seq', 'T1:_Task', 'T2:_Task', 'T3:_Task', 'T4:_Task', '
   T5:_task', 'T6:_Task'),\
('means_end', 'G1:_Subgoal', 'T1:_Task'),\
('opt', 'G3:_Subgoal', 'T1.1:_task'),\
('and_par', 'T7:_Task', 'T8:_Task', 'T9:_Task', 'T10:_task',
   'T11:_Task'),\
('means_end', 'G5:_Subgoal', 'T7:_Task'),\
('or_par', 'G0:_Goal', 'G1:_Subgoal', 'G2:_Subgoal', 'G3:
   _Subgoal', 'G4:_Subgoal', 'G5:_Subgoal'),\
], verbose=1)
```

Anexo V

Código para o 2º Experimento do Modelo *Height*

```
from __future__ import print_function
from pyhop import *

import goda_operators
import goda_methods

print('')
print_operators()

print('')
print_methods()

state = State('Height')
state.objects = {\
'T2:_Task':True,\
'T3:_Task':True,\
'T1:_Task':False,\
'G1:_Subgoal':False,\
'T10:_Task':True,\
'T11:_Task':True,\
'T9:_Task':False,\
'G2:_Subgoal':False,\
'T17:_Task':True,\
'T18:_Task':True,\
```

```
'T19:Task':True,\
'T13:_Task':False,\
'T15:_Task':False,\
'T22:_Task':True,\
'T24:_Task':True,\
'T25:_Task':True,\
'T26:_Task':True,\
'T23:_Task':False,\
'T16:_task':False,\
'T17:_Task':True,\
'T14:_Task':False,\
'T4:_Task':False,\
'G3:_Subgoal':False,\
'G0: Goal':False}

pyhop(state, [\
('and_seq', 'T1:_Task', 'T2:_Task', 'T3:_Task'),\
('k_times_par', 'G1:_Subgoal', 'T1:_Task', '3'),\
('and_par', 'T9:_Task', 'T10:_Task', 'T11:_Task'),\
('k_tries', 'G2:_Subgoal', 'T9:_Task', '3'),\
('and_par', 'T13:_Task', 'T17:_Task', 'T18:_Task', 'T19:Task'),\
('and_seq', 'T23:_Task', 'T24:_Task', 'T25:_Task', 'T26:_Task'),\
('and_par', 'T16:_task', 'T22:_Task', 'T23:_Task'),\
('and_seq', 'T14:_Task', 'T15:_Task', 'T16:_task', 'T17:_Task'),\
('xor', 'T4:_Task', 'T13:_Task', 'T14:_Task'),\
('k_times', 'G3:_Subgoal', 'T4:_Task', '3'),\
('or_par', 'G0:_Goal', 'G1:_Subgoal', 'G2:_Subgoal', 'G3:_Subgoal'),\
], verbose=1)
```

Anexo VI

Código para o 1º Experimento do Modelo Mobee Mobile

```
from __future__ import print_function
from pyhop import *

import goda_operators
import goda_methods

print('')
print_operators()

print('')
print_methods()

state = State('Mobee Mobile')
state.objects = {\
'T1.1:_Render_view':True,\
'T1.21:_Validate_data':True,\
'T1.22:_Post_data':True,\
'T1.2:_Process_data':False,\
'T1.3:_Update_view':True,\
'T1:_Process_modification':False,\
'G8:_Modification_is_collected':False,\
'T2.1:_Render_view':True,\
'T2.21:_Validate_data':True,\
'T2.22:_Post_data':True,\
```

```
'T2.2:_Process_data':False,\
'T2:_Process_qualification':False,\
'G9:_Qualification_is_collected':False,\
'G3:_Manual_data_is_sent':False,\
'T3.11:_Fetch_GPS':True,\
'T3.12:_Fetch_triangulation':True,\
'T3.1:_Fetch_geolocation':False,\
'T3.21:_Validate_data':True,\
'T3.22:_Post_data':True,\
'T3.2:_Process_data':False,\
'T3:_Track_line_locator':False,\
'G10:_Line_locations_tracked':False,\
'G4:_Automatic_data_is_sent':False,\
'G1: Transport info is shared':False}

pyhop(state, [\
('and_par', 'T1.2:_Process_data', 'T1.21:_Validate_data', 'T1
   .22:_Post_data'),\
('and_seq', 'T1:_Process_modification', 'T1.1:_Render_view',
   'T1.2:_Process_data', 'T1.3:_Update_view'),\
('means_end', 'G8:_Modification_is_collected', 'T1:
   _Process_modification'),\
('and_seq', 'T2.2:_Process_data', 'T2.21:_Validate_data', 'T2
   .22:_Post_data'),\
('and_seq', 'T2:_Process_qualification', 'T2.1:_Render_view',
    'T2.2:_Process_data'),\
('means_end', 'G9:_Qualification_is_collected', 'T2:
   _Process_qualification'),\
('and_par', 'G3:_Manual_data_is_sent', 'G8:
   _Modification_is_collected', 'G9:
   _Qualification_is_collected'),\
('xor', 'T3.1:_Fetch_geolocation', 'T3.11:_Fetch_GPS', 'T3
   .12:_Fetch_triangulation'),\
('and_seq', 'T3.2:_Process_data', 'T3.21:_Validate_data', 'T3
   .22:_Post_data'),\
('try_op', 'T3:_Track_line_locator', 'T3.1:_Fetch_geolocation
   ', 'T3.2', 'T3.2:_Process_data'),\
```

```
51 ('means_end', 'G10:_Line_locations_tracked', 'T3:
     _Track_line_locator'),\
52 ('means_end', 'G4:_Automatic_data_is_sent', 'G10:
     _Line_locations_tracked'),\
53 ('and_par', 'G1:_Transport_info_is_shared', 'G3:
     _Manual_data_is_sent', 'G4:_Automatic_data_is_sent'),\
54 ], verbose=1)
```

Anexo VII

Código para o 2º Experimento do Modelo Mobee Mobile

```
1
2 from __future__ import print_function
3 from pyhop import *
4
5 import goda_operators
6 import goda_methods
7
8 print('')
9 print_operators()
10
11 print('')
12 print_methods()
13
14 state = State('Mobee Mobile')
15 state.objects = {\
16 'T1.1:_Render_view':True,\
17 'T1.21:_Validate_data':True,\
18 'T1.22:_Post_data':True,\
19 'T1.2:_Process_data':False,\
20 'T1.3:_Update_view':True,\
21 'T1:_Process_modification':False,\
22 'G8:_Modification_is_collected':False,\
23 'T2.1:_Render_view':True,\
24 'T2.21:_Validate_data':True,\
```

```
'T2.22:_Post_data':True,\
'T2.2:_Process_data':False,\
'T2:_Process_qualification':False,\
'G9:_Qualification_is_collected':False,\
'G3:_Manual_data_is_sent':False,\
'T3.11:_Fetch_GPS':True,\
'T3.12:_Fetch_triangulation':False,\
'T3.1:_Fetch_geolocation':False,\
'T3.21:_Validate_data':True,\
'T3.22:_Post_data':True,\
'T3.2:_Process_data':False,\
'T3:_Track_line_locator':False,\
'G10:_Line_locations_tracked':False,\
'G4:_Automatic_data_is_sent':False,\
'G1: Transport info is shared':False}

pyhop(state, [\
('and_par', 'T1.2:_Process_data', 'T1.21:_Validate_data', 'T1
   .22:_Post_data'),\
('and_seq', 'T1:_Process_modification', 'T1.1:_Render_view',
   'T1.2:_Process_data', 'T1.3:_Update_view'),\
('means_end', 'G8:_Modification_is_collected', 'T1:
   _Process_modification'),\
('and_seq', 'T2.2:_Process_data', 'T2.21:_Validate_data', 'T2
   .22:_Post_data'),\
('and_seq', 'T2:_Process_qualification', 'T2.1:_Render_view',
    'T2.2:_Process_data'),\
('means_end', 'G9:_Qualification_is_collected', 'T2:
   _Process_qualification'),\
('and_par', 'G3:_Manual_data_is_sent', 'G8:
   _Modification_is_collected', 'G9:
   _Qualification_is_collected'),\
('xor', 'T3.1:_Fetch_geolocation', 'T3.11:_Fetch_GPS', 'T3
   .12:_Fetch_triangulation'),\
('and_seq', 'T3.2:_Process_data', 'T3.21:_Validate_data', 'T3
   .22:_Post_data'),\
```

```
51 ('try_op', 'T3:_Track_line_locator', 'T3.1:_Fetch_geolocation
     ', 'T3.2', 'T3.2:_Process_data'),\
52 ('means_end', 'G10:_Line_locations_tracked', 'T3:
     _Track_line_locator'),\
53 ('means_end', 'G4:_Automatic_data_is_sent', 'G10:
     _Line_locations_tracked'),\
54 ('and_par', 'G1:_Transport_info_is_shared', 'G3:
     _Manual_data_is_sent', 'G4:_Automatic_data_is_sent'),\
55 ], verbose=1)
```

Anexo VIII

Instruções para Instalação do *Protótipo*

Ambiente

1. Faça o download do Eclipse 4.5
2. Help > Install new Software > Add
   (a) Name: Taom4e
   (b) Location: http://selab.fbk.eu/taom/eu.fbk.se.taom4e.updateSite/
3. Help > Eclipse Market Place > "PDE"
4. Clone o repositório: https://github.com/allienson/GODAtoPyhop
5. Substitua o arquivo **taom4eplatform.jar** presente em eclipse/plugins/it.itc.sra.taom4e.platform_0.6.3.1 pelo arquivo de mesmo nome em GODAtoPyhop/lib

Primeira Execução

Na primeira vez que o protótipo for executado é preciso fazer algumas configurações

1. Reinicie o Eclipse
2. Escolha o diretório onde foi clonado o repositório
3. Aceite as opções padrão
4. Clique com obtão direito no projeto
5. Run As > Eclipse Application (uma nova janela será aberta)

6. Na nova janela, Window -> Preferences -> TAOM4E -> PRISM GENERATOR
7. Desmarque 'Use standard'
8. Clique em 'Select'para selecionar um caminhoi para o diretório do template
9. Selecione o diretório dentro do projeto em src/main/resources/TemplateInput
10. Clique em 'Select' para escolher um caminho para o diretório de output
11. Escolha qualquer diretório
12. Clique em 'Select' para escolher o caminho do diretório que contém as Tools
13. Escolha qualquer diretório
14. Copie os binários do PRISM e do PARAM para uma pasta PATH sem sufixos de extensão
    (a) PATH/prism
    (b) PATH/param

Anexo IX

Instruções para Utilização do Tao4me com GODA

Criando um Projeto GODAtoPyhop

1. File > New Project
2. Clique com o botão direito na pasta do projeto
3. File > New > Other > Tropos Model

Sintaxe utilizada para nomear Tarefas e Objetivos

Tarefas e Objetivos devem ser nomeados de acordo com as regras abaixo:

```
IDENTIFICADOR: DESCRIÇÃO [REGRA]
```

- Objetivos precisam ser nomeados com o prefixo "G"(goal) seguido de um ID numérico único, seguido de ":", seguido de uma descrição textual do objetivo. Ex: G1: Descrição
- Tarefas precisam ser nomeadas com o prefixo "T"(task) seguido de um ID numérico único, seguido de ":", seguido de uma descrição textual da tarefa. Ex: T1: Descrição
  - O ID de uma tarefa deve ser relativo ao seu nível. Ex: Tarefa de nível 1 "T1", tarefa de nível 2 "T1.1"(filha de T1), tarefa de nível 3 "T1.11"(filha de T1.1)
- Todo nó que não é folha (que possui decomposições) deve possuir anotação de Regra GODA. Essas anotações devem estar entre "[ ]". Ex: G1: Descrição [G2;G3]
- Todo nó que for folha pode receber anotação, porém apenas da regra $opt(n)$