



Herbet de Souza Cunha

**Desenvolvimento de Software Consciente com Base em
Requisitos**

Tese de Doutorado

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio.

Orientador: Prof. Julio Cesar Sampaio do Prado Leite

Rio de Janeiro
Março de 2014



Herbet de Souza Cunha

Desenvolvimento de Software Consciente com Base em Requisitos

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Julio Cesar Sampaio do Prado Leite

Orientador
Departamento de Informática – PUC-Rio

Prof. Ruy Milidiu

Departamento de Informática – PUC-Rio

Prof. Daniel Schwabe

Departamento de Informática – PUC-Rio

Prof. Vera Maria Benjamim Werneck

Universidade do Estado do Rio de Janeiro – UERJ

Prof. Claudia Cappelli

Universidade Federal do Estado do Rio de Janeiro – UNIRIO

Prof. José Eugenio Leal

Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 11 de março de 2014

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Herbet de Souza Cunha

Graduou-se em Bacharelado em Ciência da Computação na UFPE em 2000. Recebeu o título de Mestre em Informática na PUC-Rio em 2007.

Ficha Catalográfica

Cunha, Herbet de Souza

Desenvolvimento de software consciente com base em requisitos / Herbet de Souza Cunha ; orientador: Julio Cesar Sampaio do Prado Leite. - 2014.

215 f. : il. (color.) ; 30 cm

Tese (doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2014.

Inclui bibliografia.

1. Informática – Teses. 2. Consciência de Software. 3. Requisito de consciência. 4. Desenvolvimento de software adaptativo. 5. i* (i-estrela). 6. Requisito não funcional (RNF). 7. Engenharia de requisitos. 8. Sistemas multiagente (SMA).. I. Leite, Julio Cesar Sampaio do Prado. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

Aos meus pais Lindinalva e Milton de Souza Cunha.

Agradecimentos

Sou plenamente consciente de que não teria conseguido concluir esse trabalho sem a ajuda valiosa de inúmeras pessoas, e agradeço sinceramente a todos.

Agradeço aos colegas do grupo de engenharia de requisitos da PUC-Rio pela convivência e troca de experiências sempre produtiva. André, Edgar, Elizabeth, Henrique, Joana, Letícia, Marília, Priscila e em especial a Eduardo com quem tive o prazer de dividir alguns trabalhos.

Agradeço especialmente ao Prof. Julio Leite pela orientação, inspiração e paciência. Pela disponibilidade em ouvir, argumentar e contra argumentar me levando a enxergar novos aspectos desse trabalho.

Agradeço aos amigos da Petrobras pelo incentivo desde o início, em especial a Anna Neville pelo carinho, apoio e cobrança. Ao amigo Ronnie Breno pelo incentivo e preciosa ajuda na obtenção dos dados para simulação.

Ao querido Sergio Andrade pela força, paciência e presteza em apresentar-me novos conceitos e elucidar os meus inevitáveis equívocos de interpretação.

Agradeço especialmente a minha mulher Judite Araujo, pelo apoio, incentivo, paciência e compreensão nos momentos difíceis.

A todos os que conviveram comigo durante esses anos, e que de alguma forma fizeram parte do meu caminho, meu muito obrigado.

Resumo

Cunha, Herbet de Souza; Leite, Julio Cesar Sampaio do Prado. **Desenvolvimento de software consciente com base em requisitos.** Rio de Janeiro, 2014. 215p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Consciência de software (*software awareness*) tornou-se um requisito importante na construção de sistemas com capacidade de autoadaptação. Para que aplicações de software possam melhor se adaptar a mudanças nos diversos ambientes em que operam, ter consciência (no sentido de perceber e entender esses ambientes e a seu próprio funcionamento nestes ambientes) é fundamental. Entretanto, mesmo em um nível básico aplicado a software, consciência é um requisito difícil de definir. Nosso trabalho propõe a organização de um catálogo para o requisito de consciência de software, com mecanismos para instanciação e uso do conhecimento armazenado neste catálogo na modelagem e implementação de software para problemas onde a autoadaptação, e por consequência consciência, sejam requisitos chave.

Palavras-chave

Consciência de Software; Requisito de Consciência; Desenvolvimento de Software Adaptativo; i* (i-estrela); Requisito Não funcional (RNF); Engenharia de Requisitos; Sistemas Multiagente (SMA).

Abstract

Cunha, Herbet de Souza; Leite, Julio Cesar Sampaio do Prado (Advisor).

Aware Software Development Based on Requirements. Rio de Janeiro, 2014. 215p. DSc Thesis – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software awareness has become an important requirement in the construction of self-adaptive systems. As such, the software should better adapt to changes in the various environments in which they operate, be aware of (in the sense of perceiving and understanding) these environment and be aware of its own operation in these environments. However, even at a basic level applied to software, awareness is a requirement difficult to define. Our work proposes the creation of a catalog to the awareness requirement through non-functional requirements patterns (NFR patterns). We also propose mechanisms for enabling the instantiation and use of the knowledge about awareness, represented in this catalog.

Keywords

Software Awareness, Awareness Requirement, Development of Adaptive Software, i* (i-star); Non-Functional Requirement; Requirement Engineering.

Sumário

1 Introdução	15
1.1. Motivação	15
1.2. Desafios de pesquisa	16
1.3. Estratégia adotada	18
2 Catálogo para o requisito de consciência de software	20
2.1. Refinamento do requisito de consciência de software	21
2.1.1. Consciência do contexto	23
2.1.2. Consciência do tempo	24
2.1.3. Consciência de relações sociais	25
2.1.4. Consciência de autocomportamento	26
2.2. SIG de Consciência de Software Software	28
2.3. Operacionalização de consciência de contexto.	30
2.3.1. Operacionalização de consciência do ambiente físico	31
2.3.2. Operacionalização de consciência do ambiente computacional	32
2.3.3. Operacionalização de consciência de localização	34
2.4. Operacionalização de consciência do tempo	38
2.5. Operacionalização de consciência do autocomportamento	38
2.6. Operacionalização de consciência do contexto social	43
2.6.1. Operacionalização de consciência de normas	44
2.6.2. Operacionalização de consciência das relações sociais	47
2.6.3. Operacionalização de consciência de usuários	49
2.7. Uso e evolução do catálogo	53
2.8. Considerações sobre o catálogo de consciência de software	55
3 Modelagem do requisito de consciência de software	57
3.1. Abordagem orientada a meta	59
3.2. Engenharia de software orientada a agentes	61
3.3. Modelagem com i* (i-estrela)	63
3.3.1. i* - Modelo SD	64
3.3.2. i* - Modelo SR	66
3.3.3. i* - Modelo SA (Strategic Actor)	69
3.4. Representação descritiva dos modelos i* em iStarML	72

3.4.1. Acrescentando abstrações de consciência aos modelos de requisitos	76
3.5. Extensão do framework i*	77
3.6. Considerações sobre a modelagem do requisito de consciência	83
3.7. Processo de definição do requisito de consciência	91
3.8. Heurísticas para uso do catálogo na definição do requisito de consciência	92
4 <i>Framework istarjade</i>	97
4.1. JADE	98
4.1.1. Agentes em JADE	100
4.1.2. Comportamento dos agentes em JADE	101
4.2. Mapeamento de elementos básicos de i* para JADE	106
4.2.1. Acréscimo de atributos em iStarML	109
4.3. Criação de agentes e adição de comportamentos	112
4.4. Consciência e avaliação de alternativas;	124
4.5. Considerações sobre o framework <i>istarjade</i>	126
5 Validação Experimental	130
5.1. Caso 1 – Sistema de controle automático de porta	130
5.1.1. Descrição do problema:	130
5.1.2. Dados da simulação:	131
5.2. Caso 2 – Sistema de controle de elevadores	136
5.2.1. Descrição do problema:	136
5.2.2. Cenário com um elevador (agente elevator1)	136
5.2.3. Cenário com dois elevadores	141
5.2.4. Cenário com três elevadores	148
5.3. Caso 3 – Sistema multiagente para operação na bolsa de valores	156
5.3.1. Descrição do problema	156
5.3.2. Considerações sobre o problema	157
5.3.3. Construção do modelo de aprendizado supervisionado	158
5.3.4. Simulação e dados teste	161
5.3.5. Simulação de Operação	168
5.3.6. Resultados da Simulação	168
5.3.7. Considerações sobre os resultados da simulação	170
5.3.8. Considerações sobre a validação experimental	171
6 Conclusão	173
6.1. Trabalhos relacionados	173

6.2. Contribuições do trabalho	175
6.3. Limitações do trabalho	176
6.4. Trabalhos futuros	177
6.5. Considerações finais	178
Referências bibliográficas	180

Listas de Figuras

Figura 1 – SIG de Consciência de Software	29
Figura 2 - Padrão Questão (<i>Question Pattern</i>) para consciência de contexto	36
Figura 3 – Operacionalização – ambiente computacional	36
Figura 4 - Operacionalizações - ambiente físico	37
Figura 5 - Operacionalizações - localização	37
Figura 6 - Padrão questão - consciência de autocomportamento	42
Figura 7 - Operacionalizações - consciência de autocomportamento	43
Figura 8 - Padrão questões - consciência do contexto social	46
Figura 9 - Operacionalizações - consciência de norma	46
Figura 10 - Operacionalizações - consciência de relações sociais	49
Figura 11 - operacionalizações sugeridas para consciência de usuários	54
Figura 12 - Arquitetura de referência da IBM - computação autônoma	60
Figura 13 - Tipos de dependência em um modelo SD	66
Figura 14 - Diagrama UML - Ator e seus refinamentos (Leite et al., 2007)	69
Figura 15 - Diagrama UML - Relações entre atores (Leite et al., 2007)	70
Figura 16 - <i>SRConstruct</i> para a meta flexível de consciência	83
Figura 17 - modelo i* estendido para problema da porta automática	84
Figura 18 - Visão geral das etapas para definição do requisito de consciência	93
Figura 19 - Arquitetura plataforma de agentes FIPA (Bellifemine et al., 2010)	99
Figura 20 - Arquitetura de agentes detalhada FIPA (Bellifemine et al., 2010)	99
Figura 21 - Ciclo de vida de um Agente FIPA (Bellifemine et al., 2010)	101
Figura 22 - Caminho de execução <i>thread</i> de Agente (Bellifemine et al., 2010)	102
Figura 23 - Modelo UML da hierarquia <i>Behaviour</i> (Bellifemine et al., 2010)	104
Figura 24 - Diagrama do SMA para controle de porta	108
Figura 25 - Diagrama de classe para os elementos intencionais	108
Figura 26 - Diagrama de classe UML, com atores, elos e dependência	109
Figura 27 - Parâmetros para execução do <i>framework</i> istarjade	113
Figura 28 - GUI de JADE após criação do agente loader_agent	114
Figura 29 - Método <i>loadIstarModel()</i> da classe IstarModelLoader	116
Figura 30 - Trecho de código - elementos básicos são carregados.	117
Figura 31 - Trecho de código - elementos principais são carregados	118
Figura 32 - Trecho de código - dependências são carregadas	119

Figura 33 - Trecho de código - método setup() de IstarJadeAgent	120
Figura 34 - Interfaces definidas no <i>framework</i> istarjade	128
Figura 35 – Classes de comportamentos do <i>framework</i> istarjade	128
Figura 36 – Algumas classes do <i>framework</i> istarjade	129
Figura 37 - modelo i* para o cenário com um elevador	137
Figura 38 - Modelo i* para o cenário com dois elevadores	143
Figura 39 - Modelo i* para o cenário com três elevadores	149
Figura 40 - Modelo de ascendência analítica do Gartner (Laney e Kart, 2012)	158
Figura 41 – Máquina de estados do modelo de decisão do Operador	159
Figura 42 - Valor de VALE5 no intervalo de treinamento (13/04 a 14/10/2009)	161
Figura 43 - Modelo i* do SMA para operação na bolsa	163
Figura 44 - Curva com a evolução de VALE5 durante simulação	166
Figura 45 - Curva com a evolução de VALEK48 e VALEK50 no período	166
Figura 46 - Amostra dos dados usados na simulação	167
Figura 47 - Resultado da simulação sem e com uso de stop 2% (loss e gain)	171

Lista de Tabelas

Tabela 1 - símbolos da BNF estendida utilizados	72
Tabela 2 - Conceitos de i* e tags iStarML (Cares et al., 2007)	73
Tabela 3 - Tags complementares de iStarML	73
Tabela 4 - Sintaxe de iStarML / <iastarml> syntax	74
Tabela 5 - Sintaxe de ator / <actor> syntax	74
Tabela 6 - Sintaxe de elementos intencionais / <ielement> syntax	74
Tabela 7 - Sintaxe para fronteira dos atores / <boundary> syntax	75
Tabela 8 - Sintaxe para elos entre elementos / <ielementLink> syntax	75
Tabela 9 - Sintaxe para dependência entre atores / <dependency> syntax	75
Tabela 10 - Sintaxe para elos entre atores / <actorLink> syntax	76
Tabela 11 - Sintaxe de elementos intencionais estendida / <ielement> syntax	81
Tabela 12 - Sintaxe para elos entre elementos intencionais estendida	82
Tabela 13 - Modelo para especificação de consciência	85
Tabela 14 - Heurísticas para auxiliar a definição do requisito de consciência	95
Tabela 15 – Mapeamento entre elementos de i*, iStarML e Jade	107
Tabela 16 - Sintaxe para elos entre elementos intencionais modificada	110
Tabela 17 – Dados da simulação do caso do SMA para controle de porta	132
Tabela 18 - Dados para simulação do cenário com um elevador	138
Tabela 19 - Dados para simulação do cenário com dois elevadores	142
Tabela 20 - Dados para simulação com três elevadores	149
Tabela 21 – Resumo das ações durante a primeira hora da simulação	168

Lista de Abreviaturas e Siglas

FIPA	<i>The Foundation for Intelligent Physical Agents</i>
GORE	<i>Goal Oriented Requirement Engineering</i>
GQM	<i>Goal Question Metric</i>
GQO	<i>Goal Question Operationalization</i>
GUI	<i>Graphical User Interface</i>
JVM	<i>Java Virtual Machine</i>
MAS	<i>MultiAgent System</i>
MAPE	Monitorar, Analisar, Planejar, Executar
OCL	<i>Object-Constraint Language</i>
NFR	<i>Non-functional Requirements</i>
SA	<i>Strategic Actor</i>
SADT	<i>Structured Analysis and Design Technique</i>
SD	<i>Strategic Dependency</i>
SR	<i>Strategic Rationale</i>
SIG	<i>Softgoal Interdependency Graph</i>

1

Introdução

Neste capítulo discutiremos o requisito de consciência de software (software awareness) e apresentaremos a definição de consciência que adotaremos no decorrer desta tese. Apresentaremos a motivação de nossa pesquisa sobre o requisito de consciência, os desafios de pesquisa ao tratar desse assunto e a estratégia de solução que adotaremos na tese. No final do capítulo, faremos um breve resumo de como o texto está organizado.

1.1. Motivação

Consciência¹ tornou-se um requisito importante na construção de sistemas com capacidade de autoadaptação. Diversos trabalhos têm sido desenvolvidos voltados para a consciência de contexto (*context-awareness*), impulsionados principalmente pelo aumento no número de aplicações desenvolvidas para ambientes móveis, computação vestível e computação ubíqua. Estas aplicações necessitam ter a capacidade de reconhecer e tirar proveito das constantes mudanças de contexto a que são submetidas (Abowd et al., 1998), em outras palavras, necessitam se adaptar a diferentes contextos operacionais.

Além das mudanças nos contextos operacionais, softwares autoadaptativos, muitas vezes, precisam lidar com as mudanças no contexto em um sentido mais amplo: mudanças nas condições do "ambiente externo" (por exemplo, a ocorrência de eventos externos, ou disponibilidade de serviços e recursos) ou mudanças no funcionamento do sistema de software devido a causas internas (falta ou não cumprimento de metas). Como o software tornou-se onipresente, a criação de sistemas de software que são *auto** tem sido o foco de desenvolvimento em computação autônoma (Hinchey e Sterritt, 2006) - o

¹ Embora empreguemos o termo consciência em nosso trabalho, esse termo é aplicado a software em sentido mais básico, restringindo-se a "estar a par do que acontece", não abrangendo nenhum tipo de questão metafísica. Empregamos o termo consciência como tradução do termo *awareness* em inglês.

termo *auto**, tradução de *self-**, refere-se a uma classe de sistemas autoadaptativos e autogerenciados com capacidade de autonomamente se adaptar a novas circunstâncias e se recuperar de faltas (Dalpiaz et al., 2009). Além disso, muitos dos aplicativos que dependemos atualmente são sistemas de tipo aberto, como na Teoria Geral dos Sistemas (Von Bertalanffy, 1968), ou seja, são passíveis de mudar devido a condições externas - por exemplo, serviços bancários, operações de câmbio, planejamento de viagem, mercado de ações. Este tipo de aplicação reúne diferentes atores - cada um sendo uma entidade autónoma, com suas próprias metas e políticas particulares. Os atores interagem uns com os outros de forma a cumprir os respectivos objetivos. Nestes tipos de aplicações, os agentes de software precisam se adaptar aos sistemas abertos e interagir com outros atores de software, que são referidos como agentes de software. A fim de interagir com sucesso com outros agentes de software, um agente precisa estar ciente da existência dos outros agentes e saber como poderia se relacionar com eles (como eles poderiam ajudar a alcançar seus objetivos).

Sistemas autogerenciados, supostamente, têm a capacidade de satisfazer seus objetivos na mudança de ambientes, sem necessitar de supervisão humana, e continuar a operação em diferentes condições. Para isso, é necessário que sistemas *auto** estejam conscientes das metas que devem satisfazer, as condições para a satisfação das metas, as possíveis ações ou alternativas que poderiam levar à satisfação das metas, e algum mecanismo de raciocínio para escolher uma ou mais alternativas disponíveis.

Por estas razões, acreditamos que a consciência é uma qualidade crucial para satisfazer a contento as qualidades *auto**. Usamos a expressão ‘satisfazer a contento’ como tradução de *satisfice* - termo introduzido por Herbert Simon (Simon, 1968), em vez do termo satisfazer, para enfatizar a diferença entre a consciência e as qualidades de *auto** dos objetivos comuns que os atores de software têm de alcançar.

1.2. Desafios de pesquisa

Considerando o que discutimos anteriormente, podemos dizer que ter consciência ajuda a melhorar a capacidade de autoadaptação e autogerenciamento do software. Mas o que realmente consideramos que significa consciência para software?

A Enclopédia de Filosofia de Stanford refere-se a "estar ciente" como um estado de consciência, onde "... um estado mental consciente é simplesmente um estado mental, um está ciente de estar em (Rosenthal 1986, 1996)". E explica que "ter um desejo consciente por uma xícara de café é ter um desejo e também ser simultaneamente e diretamente consciente de que a pessoa tem um tal desejo. Pensamentos e desejos inconscientes, nesse sentido, são simplesmente aqueles que temos sem ter consciência de tê-los, seja a nossa falta de autoconhecimento resultado de simples desatenção ou de causas psicanalíticas mais profundas" (Stanford Encyclopedia for Philosophy, 2013).

A Enclopédia Britannica Academic Edition refere-se à autoconsciência como tornar-se consciente de seus próprios estados emocionais, características, habilidades e potenciais para a ação e refere-se à consciência como uma condição psicológica definida pelo filósofo Inglês John Locke como "a percepção do que se passa na própria mente de um homem" (Encyclopedia Britannica Academic Edition, 2013).

O termo consciência é descrito como "ter conhecimento de" pelo WordNet on-line (WordNet, 2013). No The Free Dictionary (The Free Dictionary, 2013) consciência é descrito como "ter conhecimento ou percepção" (*knowledge* ou *cognizance*) e "estar consciente implica adquirir conhecimento através das próprias percepções ou por meio de informação". Já na Wikipedia (Wikipedia, 2013) o termo é definido como "o estado ou a capacidade de perceber, sentir, ou estar consciente dos acontecimentos, objetos ou padrões sensoriais".

Em nosso trabalho, adotamos a seguinte definição para consciência de software: "a capacidade do software de adquirir conhecimento sobre o que acontece no ambiente no qual está inserido e sobre seu próprio comportamento, através de suas próprias percepções ou por meio de informações externas".

Nosso trabalho é focado em consciência de software (*software awareness*), com uma abordagem baseada na perspectiva da engenharia de requisitos. Uma questão central em nosso trabalho é que, à semelhança do que Professor John Mylopoulos disse sobre o requisito de transparência "ser uma qualidade interessante, pois torna-se necessário anexar modelos de requisitos ao software", para melhorar o requisito de consciência de software é necessário que o software seja consciente dos seus requisitos.

Objetivamente, os problemas que nosso trabalho se propõe a abordar são:

- Nos casos em que o software precise de algum nível de consciência, ou seja, nos casos em que consciência seja um requisito para o software, como tratar este requisito?
- Como descobrir que tipo de consciência é necessário e em qual nível?
- Como modelar o requisito de consciência?
- Como analisar o requisito de consciência?
- Como guiar o atendimento do requisito de consciência na implementação do software?

1.3. Estratégia adotada

Para responder adequadamente as questões apresentadas anteriormente, nós propomos uma abordagem em três frentes inter-relacionadas, a saber:

- A elaboração de um catálogo para o requisito de consciência de software organizando conhecimento existente na literatura sobre este requisito e provendo aos desenvolvedores de software a possibilidade de reusar esse conhecimento quando necessário.
- Uma abordagem para modelagem e análise do requisito de consciência de software.
- Por temos convicção que há casos em que o requisito de consciência não pode ser resolvido em tempo de desenho, proporemos também alguns mecanismos, implementados em um *framework* experimental, para instanciar o requisito de consciência em tempo de execução.

Consciência, bem como as qualidades auto*, são requisitos não funcionais (NFRs). Portanto, não há critérios claros para a sua satisfação *a priori* – estes requisitos podem apenas ser considerados suficientemente satisfeitos em uma situação específica. Para construção do catálogo de consciência de software usaremos o *framework NFR* (Chung et al., 2000) e apresentaremos o catálogo na forma de padrões para requisitos não funcionais - *NFR Patterns* (Supakkul et al., 2010). A elaboração do catálogo será apresentada no Capítulo 2 – Catálogo para o requisito de consciência de software.

Como o requisito de consciência está diretamente ligado aos requisitos de autoadaptação e autogerenciamento, optamos por usar estratégias de modelagem orientadas a meta e a agentes para modelagem e análise do requisito de consciência, que apresentaremos no Capítulo 3 – Modelagem do requisito de consciência.

No Capítulo 4 – *Framework* para implementação, apresentamos o *framework* experimental que desenvolvemos para fins de validação de nossa pesquisa. Este *framework* possibilita ao software lidar com o requisito de consciência em tempo de execução.

No Capítulo 5 – Validação experimental, apresentaremos os resultados de três casos de validação, dois deles com mais de um cenário, das propostas de nossa pesquisa.

Finalmente, apresentaremos nossas considerações sobre a pesquisa e nossas conclusões no Capítulo 6 – Considerações e Conclusões.

2

Catálogo para o requisito de consciência de software

Neste capítulo apresentamos um catálogo para o requisito de consciência de software. O catálogo, organizado na forma de um Softgoal Interdependency Graph (SIG) complementado por padrões para requisitos não funcionais (NFR Patterns), visa armazenar de forma organizada conhecimento sobre consciência que pode ser reusado por desenvolvedores nos casos em que o requisito de consciência seja necessário. Neste catálogo, o requisito de consciência é refinado em subtipos para os quais são elencadas algumas operacionalizações representando alternativas para ‘satisfação a contento’ do requisito de consciência.

Apesar do desafio de conceituar o requisito consciência de software, discutido no capítulo anterior, em cenários que autoadaptação é um requisito não funcional do software, algum nível de consciência é necessária. O tipo e o nível de consciência envolvidos podem variar de acordo com o problema e com o nível de satisfação desejado.

Os problemas nos quais a autoadaptação é um requisito são, em geral, melhor modelados através de modelos intencionais. A ideia geral em um modelo intencional é que as metas (ou intenções) são as principais entidades para orientar o comportamento do software, sendo possível projetar diferentes alternativas para a satisfação de uma meta. A possibilidade de escolher uma alternativa de acordo com a situação subjacente aumenta a capacidade do software de adaptar seu comportamento. No que diz respeito ao requisito de consciência, é necessário que o software tenha a capacidade de perceber, por si mesmo ou a partir de informação externa, as diferentes situações que podem enfrentar.

Nós organizamos conhecimento sobre o requisito de consciência em um catálogo para ajudar a definição (elicitação e análise) de consciência de software em um processo de engenharia de requisitos. Este catálogo foi desenvolvido com base no framework *NFR* (Chung et al., 2000), na forma de um *Softgoal Interdependency Graph* (SIG), complementado por padrões de descrição de requisitos não funcionais voltados a auxiliar o reuso - *NFR Patterns* (Supakkul et al., 2010).

Tanto no *framework NFR*, quanto em *i** (i-estrela) (Yu, 1995), dois trabalhos que tomamos como base, os requisitos não funcionais podem ser representados por metas flexíveis (*softgoal*), enquanto objetivos centrais do software são representados por metas (*goal*). Metas flexíveis são determinadas por um tipo - que representa a qualidade, tais como precisão e segurança – e um tema – o contexto para o qual a meta flexível é aplicada, como, por exemplo, contas bancárias. No NFR Framework, existem três tipos de metas flexíveis:

- (i) NFRs - que são requisitos não funcionais a serem ‘satisfeitos a contento’;
- (ii) Operacionalização de metas flexíveis - técnicas de desenvolvimento que podem ajudar satisfazer a contento os NFRs;
- (iii) Argumentação - usada para justificar decisões.

A vantagem de usar um SIG como um catálogo para compartilhar o conhecimento sobre um requisito não funcional (NFR) é que, através dos métodos do catálogo, o NFR pode ser refinado, reduzindo seu nível de abstração. No âmbito de requisitos não funcionais, uma meta flexível de requisito não funcional (*softgoal NFR*) pode ser refinada por tipo ou por assunto. Através do refinamento por tipo, os tipos de nível (de abstração) mais elevados, como consciência, são decompostos em tipos ou subtipos de nível mais baixo. A operacionalização do tipo de nível superior é alcançada pelas operacionalizações dos subtipos. Ao fazer isso, a tarefa de encontrar possíveis operacionalizações (soluções) torna-se mais fácil. Seguindo esse raciocínio, identificamos algumas soluções possíveis para cada subtipo de consciência de software.

2.1. Refinamento do requisito de consciência de software

Um aspecto interessante sobre consciência é que ela é popularmente vista como um conceito relativo. De acordo com a Wikipedia – usamos Wikipedia como fonte de uma visão popular sobre o conceito – a consciência é descrita como "o estado ou a capacidade de perceber, sentir, ou estar consciente de eventos, objetos ou padrões sensoriais". A descrição do conceito é complementada com "Um animal pode ser parcialmente consciente (*aware*),

pode ser subconscientemente consciente (*aware*), ou pode ser, de forma aguda, inconsciente (*unaware*) de um evento. Consciência pode ser focada em um estado interno, como um sentimento visceral, ou em eventos externos por meio da percepção sensorial". Deixando de lado o fato do sujeito ser um animal em vez de software, este conceito reforça dois aspectos de nossa opinião: como é um conceito relativo, podendo um indivíduo ser parcialmente consciente, combina perfeitamente com as características de requisitos não funcionais. O outro aspecto é que a consciência pode surgir a partir de percepção ou de eventos externos. Tomando como base nossa definição de consciência - a capacidade do software de ter conhecimento, através de suas próprias percepções ou por meio de informações externas, sobre o que acontece no ambiente em que está inserido, e sobre seu próprio funcionamento - trabalhamos para alcançar uma compreensão mais detalhada de como lidar com esse requisito.

Inicialmente, nesta definição, há alguns aspectos importantes a destacar: a primeira é que, ser consciente significa adquirir conhecimento. Em alguns contextos em que a consciência é um requisito, a maioria deles relacionados com qualidades de auto*, algum conhecimento tem de ser adquirido em tempo de execução. Os outros aspectos importantes são as formas do software adquirir conhecimento: sua própria percepção ou por meio de informações externas. Estes aspectos: conhecimento a ser adquirido, percepções do software e obtenção de informações externas (para adquirir conhecimento), guiaram nossa pesquisa, a fim de melhorar a consciência de software. O primeiro desafio foi responder à pergunta: "Qual (tipo de) conhecimento o software precisa para ser consciente?".

Para responder à pergunta, começamos a explorar o aspecto ou dimensões de consciência. Em uma primeira abordagem, é possível distinguir três aspectos diferentes: consciência do contexto (ou ambiente) em que o software está inserido, consciência de seu próprio comportamento e consciência das interações sociais. Propomos esta decomposição por estes serem aspectos fundamentais para a adaptação de software: é necessário ao software perceber o ambiente em que irá operar e ao qual precisa se adaptar, assim como perceber as mudanças nesse ambiente. Como adaptação envolve mudanças no comportamento do próprio software, é importante que o mesmo perceba e analise o seu próprio comportamento para que possa corrigi-lo para melhor atender seus objetivos. O terceiro aspecto, as interações sociais, é importante para consciência de software na medida em que ajuda a na adaptação em

ambientes com a presença de outros atores com os quais deva interagir, especialmente em ambientes abertos.

Estes aspectos da consciência de software são discutidos a seguir.

2.1.1. Consciência do contexto

Consciência do ambiente externo, a que chamaremos de consciência de contexto (*context-awareness*) por ser o termo mais difundido na literatura, é um candidato a subtipo de consciência de software. Há muita literatura voltada à consciência do contexto afirmando a importância, para software autoadaptativo, de compreender o contexto em que precisa interoperar (Schmidt, 2002; Schilit et al., 1994; Ward et al., 1997; Korteum et al., 2008). O aumento da computação ubíqua, naturalmente, demandou que o software se tornasse consciente de diferentes características de ambientes em que necessitasse operar. Com isso, consciência de contexto, ou mais especificamente como criar aplicativos que são conscientes de contexto, é uma questão central para a pesquisa em Computação Ubíqua (Schmidt, 2002; Weiser, 1993).

Muitos pesquisadores (Schilit et al., 1994; Ward et al., 1997; Korteum et al., 2008) consideram sistemas conscientes de contexto como sendo sistemas que mudam dinamicamente ou adaptam o seu comportamento com base no contexto reconhecido. Nós não consideramos a adaptação do comportamento com base no contexto como uma característica de consciência de contexto, mas como uma característica de autoadaptação do software – em nossa visão, a consciência do contexto é uma condição prévia para que o software se adapte.

Passando para uma descrição mais refinada, nós especializamos consciência do contexto em três (sub) tipos de contexto: ambiente de computacional; ambiente físico e localização. Decidimos agrupar sob ambiente computacional os aspectos relativos à operação do software, separando estes do ambiente físico que o software deve perceber. Dos aspectos relativos ao ambiente físico, destacamos do grupo o requisito de localização por se tratar com tecnologias específicas e sedimentadas que se tornaram mais populares com a explosão do uso de dispositivos móveis.

Como a variedade de dispositivos aumentou e o software tornou-se onipresente, a complexidade no ambiente computacional em que o software tem que operar aumentou proporcionalmente. E o software precisa ser consciente dos recursos disponíveis, da conectividade, da capacidade de rede, das

interfaces de middleware. Alguns software, especialmente os embarcados, precisam ser conscientes de características físicas, a fim de adaptar-se a elas. Por exemplo, software para controlar a temperatura em um quarto precisa estar ciente da temperatura real na sala, e se ela não for adequada, agir para corrigi-la. Da forma análoga, o nível de ruído, a presença de pessoas, a presença de fumaça, são outros exemplos de características no ambiente físico que algum software precisa estar ciente. Localização é uma dimensão especial do contexto. Ser consciente do contexto é importante exatamente porque o contexto muda. Consciência de localização refere-se a sistemas que podem passiva ouativamente determinar sua localização. O termo se aplica a navegação, localização em tempo real, e suporte ao posicionamento com escopo global, regional ou local. Com a utilização massiva de dispositivos móveis, a localização está em constante mutação. Uma nova classe de aplicativos móveis, em que a localização do usuário é um requisito fundamental, tem sido desenvolvida recentemente. Tais aplicações podem, por exemplo, adaptar o seu comportamento publicando a localização de um usuário aos membros apropriados de uma rede social ou permitir que varejistas publiquem ofertas especiais para clientes potenciais que estejam por perto.

Ter consciência de contexto para software significa, em suma, ser consciente do ambiente no qual está inserido, embora tenhamos distinguido os aspectos sociais do ambiente em um subtipo de consciência específico que chamando de consciência de relações sociais.

2.1.2. Consciência do tempo

Outra dimensão importante do contexto é o tempo, não só para sistemas autoadaptativos. À medida que os ambientes dinâmicos se tornam mais difíceis, o tempo é uma parte crucial de qualquer contexto. É importante estar ciente de quando o contexto muda. Ao contrário de outros autores, consideramos o tempo como uma dimensão ortogonal ao contexto e às dimensões de consciência. Fazemos isso, em parte, porque consideramos outras dimensões (ou subtipos) de consciência além de contexto. Não estamos dizendo que o tempo não é uma dimensão do contexto, mas nós optamos por uma visão diferente em que o tempo é importante para a consciência como um todo.

2.1.3. Consciência de relações sociais

Nós agrupamos sob consciência de relações sociais algumas características importantes para a consciência de software, especialmente em ambientes abertos e distribuídos. Os sistemas abertos envolvem participantes autônomos e heterogêneos que interagem para atingir suas respectivas metas (Dalpiaz et al., 2010). Neste sentido, é importante estar ciente de quem são os outros participantes, que chamamos de atores. Além disso, é necessário estar ciente dos papéis desempenhados por todos os atores, incluindo o próprio software. Uma classe de atores é particularmente especial: os usuários. Muitos software pretendem adaptar o seu comportamento e aparência de acordo com os papéis dos usuários ao interagir com eles. Há casos em que é desejável personalizar o comportamento do software para cada usuário específico. Nestes casos, o software precisa ter consciência do usuário, incluindo sua identidade (quem é o usuário, quais são as suas preferências), o papel que o usuário está desempenhando e suas metas individuais em cada situação. Na interação com outros atores de software (agentes), o software precisa ter consciência desses outros atores de forma semelhante: é preciso conhecer a identidade do outro software ou de quem eles representam, que papéis desempenham e quais são suas metas.

Para que ocorra interação entre os diferentes atores de software, a base para colaboração precisa ser criada. Em ambientes fechados, todos os participantes são conhecidos a priori e esta base pode ser totalmente estabelecida em tempo de desenho. Entretanto, ainda em ambientes fechados – apesar de poder ser feito em tempo de desenho – o software precisa minimamente conhecer com que atores ele irá interagir e como esta interação pode ser realizada. Em ambientes abertos, por outro lado, não é possível conhecer a priori, em tempo de desenho, todos os outros atores com os quais o software terá de interagir. Em ambientes abertos e distribuídos, ter consciência dos outros atores é crucial, dado que o ambiente "social" em que o software irá operar é aberto e suscetível a mudanças. Mesmo que a base para interação em ambientes abertos possa ser estabelecida em tempo de desenho, em muitos casos, é necessário negociar acordos com outros atores em tempo de execução para estabelecer efetivamente a cooperação. Se os outros atores são autônomos, não há garantia de que um acordo será alcançado em tempo de execução. Assim, além de estar ciente dos outros atores e dos respectivos

papéis que desempenham, em ambientes abertos "habitados" por agentes autônomos é necessário estar ciente dos mecanismos (ou protocolos) que permitem aos atores negociar e, eventualmente, chegar a um acordo.

As normas são outro aspecto "social" de ambientes abertos que os agentes tem que observar. Normas podem ser vistas como a definição de padrões de comportamento e visam regular o funcionamento do ambiente como um todo. Elas têm sido amplamente propostas como um meio de coordenação e regulação dos comportamentos dos agentes individuais para garantir propriedades globais de um sistema multiagente (Dybalova et al., 2013). Além da abordagem de sistemas multiagente normativos, os padrões de comportamento esperados na maioria dos casos são inerentes ao domínio da aplicação (por exemplo, o limite de velocidade em uma estrada). Há duas maneiras diferentes pelas quais normas podem ser implementadas: de forma endógena, integrando-os nos programas dos agentes individuais (um carro autônomo pode ser programado para não exceder o limite de velocidade), ou de forma exógena por componentes adicionais que observam e avaliam os comportamentos dos agentes, a fim de verificar o cumprimento ou violação das normas (monitorar as câmeras de estrada carros de velocidade e registrar as identidades dos carros que violem limitações de velocidade) (Dybalova et al., 2013).

Em sistemas multiagente, onde as normas são implementadas de forma exógena, a regulação é realizada através do processamento das normas em tempo de execução, o que significa que os agentes (atores de software) devem estar ciente das normas que devem observar. Além disso, agentes se beneficiam em fazer parte de uma sociedade pela satisfação das metas cujo sucesso depende das habilidades de outros agentes. Uma vez que sociedades são controladas por normas, agentes devem ser capazes de identificar as diferentes relações em que eles podem ser envolvidos, devido às normas, a fim de agir de forma adequada (y López e Luck, 2004). Em outras palavras, os agentes devem ser conscientes das normas nas sociedades em que estejam envolvidos.

2.1.4. Consciência de autocomportamento

O tipo mais desafiador da consciência de software deve ser a consciência de autocomportamento. Na língua inglesa, os termos *awareness* e

consciousness se confundem e muitas vezes são considerados sinônimos. Em uma das noções, consciência - como tradução do inglês *consciousness* - é vista como uma qualidade humana ainda não totalmente compreendida pela ciência, sendo um ponto de discussão da filosofia. Como podemos implementar em software tal conceito difícil? Neste ponto, precisamos nos ater a nossa definição básica de consciência para software: *a capacidade do software de adquirir conhecimento sobre o que acontece no ambiente no qual está inserido e sobre seu próprio comportamento, através de suas próprias percepções ou por meio de informações externas.* Nesse sentido, a consciência de software é uma qualidade inspirada na noção de consciência humana, mas nem de longe é a mesma coisa. No entanto, a qualidade de estar consciente sobre seu próprio comportamento - ou a capacidade de adquirir conhecimento sobre seu próprio comportamento - em alguma extensão é essencial para se adaptar software. Para software, se autoadaptar significa adaptar autonomamente seu comportamento de acordo com a situação a fim de satisfazer suas metas, inclusive as metas flexíveis. O primeiro aspecto que consideramos que um software precisa estar consciente sobre seu próprio comportamento são as suas metas.

As metas oferecem várias vantagens, como por exemplo, critérios precisos para a completude satisfatória de uma especificação de requisitos (em relação às metas das partes interessadas - *stakeholders*). Metas também possibilitam a exploração do racional de cada requisito individualmente, justificando sua pertinência, a investigação de propostas de sistemas alternativos e a detecção e resolução de conflitos (Van Lamsweerde, 2001; Souza, 2012).

Para ser consciente de suas metas, o software precisa conhecer quais são suas metas, e para cada meta qual é a situação em relação à sua satisfação. Além disso, o software precisa conhecer as possíveis alternativas ou estratégias para satisfazer as suas metas, e as limitações que podem restringir a adoção das possíveis alternativas. Adicionalmente, o software precisa também conhecer as suas metas flexíveis (*softgoals*), e para cada meta flexível qual é a situação em relação à sua ‘satisfação a contento’.

Consideramos que mais dois aspectos relacionados à consciência são importantes em um cenário em que o software precise se adaptar: raciocinar sobre as alternativas avaliando-as sob a perspectiva da satisfação de suas metas, e quanto eficaz é o desempenho do software tanto na satisfação de suas metas quanto na satisfação a contento de suas metas flexíveis.

Para raciocinar sobre a satisfação de suas metas, o software precisa avaliar cada alternativa disponível. Ou seja, para cada alternativa é necessário avaliar se ela se aplica a uma situação real específica. Isto pode ser feito com base na percepção do contexto em que o software opera. Se mais de uma alternativa se aplica, é preciso avaliá-las em relação à satisfação das metas. Essa avaliação é, em última análise, o resultado de como o software percebe as alternativas em um contexto específico. O mesmo raciocínio é válido também para a avaliação das alternativas na satisfação a contento de suas metas flexíveis.

Outro aspecto importante é a eficácia na satisfação das metas e metas flexíveis. Em última análise, a eficácia da satisfação das metas e metas flexíveis representa a eficácia do software em si. É importante conhecer o quanto uma alternativa é eficaz em uma situação específica. Nos casos em que determinadas alternativas não sejam eficazes, esta informação pode servir para melhorar o mecanismo de escolha de alternativas. Nós destacamos eficácia porque ela resume se o software consegue satisfazer suas metas, e satisfazer a contento suas metas flexíveis, em diferentes situações que enfrenta. O aspecto mais geral que visamos captar com a eficácia é o funcionamento do software, e por isso optamos por usar funcionamento como o terceiro aspecto de consciência do autocomportamento.

2.2. SIG de Consciência de Software Software

O SIG representando consciência de software é apresentado na Figura 1. Começamos o SIG representando consciência de software, como uma meta flexível de NFR raiz e a refinamos através do método de decomposição por tipo nos aspectos discutidos anteriormente: consciência do contexto, consciência do tempo, consciência do autocomportamento e consciência das relações sociais.

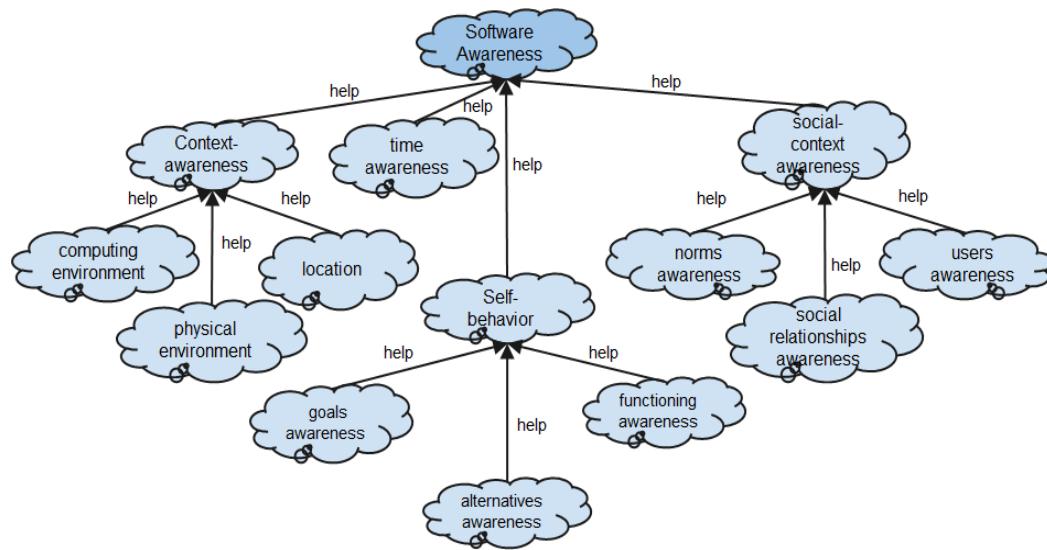


Figura 1 – SIG de Consciência de Software

A fim de identificar operacionalizações que são relevantes para consciência de software, nós usamos o método Goal-Question-Operacionalization (GQO), que é uma adaptação do método do Goal-Question-Metric - GQM (Basili, 1992), proposto em (Serrano e Leite, 2011). O método GQM ajuda a definição de um sistema de medição para as metas, definindo questões para avaliar a satisfação das metas e cria métricas que permitem que estas perguntas sejam respondidas quantitativamente. Na adaptação GQO do método, as métricas são substituídas por operacionalizações, que são formas conhecidas maneiras de responder positivamente a questão e, portanto, para a satisfação a contento das metas flexíveis. Perguntas são importantes para direcionar a busca por operacionalizações para uma meta flexível, enquanto operacionalizações podem ser encontradas na literatura de desenvolvimento de software como boas práticas ou derivadas da experiência do praticante. As perguntas são apresentadas no catálogo através de padrões de perguntas para requisitos não funcionais, enquanto as operacionalizações são apresentadas através de padrões de operacionalização de requisitos não funcionais.

O método geral para operacionalizar qualquer subtipo de consciência de software pode ser dividido em dois grandes passos integrados: (i) de aquisição de dados, e (ii) a interpretação de dados. Por aquisição de dados, queremos dizer uma função que retorna os valores de dados que representam a situação subjacente em um determinado instante no tempo.

Os instrumentos utilizados para a aquisição de dados variam de acordo com o subtipo de consciência de software, assim como os mecanismos

utilizados para interpretar estes dados variam de acordo com o tipo e a quantidade de dados adquiridos. Além disso, os dados que devem ser adquiridos variam de acordo com o domínio do problema, que é representado no NFR framework pelo tópico. Segundo o método GQO, algumas perguntas gerais a consciência software são: "Quais dados devem ser adquiridos?", "Como obter os dados desejados?" e "Como os dados adquiridos devem ser interpretados?". Como a primeira questão depende do domínio do problema, vamos nos concentrar em mecanismo para responder às duas últimas perguntas.

Apesar de não sugerir qualquer operacionalização para responder à pergunta "Quais dados devem ser adquiridos", sugerimos uma dica útil: como a informação necessária está relacionada com o problema a ser resolvido, os principais elementos a analisar, a fim de responder à pergunta são as metas e as alternativas de ação. Em geral, as informações solicitadas estão intrinsecamente relacionadas com a satisfação das metas do software, e têm impacto sobre a escolha das alternativas.

Para responder à pergunta "Como os dados adquiridos devem ser interpretados?", é necessário encontrar formas que permitam orientar o comportamento do software com base nos dados de contexto adquiridos. Vamos voltar a esta questão quando discutimos a operacionalização para a consciência do autocomportamento, e em mais detalhes no capítulo de modelagem.

Na seção seguinte iremos nos concentrar na operacionalização para a aquisição de dados de cada subtipo do requisito de consciência.

2.3. Operacionalização de consciência de contexto.

De acordo com Truong (Truong, 2009), "estar consciente do contexto permite ao software não só ser capaz de lidar com as mudanças no meio ambiente que o software opera, mas também ser capaz de melhorar a resposta para a utilização do software. Isso significa que técnicas de consciência de contexto (context-awareness) têm por objectivo apoiar os requisitos funcionais e não funcionais do software". Esta colocação não nos causa nenhuma surpresa, visto que a operacionalização que irá melhorar a satisfação a contento de requisitos não funcionais é muitas vezes obtida através de requisitos funcionais. Nesta seção, vamos focar em responder à pergunta "Como adquirir os dados desejados" para consciência de contexto, abordando cada subtipo separadamente. Por consciência de contexto, queremos dizer consciência do

contexto externo ao software no qual o mesmo está inserido, ou em outras palavras, o ambiente no qual o software está inserido.

2.3.1.

Operacionalização de consciência do ambiente físico

Como software tornou-se pervasivo, ser consciente do ambiente de computacional subjacente tornou-se essencial. Como uma visão conceitual, as técnicas para operacionalização de consciência de contexto compartilham um mesmo núcleo: sensoriamento (ou monitoração) do contexto. O desafio fundamental da operacionalização de consciência de contexto é como detectar e medir informações representativas do contexto subjacente e, como desafios derivados, como eliciar e modelar informações de contexto que precisam ser detectadas e medidas. Em ambientes físicos, as informações contextuais são normalmente determinadas por sensores de hardware ou software embarcados, como o GPS e programas de monitoramento, ou são fornecidas pelos usuários. Normalmente, os sensores dependem de protocolos de comunicação de baixo nível para enviar as informações de contexto coletadas ou eles estão fortemente acoplados dentro dos sistemas sensíveis ao contexto (Truong, 2009). Como os sistemas conscientes do contexto têm de lidar com dispositivos e serviços difusos, sensores nestes sistemas têm de suportar vários protocolos e interagir com sistemas de terceiros. Como resultado, a interface entre os sensores e outros componentes de apoio são diversas.

Retornando ao método GQO, refinamos a pergunta: "Quais dados devem ser adquiridos?" no caso de ambiente físico para "Quais dados devem ser detectados e medidos?". Como questões subsequentes temos "Que tipo de sensores ou instrumentos devem ser utilizados?" e "Como as informações fornecidas por sensores ou instrumentos utilizados devem ser acessadas?".

Para responder a estas questões, deve-se começar com foco na primeira pergunta, cuja resposta irá conduzir o esforço em busca de respostas para as perguntas seguintes. Uma vez definido que informação deve ser monitorada, sugerimos o uso de sensores adequados para obtê-la – sensores para ambiente físico são muito específicos.

A terceira questão, como acessar as informações fornecidas pelos sensores é atendida por protocolo de sensores – uma vez em que os sensores sejam escolhidos, o software precisa "estar a par" dos protocolos dos sensores, ou em outras palavras, o software precisa ter a capacidade de interagir com

sensores por meio dos protocolos específicos, para recuperar/acessar as informações desejadas.

Resumindo, sugerimos como relevantes para consciência do ambiente físico as seguintes questões: "Quais dados devem ser detectados e medidos?", "Que tipo de sensores ou instrumentos devem ser utilizados?" e "Como as informações fornecidas por sensores ou instrumentos utilizados devem ser acessadas?". A fim de responder a estas perguntas, sugerimos duas operacionalizações: monitoramento através de sensores e instrumentos e uso de protocolos de comunicação com sensores – os protocolos estão intimamente ligados aos sensores utilizados. O monitoramento através de sensores pode ser refinado por sensores específicos de acordo com a medida alvo desejada. Por exemplo, sensores de temperatura, áudio, campo magnético e orientação, peso, luz e sensores de visão, movimento e aceleração, bio-sensores, detecção de movimento, gás e narizes eletrônicos, proximidade, sensores, umidade e pressão do ar, toque e interação do usuário.

2.3.2.

Operacionalização de consciência do ambiente computacional

Em computação ubíqua, bem como em outros tipos de computação distribuída, esforços têm sido feitos para se construir middleware, proporcionando, na maioria dos casos, suporte ao intercâmbio de informações de contexto e interfaces para tarefas básicas no sistema subjacente. Assim, chegamos a uma primeira pergunta: "Algum middleware é usado?". Em cenários em que há um middleware, uma questão relevante é "Quais são os recursos disponíveis e como eles podem ser acessados?".

Considerando a informação como um recurso especial, surgem outras questões que também são relevantes: "Como as informações são trocadas?", ou em outras palavras, "Como a informação é modelada?"; "Como as informações são armazenadas?" e "Como a informação é acessada a partir de seu armazenamento?"; "Como as informações podem ser buscadas e recuperadas?"; "Como a informação é distribuída e propagada para diferentes dispositivos ou atores em sistemas distribuídos?".

Para abordar as questões relativas ao lidar com informação distribuída, algumas técnicas são usadas para descrever informações de contexto de uma forma aberta e interoperável, como por exemplo, XML (Extensible Markup Language, 2004), RDF (Resource Description Framework, 2006) e OWL (OWL

2, 2012). Abordagens como RDF e OWL facilitam o reuso de vocabulários comuns, o que é importante para interoperabilidade. Do ponto de vista de engenharia de software, abordagens como UML e orientadas a meta como i* (Yu, 1995) e KAOS (Van Lamsweerde, 2001), oferecem vantagens para engenharia baseada em modelos (Schmidt, D., 2006), enquanto XML é considerada aberta e interoperável.

Em termos de fornecimento de recursos básicos, diversos tipos de plataformas de infraestrutura ou middleware têm sido desenvolvidos com o objetivo de tornar mais fácil a realização de comunicação e entrada/saída de dados em sistemas de computação distribuída. Como plataformas de agentes de software, podemos destacar JASON (Bordini et al., 2005) como uma plataforma de desenvolvimento, JADE - Java Agent Development Platform (Bellifemine et al., 1999) e SACI - Simple Agent Communication Infrastructure (Hübner e Sichman, 2003) como middleware para prover recursos básicos como comunicação, localização (em ambiente de computacional) e serviços de páginas amarelas para registro e descoberta.

Na Figura 2, são apresentadas as seguintes questões relevantes que identificamos para consciência do ambiente computacional: "Algum middleware é usado?", "Quais são os recursos disponíveis?", "Como os recursos são acessados?", "Como a informação é modelada?", "Como as informações são armazenadas?", "Como as informações são recuperadas?" e "Como as informações são trocadas entre diferentes atores?".

Na Figura 3, sugerimos como operacionalizações de primeiro nível para consciência do ambiente computacional o uso de: "middleware para sistemas multiagentes", "modelos de recursos e informação" e "padrões para a troca de informações". Estas operacionalizações devem ser refinadas e sugerimos como middleware usar JADE ou SACI – ambos são middleware para sistemas multiagente (*Multi-Agent System – MAS*). Para operacionalizar "Modelo de recursos e informação", sugerimos utilizar uma linguagem para modelagem de software, ontologias ou XML. XML pode ser utilizada também para operacionalizar "Padrões para a troca de informações". Como linguagem para modelagem de software, sugerimos usar UML, i* (i-estrela) ou KAOS – sendo as duas últimas orientadas a metas, o que é mais adequando a modelagem de sistemas multiagente. Para ontologias sugerimos usar OWL ou RDF.

2.3.3.

Operacionalização de consciência de localização

Pode-se observar que consciência de contexto (context-awareness) e sistemas conscientes de contexto evoluíram a partir de consciência de localização (location-awareness) por generalização (Schmidt, 2002). Localização como contexto principal é muito bem compreendida (Leonhardt, 1998) e os dispositivos de aquisição de contexto estão disponíveis off-the-shelf, pelo menos para uso ao ar livre. Além disso, o valor de localização como contexto é óbvio. O valor de outras informações de contexto, especialmente sobre o meio ambiente, muitas vezes não é claro e medi-los muitas vezes requer hardware específico (Schmidt, 2002).

A questão sobre a localização é bastante óbvia, e em certa medida, é válido para qualquer domínio: "Onde está o usuário/dispositivo/software?". Para responder a essa pergunta, algumas tecnologias conhecidas como Sistema de Posicionamento têm sido desenvolvidas. De um modo geral, essas tecnologias poderiam ser classificadas em dois grupos de acordo com sua cobertura, variando de cobertura mundial, conhecido como sistema global, à cobertura de área de trabalho/intramuros, conhecida como sistema local.

Os sistemas globais são baseados em sistemas de navegação global por satélite (Global Navigation Satellite System – GNSS), que permitem que os receptores de rádio especializados possam determinar a sua posição no espaço 3D, bem como o tempo, com uma precisão de 2 a 20 metros ou dezenas de nanosegundos. Neste grupo podemos destacar o Sistema de Posicionamento Global (GPS) – sistema militar dos EUA, em pleno funcionamento desde 1995 e GLONASS – sistema militar russo, em pleno funcionamento desde outubro de 2011. Sistemas globais como GPS são precisos em áreas abertas, mas funcionam mal em ambientes fechados ou entre prédios altos (conhecido como o efeito canyon urbano).

Ao contrário dos sistemas globais, os sistemas de posicionamento local (Local Positioning System - LPS) não fornecem cobertura global. Em vez disso, eles usam (um conjunto de) âncoras, que têm um alcance limitado, exigindo, portanto, que o usuário esteja perto deles. Este tipo de tecnologia é complementar aos sistemas globais e trabalham melhor onde os sistemas globais falham: dentro e entre edifícios altos. Neste grupo podemos destacar Sistema de Posicionamento baseado em redes sem fio (Wi-Fi) e sistemas de localização em tempo real (RTLS), usados para identificar e rastrear a

localização de objetos ou pessoas em tempo real, geralmente dentro de um edifício ou outra área limitada. Marcadores (tags) para os RTLS sem fio são anexados a objetos ou usados por pessoas, e na maioria dos RTLS, pontos de referência fixos recebem sinais sem fio a partir dos marcadores para determinar a sua localização.

Outra tecnologia específica para telefones celulares é o posicionamento móvel, que inclui o serviço baseado em localização que divulga as coordenadas reais de um telefone celular via multilateral – técnica baseada na medição da diferença de distância de duas ou mais estações em locais conhecidos que transmitem sinais em tempos conhecidos – de sinais de rádio entre (várias) torres de rádio da rede e do telefone.

Para guiar a escolha da opção de operacionalização adequada do requisito de localização, adicionamos a pergunta: "onde o usuário/dispositivo/software estará?". Duas perguntas adicionais precisam ser respondidas: "qual a cobertura necessária, local ou global" e "o software irá operar em um ambiente interno ou em um lugar aberto?".

As operacionalizações para localização, sugeridas na Figura 3, utilizam um Sistema de Posicionamento Local ou um Sistema de Posicionamento Global. Enquanto sugerimos a utilização de um sistema de posicionamento baseado em Wi-Fi ou um sistema de localização em tempo real para operacionalizar sistemas de posicionamento locais, para operacionalizar sistemas globais sugerimos o uso de GPS.

Na Figura 2, apresentamos o que acreditamos ser um conjunto de questões relevantes sobre a consciência de contexto que podem ajudar a encontrar operacionalização a este requisito, soa a forma de padrão de pergunta para requisitos não funcionais - *NFR Question Pattern* (Serrano e Leite, 2011). Na Figura 3, Figura 4 e Figura 5, apresentamos algumas alternativas de operacionalização sugeridas para ambiente físico, ambiente de computacional e localização, respectivamente, com base no que discutimos anteriormente nesta seção.

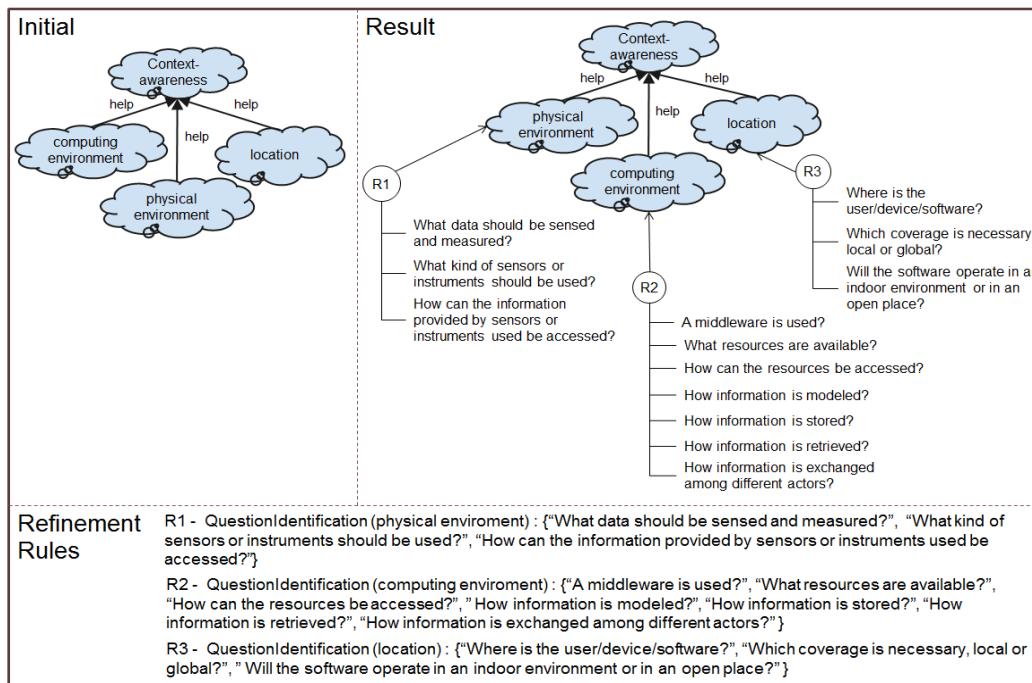


Figura 2 - Padrão Questão (Question Pattern) para consciência de contexto

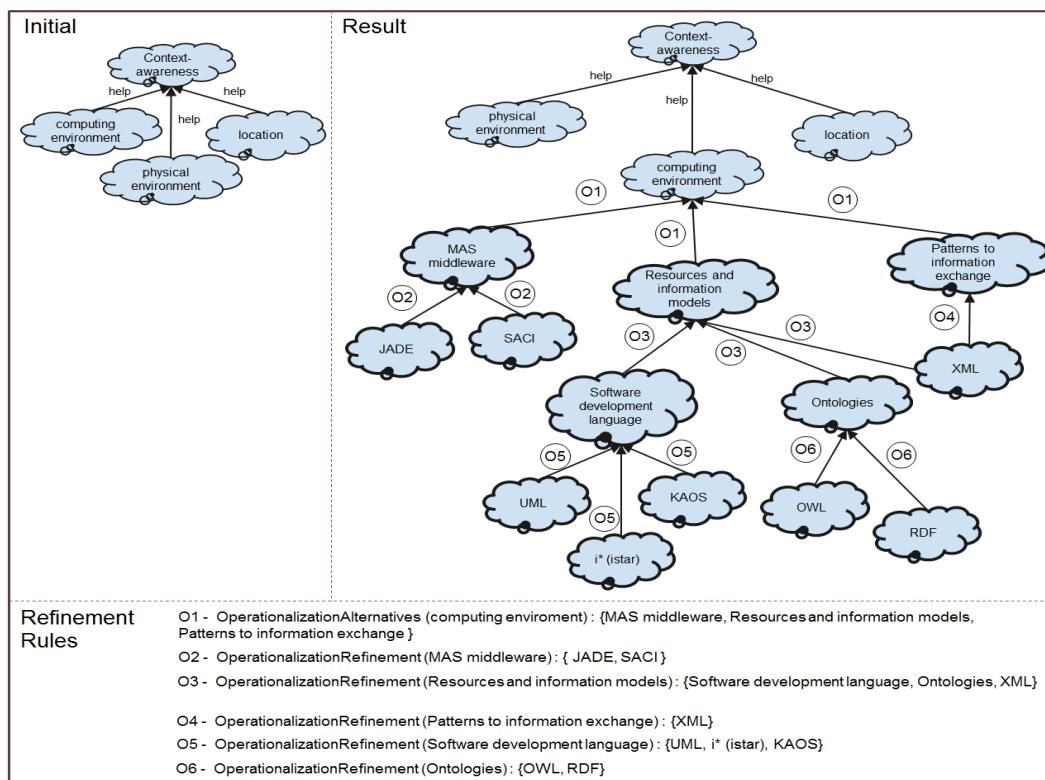


Figura 3 – Operacionalização – ambiente computacional

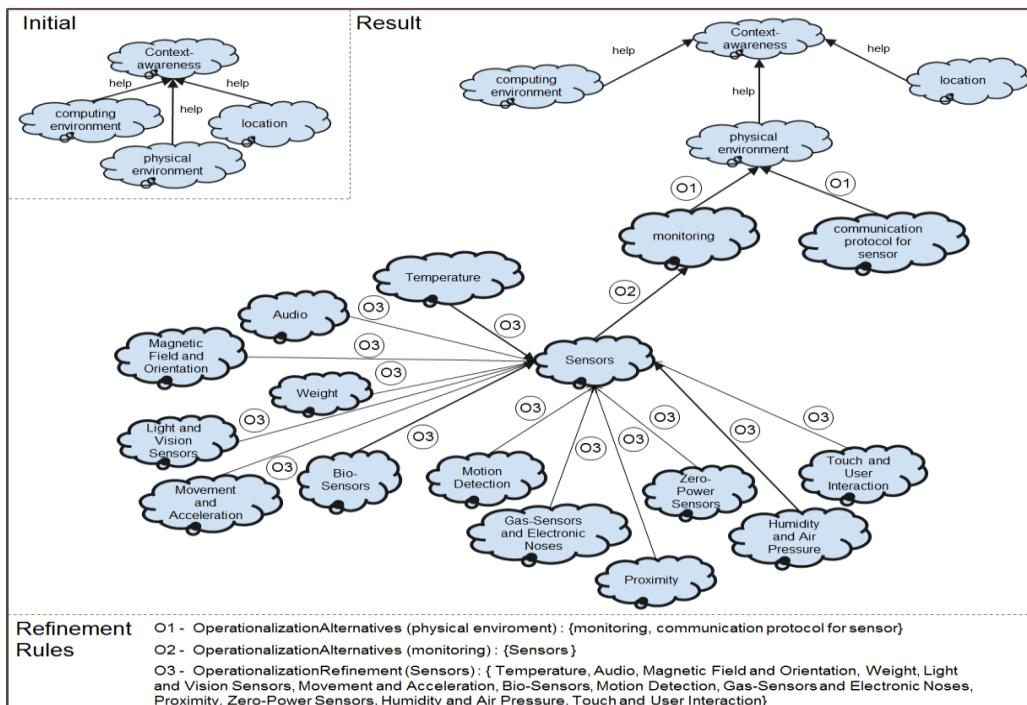


Figura 4 - Operacionalizações - ambiente físico

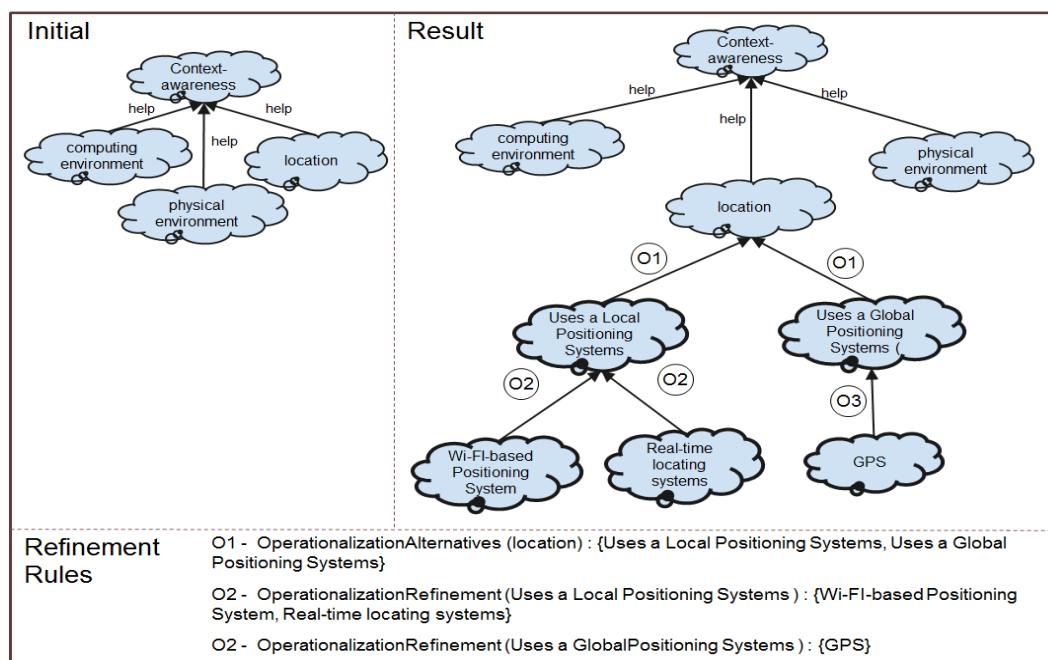


Figura 5 - Operacionalizações - localização

2.4.

Operacionalização de consciência do tempo

Consciência do tempo é provavelmente o subtipo cuja operacionalização se dá forma mais direta. A questão relevante à consciência do tempo é, da mesma forma, direta: o software precisa ter consciência de “qual é o instante em que está operando?”. Adicionalmente, pode ser relevante também saber o instante corrente com alto nível de acurácia.

Em resposta a estas questões e como forma de operacionalizar a consciência do tempo, sugerimos o uso de servidores de tempo. Por exemplo, os serviços oferecidos pelo o Worldtimeserver.com (World Time Server, 2014).

2.5.

Operacionalização de consciência do autocomportamento

O desafio em fazer software ser consciente de seu próprio comportamento é, possivelmente, o mais difícil. Para enfrentar este desafio nós examinamos a especialização de autocomportamento em consciência de metas, consciência de alternativas e consciência do autofuncionamento apresentada anteriormente no SIG do requisito de consciência. A primeira pergunta que surge é: "Quais são as metas que o software precisa alcançar?". As respostas para essa pergunta dependem do domínio do problema.

Uma vez que as metas tenham sido identificadas, outra questão relevante é "Quais são as alternativas possíveis para alcançar essas metas?" e "Qual é o impacto da escolha de cada alternativa?". Estas perguntas estão no domínio do problema e devem ser tratadas a partir da fase de elicitação da engenharia de requisitos. Uma vez respondidas essas perguntas, outras questões surgem: "Como alocar essas metas (tanto as rígidas quanto as flexíveis), com as respectivas alternativas, no software?".

Uma vez que metas e alternativas tenham sido alocadas no software, surgem as questões: "Como o contexto (externo) influencia o comportamento do software na satisfação de suas metas?", "Como a influência do contexto na satisfação das metas pode ser modelada?", "Como avaliar as alternativas existentes de acordo com o contexto específico percebido pelo software?", "Como o software pode perceber seu próprio comportamento sob a influência do contexto percebido?", "O software tem conseguido satisfazer suas metas?" e "Quão eficaz o software tem sido na satisfação de suas metas?".

Para abordar as questões relativas à modelagem e alocação de metas e alternativas, recomendamos o uso de uma abordagem intencional com base na engenharia de requisitos orientada a meta (*Goal-Oriented Requirements Engineering - GORE*) por seu poder em representar explicitamente o comportamento adaptativo do software. A pesquisa em Engenharia de Requisitos (ER) tem, cada vez mais, reconhecido o papel de liderança desempenhado pelas metas (*goals*) no processo de Engenharia de Requisitos (Van Lamsweerde, 2001).

Muitas abordagens diferentes que seguem os princípios da GORE tem sido propostas. KAOS, que significa a aquisição de conhecimento em especificação automatizado (*Knowledge Acquisition in automated Specification*) ou, em outras palavras, mantenha todas as metas satisfeitas (*Keep All Objectives Satisfied*), fornece um modelo conceitual, uma linguagem associada e um conjunto de estratégias para aquisição de requisitos baseados em metas.

i^* (i-estrela), que representa a intencionalidade distribuída, introduziu aspectos da modelagem social e do raciocínio, tais como autonomia dos atores e intencionalidade, na Engenharia de Requisitos. Posteriormente, a metodologia Tropos (Bresciani et al., 2004) adotou i^* e levou seus conceitos além dos estágios requisitos iniciais (*early requirements*) para todo o processo de desenvolvimento de software.

Para responder as perguntas "Como o contexto (externo) influencia o comportamento do software na satisfação de suas metas?" e "Como o software pode perceber seu próprio comportamento sob a influência do contexto percebido?" é preciso detalhar como o software poderia perceber o contexto. Dissemos anteriormente que o método geral para operacionalizar qualquer subtipo de consciência pode ser dividido em duas grandes etapas integradas: (i) aquisição de dados e (ii) interpretação de dados. O objetivo principal da etapa de interpretação de dados é permitir que o software possa perceber a situação subjacente no mundo real e avaliar a que contexto essa situação pertence, para então (re)agir de acordo. Assim, é necessária a construção de uma "ponte" entre os dados de contexto adquiridos e as alternativas para a satisfação das metas, uma vez que as alternativas representam as possíveis ações que o software poderia tomar (ou os pontos de variabilidade no comportamento do software). Uma abordagem interessante é definir um conjunto de situações no mundo real, que pode ser capturado através de aquisição de dados, e para a qual software poderia decidir como (re)agir. Em outras palavras, as situações sobre as quais o

software poderia “raciocinar” sobre as alternativas para satisfazer suas metas atuais.

Em relação à forma de verificar a eficácia da satisfação das metas, esta questão é mais relevante em problemas de prognóstico – problemas relacionados a prever o que vai acontecer no futuro com base no contexto atual subjacente, ou seja, prever quais serão os próximos eventos, e agir em conformidade com essa previsão. A solução para este tipo de problema em geral é probabilística, com um grau de certeza associado, onde dificilmente se tem certeza absoluta de qual será o próximo evento. Em muitos casos, a ocorrência ou não do evento previsto pode ser monitorada/verificada pelo software no momento adequado – através de variáveis similares as que o software monitora para perceber o contexto atual. Por exemplo, nos casos de previsão de clima e tempo – o software pode verificar, por meio de sensores de temperatura e precipitação, o grau de precisão de sua previsão da ocorrência de um evento específico, quando chegar a hora. Esta verificação pode ser utilizada para melhorar o funcionamento do software em relação à chegada de novos eventos, configurando um processo de *feedback* interno.

Existem outras classes de problemas que o software não pode verificar por si mesmo se eles foram ou não bem-sucedidos na satisfação de suas metas. Na maioria dos problemas de recomendação, um subconjunto de problemas de classificação nos quais o software classifica algumas entidades, representadas por um conjunto de valores de variáveis, em classes de interesse dos usuários específicas. Nesse tipo de problema, o software precisa ser informado pelos usuários, se a classificação foi satisfatória ou não. Ou seja, o software precisa de informação externa, configurando um processo de *feedback* externo. Uma vez que o software obtenha o *feedback* dos usuários, estes podem ser utilizados para melhorar o funcionamento do software. Então, em relação à consciência do autofuncionamento o software pode perceber por si mesmo como ele tem funcionado (*feedback* interno) ou o software pode ser informado, geralmente por usuários, do seu funcionamento (*feedback* externo). Ambos os feedbacks, internos e externos, podem ser utilizados para melhorar o processo de como o software cria e interpreta suas percepções com base no contexto. É importante que software autoadaptativos sejam conscientes de seu próprio funcionamento.

A complexidade em lidar com a consciência do autocomportamento é especialmente difícil em problemas probabilísticos porque, em muitos casos, demanda do software uma percepção do contexto e de como o comportamento do software é afetado por mudanças no contexto, em tempo de execução (*run-*

time) – os desenvolvedores não podem saber em tempo de desenho (*design-time*) como cada situação real será percebida. Uma possível abordagem para lidar com esse problema é fazer com que as metas, as alternativas e as informações de contexto sejam avaliadas em tempo de execução. Neste sentido, podemos destacar alguns trabalhos que têm sido feitos em Tropos, com foco na auto* (*self-**) em tempo de execução, tais como (Dalpiaz et al., 2009; Dalpiaz et al., 2010), e trabalhos com requisitos em tempo de execução (*requirements at run-time*), tais como (Souza e Mylopoulos, 2011; Inverardi e Mori, 2011; Qureshi et al., 2011).

Na Figura 6 apresentamos, sob a forma de padrão de pergunta para requisitos não funcionais, o que acreditamos ser um conjunto de questões relevantes sobre consciência do autocomportamento que podem ajudar a encontrar operacionalizações para este requisito. Para consciência de metas, sugerimos as seguintes perguntas: "Quais metas o software deve satisfazer?", "Como alocar as metas (e as metas flexíveis) no software?", "Como a influência do contexto pode ser modelada?" e "Como a satisfação das metas é influenciada pelo contexto?". Estas questões podem ser abordadas através da utilização de um modelo orientado a meta, que por sua vez, podem ser operacionalizados usando-se KAOS e i* como alternativas para modelagem orientada a meta, como mostrado na Figura 7.

Para operacionalizar o requisito de consciência de alternativas, nós elicitamos como questões relevantes "Quais são as alternativas possíveis para a satisfação das metas (e das metas-flexíveis)?", "Como alocar as alternativas no software?", "Quais são os impactos da escolha de cada alternativa?", "Como avaliar as alternativas existentes de acordo com o contexto subjacente?". Para operacionalizar estas questões, sugerimos a utilização de modelos orientados a meta para fazer uma modelagem do contexto e avaliar as alternativas sob o impacto do contexto. A modelagem de contexto pode ser refinada em três operacionalizações mais específicas: descrição do contexto, definição de um conjunto de situações de contexto e definição de um mecanismo de identificação da situação, que, por sua vez, pode ser refinado pelo uso de uma função probabilística ou pelo uso de uma função determinística. Para operacionalizar consciência do autofuncionamento, nós sugerimos usar uma estratégia de "requisitos em tempo de execução" e monitoração da satisfação das metas. Esta última pode ser refinada por utilização de um mecanismo de feedback interno ou de um mecanismo de feedback externo.

Na Figura 7 apresentamos, na forma de padrão operacionalização de requisitos não funcionais, algumas sugestões de alternativas de operacionalização com base no que discutimos nesta seção.

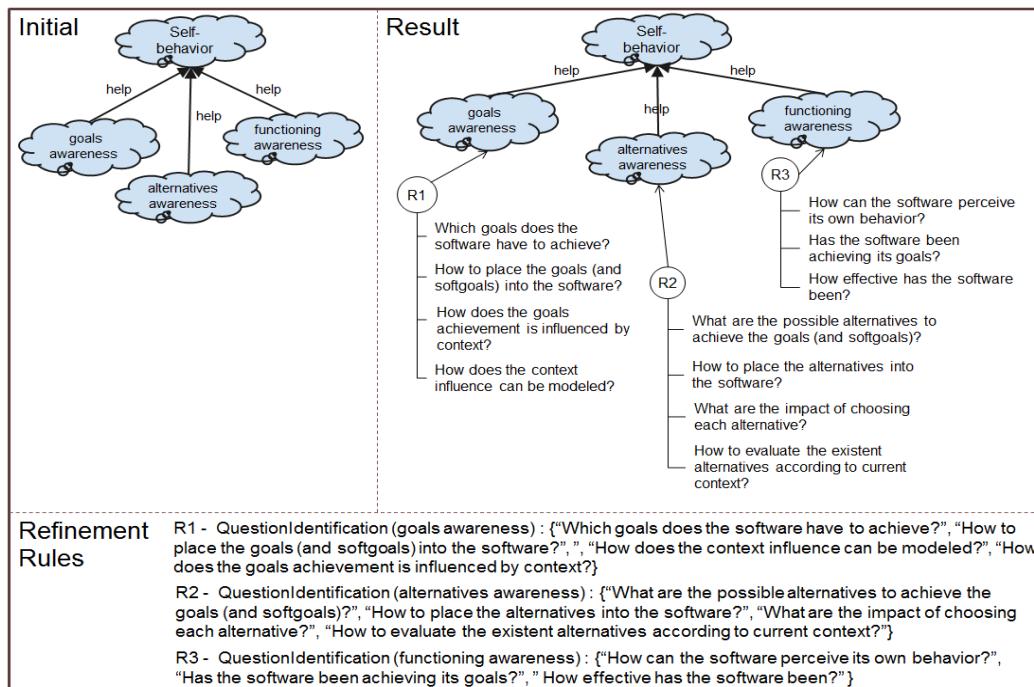


Figura 6 - Padrão questão - consciência de autocomportamento

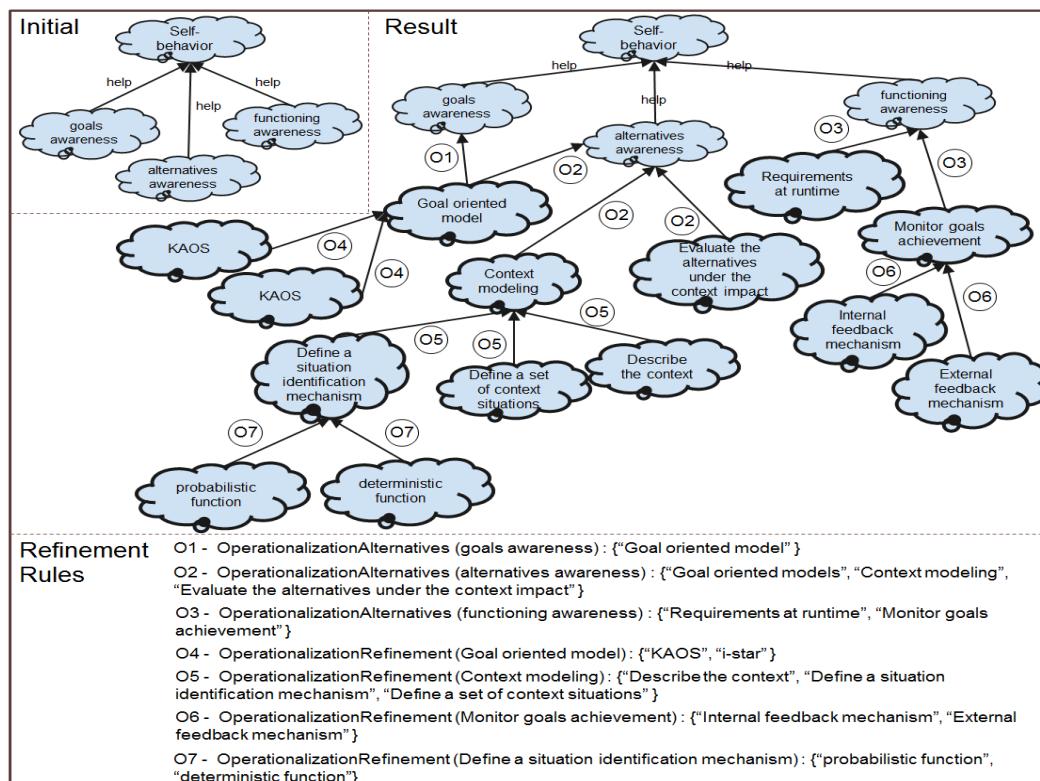


Figura 7 - Operacionalizações - consciência de autocomportamento

2.6.

Operacionalização de consciência do contexto social

Destacamos na consciência de contexto social os aspectos sociais que o software precisa ser consciente a fim de adaptar-se e satisfazer suas metas, possivelmente em colaboração com outros atores no contexto social. Fizemos isso porque as preocupações envolvidas são bastante diferentes das presentes na consciência do ambiente computacional e na consciência do autocomportamento. Para o sucesso da autoadaptação, é importante para o software perceber aspectos relacionados aos usuários do software, tais como a identidade e as preferências dos usuários, aspectos relacionados a outros agentes de software no mesmo ambiente, especialmente em ambientes abertos, e entender as regras que regulam os comportamentos dos atores no ambiente. Nós agrupamos sob o requisito de consciência do contexto social um conjunto de tipos especiais de consciência que podem permitir ao software perceber o contexto social em que ele vai operar, embora de forma limitada. Nós especializamos consciência do contexto social em três subtipos: consciência de normas, consciência de relacionamento social e consciência de usuário.

2.6.1.

Operacionalização de consciência de normas

Normas em sistemas multiagente podem ser usadas para especificar os padrões de comportamento que os agentes devem seguir para alcançar os objetivos globais do sistema (Dastani et al., 2009). Ao executar em um ambiente regulado por normas, as primeiras questões que surgem são: "Quais são as normas que o agente tem deve seguir?" e "Como essas normas afetam o comportamento do agente?". Mais uma vez, as respostas a estas perguntas dependem do domínio do problema. Uma vez as respostas tenham sido obtidas, outras questões que surgem são: "Qual o tipo das normas que agente deve seguir?" e "Quais são as penalidades no caso de não cumprimento da norma?".

Em uma abordagem exógena, onde atores externos ao agente observam e avaliam os comportamentos dos agentes a fim de verificar o cumprimento ou violação das normas, existem basicamente dois mecanismos: de arregimentação (*regimentation*) ou de coerção (*enforcement*). Em um mecanismo de arregimentação, ações externas de todos os agentes que conduzam a uma violação das normas são impedidas – o sistema impede que um agente execute uma ação proibida, o que diminui a autonomia do agente.

Um mecanismo de coerção, por sua vez, é baseado na ideia de responder depois que uma violação das normas ocorra. Tal resposta, que inclui sanções, tem como objetivo retornar o sistema a um estado aceitável/ideal. Crucial para mecanismos de coerção é que as ações que violam as normas possam ser observadas pelo sistema. Uma vantagem de usar mecanismos em vez de arregimentação é que permitir violações contribui para a flexibilidade e autonomia do comportamento dos agentes (Castelfranchi, 2004). Estas normas são muitas vezes especificadas por meio de conceitos como permissões, obrigações e proibições.

Se um agente tem que realizar ações em um sistema com mecanismos de arregimentação, onde não é possível que o agente viole as normas, essas normas podem ser colocadas, em tempo de desenho, nos comportamentos do agente como restrições que devem ser observadas.

Por outro lado, se um agente tem que executar ações em um sistema com mecanismos de coerção, na escolha entre diferentes alternativas de ações, o agente precisa estar consciente das possíveis normas que se aplicam a cada ação, com as respectivas sanções, para avaliar corretamente o impacto de suas escolhas. Em alguns casos, cumprir uma norma poderia trazer um obstáculo à

satisfação das metas do ou levar um estado de insatisfação (deny) de suas metas flexíveis. Quando essa situação ocorre, pode ser melhor para o agente decidir não cumprir a norma e receber as sanções aplicáveis.

Quando as normas são implementadas de forma endógena, integrando-as no programa dos agentes individuais, elas podem ser vistas como limitações internalizadas pelo comportamento do agente.

Então, outras questões relevantes para operacionalização de consciência de normas são: "As normas são implementadas de forma exógena ou endógena aos agentes?". Se for usada uma abordagem endógena, "Que tipo de mecanismo é usado: arregimentação ou coerção?". Em um mecanismo de coerção: "Quais sanções são aplicáveis a cada violação?".

Para operacionalizar normas, foram propostas estruturas para as organizações de agentes que englobam as normas, como a OMNI (Dignum et al., 2005), e as propostas por Dybalova et al. (Dybalova et al., 2013) e por Modgil et al. (Modgil et al., 2009).

Para operacionalizar o requisito de consciência de normas, que suscitou questões relevantes como "Quais são as normas que o agente deve seguir?", "Como essas normas afetam o comportamento do agente?", "Qual o tipo das normas que o agente deve seguir? Arregimentação ou coerção?" e "Quais sanções são aplicáveis caso a norma seja violada?". Como operacionalização do requisito de consciência de normas sugerimos duas atividades: modelagem de normas e implementação de normas. A modelagem de normas pode ser refinada em modelar as restrições e modelar as sanções, enquanto a implementação de normas são refinadas em duas formas diferentes: implementação endógena e implementação exógena. A abordagem exógena por sua vez pode ser implementada por meio de mecanismo de coerção ou mecanismo de arregimentação. Finalmente, para implementar as duas abordagens, sugerimos a utilização de um framework de norma.

Na Figura 8 apresentamos, na forma de padrão de questão de requisitos não funcionais, o que acreditamos ser um conjunto de questões relevantes sobre a consciência do contexto social que podem ajudar a encontrar operacionalizações para este requisito. A Figura 9 apresenta as operacionalizações sugeridas para consciência de norma.

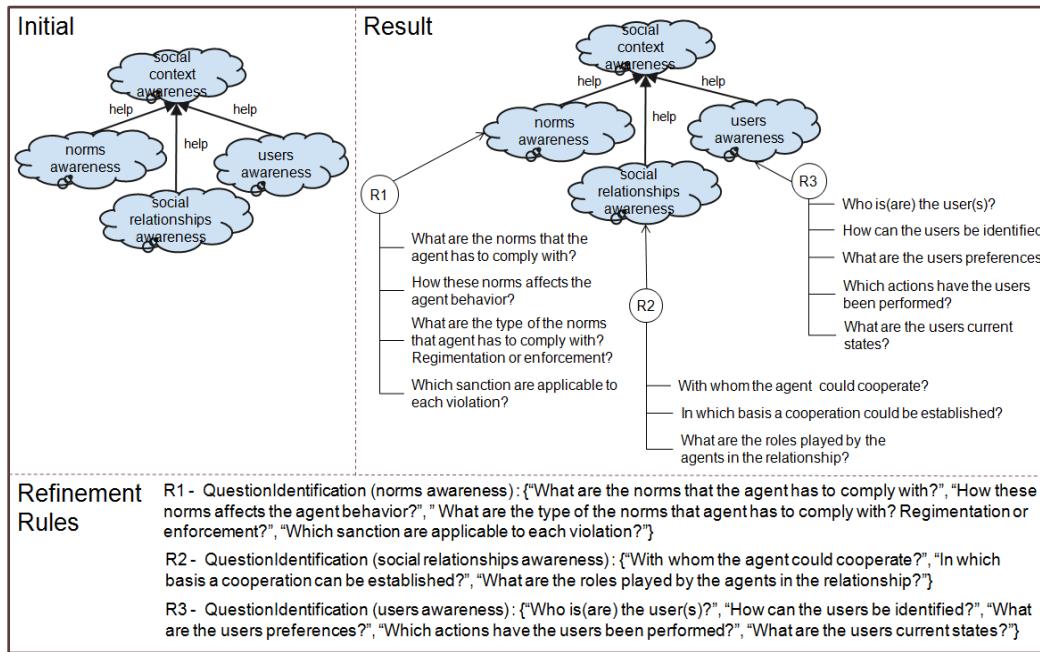


Figura 8 - Padrão questões - consciência do contexto social

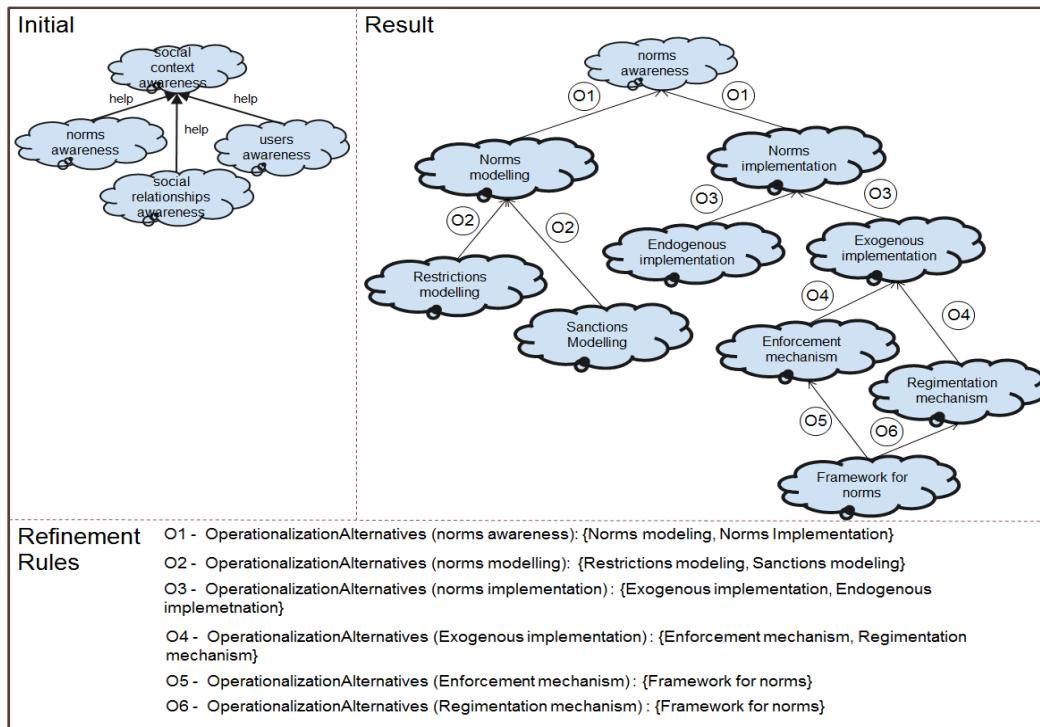


Figura 9 - Operacionalizações - consciencia de norma

2.6.2.

Operacionalização de consciência das relações sociais

As relações sociais entre os agentes têm uma importância considerável para a consciência de software uma vez que os agentes podem adquirir conhecimento por meio de informações externas. Além disso, em sistemas multiagente (MAS) os agentes normalmente cooperam uns com os outros para satisfazer suas metas. Ao cooperar com outro agente, um agente pode adaptar o seu comportamento como consequência dessa cooperação. As primeiras questões que surgem a partir dessas relações sociais são: "Com quem o agente pode cooperar?" e "Em que bases esta cooperação é estabelecida?".

Alguns middleware para MAS, como JADE e SACI, oferecem um serviço de páginas amarelas para permitir aos agentes conhecer uns aos outros. Os agentes podem registrar seus serviços nas páginas amarelas e consultar as páginas amarelas para descobrir quais serviços são oferecidos por outros agentes. Isto pode ajudar a responder à pergunta "Com quem o agente pode cooperar?".

A pergunta sobre as bases para estabelecer a cooperação é mais difícil de resolver. Ela pode ser abordada usando um mecanismo de compromisso. Chopra e Singh propõem que um compromisso pode ser descrito por quatro elementos: devedor, credor, antecedentes e consequentes em uma expressão C (*devedor, credor, antecedente, consequente*) o que significa que o devedor compromete-se com o credor a garantir o consequente se o antecedente for respeitado (Chopra e Singh, 2011).

Do ponto de vista da elicitação de requisitos, uma relação de negócios entre dois atores em um modelo de desenho representa uma intenção que esses dois atores interajam um com o outro para chegar a um compromisso. Nesta perspectiva, em um modelo de requisitos de alto nível de abstração, as relações representam uma espécie de "carta de intenções", que devem ser refinadas em modelos de requisitos de nível de abstração mais baixo, onde um modelo de contrato de alto nível seja projetado. Os modelos de dependência estratégica de i* (modelos SD) podem ser usados para modelar as dependências estratégicas entre os atores (ou agentes) envolvidos. Em uma relação de dependência em i*, um ator que depende de outro ator para ter suas necessidades atendidas. Essas necessidades são representadas pelos elementos de dependência sobre os quais os atores têm a intenção de colaborar, uma vez que um compromisso entre eles seja estabelecido.

Embora os modelos intencionais possam ajudar a extrair e projetar a intenção de cooperação entre os dois agentes, a partir da perspectiva de tempo de execução é essencial considerar a autonomia do agente. Devido a isso, o compromisso não pode ser assegurado em tempo de desenho, apenas em tempo de execução. Mesmo assim, um agente deve ter a possibilidade de negar um pedido de outro agente – nenhum agente deve ser forçado a aceitar um pedido a priori. Por isso, é extremamente importante ter uma estratégia orientada para o compromisso em tempo de execução, respeitando a autonomia dos agentes, ao invés de assumir que os agentes sempre irão colaborar uns com os outros como suposto nos modelos de desenho de alto nível. Para chegar a compromissos em tempo de execução, podem ser usados protocolos de compromisso, como proposto em (Chopra et al., 2010).

Outra abordagem relevante para as relações sociais entre os agentes de software é o uso de papel de agente (Kendall, 2001; Cabri et al., 2004). Papéis são usados principalmente para definir interações comuns entre os agentes (por exemplo, as interações entre leiloeiros e licitantes no leilão), e promover uma visão organizacional do sistema, o que se adequa bem a abordagens orientadas a agente (Zambonelli et al., 2001). Papéis incorporam todas as informações e recursos necessários, em um ambiente de execução específico, para se comunicar, coordenar e colaborar com outras entidades ou agentes. Graças a isso, um agente (e seu programador) não precisa conhecer os detalhes sobre o ambiente de execução atual, mas apenas qual o papel deve assumir e usar para interagir com (ou tirar proveito) o próprio ambiente (Cabri et al., 2004).

Em i*, um papel é uma caracterização abstrata do comportamento de um agente dentro de algum contexto ou domínio especializado. Suas características são facilmente transferíveis para os agentes. Dependências são associadas a um papel quando estas dependências são aplicáveis independentemente de quem desempenha o papel. Esta abstração aborda um aspecto diferente em uma relação social: o conjunto de comportamentos, representado pelo papel, que um agente pode assumir.

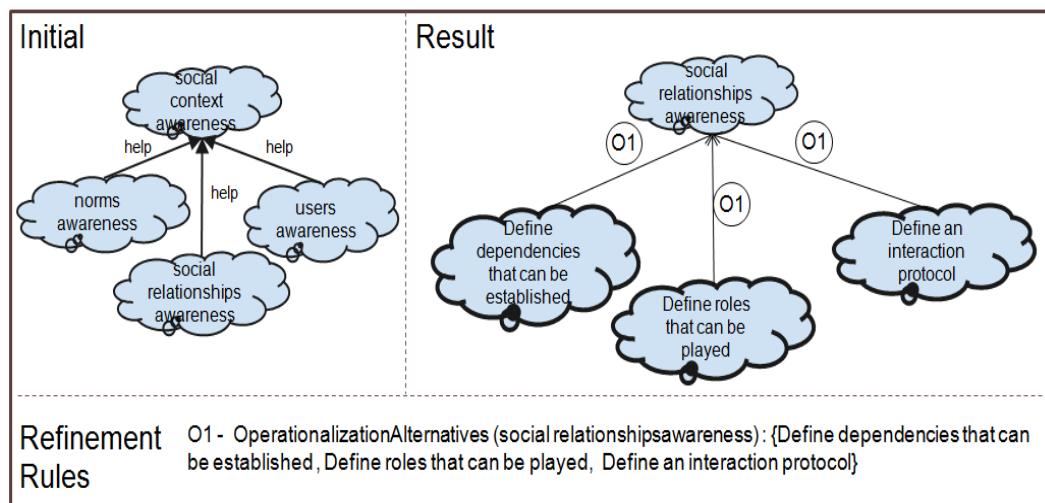


Figura 10 - Operacionalizações - consciência de relações sociais

Resumindo, para ter consciência das relações sociais que podem ser estabelecidas entre os agentes, algumas questões relevantes são "Com quem o agente poderia cooperar?", "Em que bases a cooperação pode ser estabelecida?" e "Quais são os papéis desempenhados pelos agentes no relacionamento?". Para operacionalizar estas questões, sugerimos definir as dependências que podem ser estabelecidas, definir os papéis que podem ser desempenhados e definir um protocolo de interação, como mostrado na Figura 10.

2.6.3. Operacionalização de consciência de usuários

É cada vez mais comum para software a necessidade de perceber seus usuários e adaptar-se a eles em alguma medida, seja a respeito de suas preferências ou da sua situação atual. A primeira pergunta que surge relacionada aos usuários é: "Quem é o usuário?". Esta questão é abordada através da determinação da identidade do usuário. Embora tenhamos nos referido a esta questão como identificação do usuário, em muitos casos, a questão abrange a identificação e a autenticação do usuário (para ter certeza de que o usuário é quem ele alega ser). A maioria dos métodos usados em software para identificar os usuários são baseados em informações anteriores armazenadas sobre os usuários que servem como um identificador único (id). Neste sentido, um usuário pode ser identificado somente se o sistema tem algum "conhecimento" prévio sobre ele. A maioria dos métodos usados hoje em dia para identificar e autenticar os usuários podem ser agrupados em pedir aos

usuários para fornecer uma identificação e uma senha, para apresentar um dispositivo pessoal que carrega a sua identificação (como, por exemplo, um *smartcard*), ou são baseados em autenticação biométrica (Biometrics: Overview, 2007). Mais de um método pode ser utilizado, simultaneamente, para melhorar o nível de segurança. Métodos com senha parecem ser, de longe, os mais utilizados – é difícil encontrar uma pessoa que não tenha que memorizar pelo menos uma senha pessoal hoje em dia. Políticas têm sido definidas com o objetivo de fazer com que a quebra de senha seja uma tarefa cada vez mais difícil. Por exemplo, garantir que a senha tenha um número mínimo de caracteres, impor o uso de caracteres alfabético e numérico, impor a utilização letras maiúsculas e minúsculas. Este tipo de política dificultar a quebra de senhas com o uso de glossários de senha. Além disso, as senhas devem ser encriptadas para serem transportadas e armazenadas.

O reconhecimento biométrico é baseado em duas premissas fundamentais sobre as características do corpo: distinção e permanência. A precisão e aplicabilidade da identificação por uma característica biométrica específica depende, essencialmente, de em que medida essas duas premissas são verdadeiras para a população a ser identificada (Jain e Kumar, 2010). As impressões digitais, face e íris estão entre as características fisiológicas mais populares usadas em sistemas biométricos comerciais, com impressão digital sendo usada em mais de 50% das aplicações do mercado civil (Biometrics Market Intelligence). Características comportamentais (como a assinatura, a marcha e a dinâmica de digitação) têm um caráter distintivo e permanência fracos e, por isso, poucas aplicações operacionais baseados nessas características foram implantadas até o momento.

A escolha de um mecanismo biométrico específico depende da natureza e dos requisitos da aplicação de identificação. Identificação por voz é mais apropriada em autenticação de aplicações que envolvem telefones móveis, uma vez que um sensor para capturar a voz (microfone) já está incorporado ao telefone. Impressão digital é a medida biométrica mais popular para acesso a laptops, telefones celulares e PDAs devido ao baixo custo dos pequenos sensores de varredura de impressão digital que pode ser facilmente incorporado nestes dispositivos, de acordo com (Jain e Kumar, 2010).

Dispositivos pessoais podem ser utilizados para armazenar informação, tais como identificação e perfil dos usuários. Em geral, esse mecanismo é usado em combinação com senhas para melhorar a segurança da autenticação (a lógica é que, se uma senha de usuário é quebrada, o impostor não teria acesso

indevido ao sistema porque a apresentação do dispositivo físico também seria necessária).

Uma vez que o usuário tenha sido identificado, alguns aspectos comportamentais sobre o usuário precisam ser conhecidos pelo software, tais como: as suas preferências, as ações que eles têm realizado e os estados atuais dessas ações.

As preferências de usuários podem ser informadas ou inferidas. Nos primeiros casos, o software precisa manter as preferências juntamente com os perfis dos usuários, a fim de se adaptar a essa preferências. Estas informações de preferências do usuário podem ser usadas para escolher alternativas nos pontos de variabilidade software em que estas preferências se apliquem.

Nos casos em que as preferências dos usuários precisem ser inferidas, isso pode ser feito com base em informações sobre os hábitos dos usuários. Por exemplo, o Facebook infere que mensagens um usuário prefere ler com base nos amigos que este usuário interage mais em sua rede. Preferências também podem ser inferidas com base nos hábitos de um grupo de usuários, em vez de um único usuário. Por exemplo, a Amazon informa ao usuário interessado em um livro específico, que outros livros o usuário poderia se interessar em comprar com base em que outros livros os usuários (que compraram o mesmo livro específico) compraram.

Em alguns casos, as preferências com base em hábitos passados de um grande grupo de usuário não pode ser obtida simplesmente olhando para o que outros usuários fizeram na mesma situação. O Prêmio Netflix (Bennett e Lanning, 2007) é um exemplo. A Netflix incentiva os assinantes a avaliar os filmes que eles assistirem, expressando uma opinião sobre o quanto eles gostaram, ou não, de um determinado filme. Em 2007, a Netflix estava recebendo mais de 2 milhões de avaliações de diárias. O sistema de recomendação da Netflix, Cinematch, analisa as classificações de filmes acumuladas e as usa para fazer várias centenas de milhões de recomendações personalizadas aos assinantes por dia, cada uma com base em seus gostos particulares. Em outubro de 2006 a Netflix disponibilizou um conjunto de dados contendo 100 milhões de avaliações anônimas de filmes e lançou um desafio às comunidades de mineração de dados, aprendizado de máquina para desenvolver sistemas que pudessem melhorar em 10% a precisão do Cinematch – a precisão do sistema é determinada pelo cálculo do erro médio quadrático de uma recomendação do sistema contra a avaliação real que um assinante provê.

Outras questões importantes para software ter consciência são as ações dos usuários. Com base na informação observada derivada de ações do usuário, padrões de comportamento podem ser detectados ou inferidos. Alguns sistemas de prevenção de fraude identificam comportamentos suspeitos com base em ações realizadas por seus usuários (Fawcett e Provost, 1997). Por exemplo, o banco comercial que sou cliente tem um sistema de prevenção da fraude que bloqueado o meu cartão de conta bancária porque eu fiz três operações diferentes, por diferentes canais, em lugares diferentes, no mesmo dia: eu havia comprado dólares norte-americanos no caixa, havia retirado dinheiro em um terminal eletrônico e havia pagado uma compra on-line através de débito em conta. O sistema de prevenção de fraudes considerou essas três ações diferentes em um mesmo dia como um padrão de comportamento que indica uma possível fraude e bloqueou o meu cartão. Após o bloqueio, eu fui informado que o banco estava fazendo isso porque o sistema de prevenção de fraudes havia detectado uma fraude em potencial.

Os estados dos usuários são uma questão crucial para a conscientização do usuário de software. Estados físicos dos usuários, assim como em consciência de ambiente físico, podem ser adquiridos por meio de sensores que monitoram os estados usuários. Aparte à localização, discutido anteriormente, os estados de usuário que o software precisa ser consciente dependem do domínio do problema. Alguns software, tais como sistemas de assistência médica, monitoram sinais vitais dos usuários e eventualmente disparam alertas quando estes sinais indicam um risco iminente ou real para o paciente. Outro exemplo é a detecção de mecanismos de sonolência em condutores de veículos. Isso pode ser feito usando um dispositivo sobre a orelha – um dispositivo de plástico leve, com um braço que desliza sobre uma orelha, como alguns fone-de-ouvido para telefone. Uma vez ligado, um sensor dentro da caixa mede o ângulo de uma perspectiva perpendicular. Se o motorista está olhando para a frente – como ele ou ela deveria – o alarme mede o ângulo de zero graus (howstuffworks). Quando uma pessoa está para adormecer, sua cabeça tende a cair para a frente quando a pessoa cochila. E a pessoa pode cair no sono por alguns segundos ou alguns minutos antes que solavancos ponham a cabeça em pé deixando a pessoa acordada novamente. Os sensores são capazes de detectar esta inclinação de cabeça para frente por um tempo que indique que a pessoa está caindo no sono e disparar um alarme para que a pessoa acorde. Outro tipo de dispositivo com o mesmo objetivo está baseado na comparação da taxa de fechamento dos olhos (PERCLOS) com a frequência vertical movimento

dos olhos entre os estados de vigília e sonolentos de motoristas que estavam dirigindo um carro em um simulador de condução (Bergasa et al., 2006).

Com base no que discutimos nesta seção sugerimos como relevantes para a consciência de usuários, as seguintes questões: "Quem é o usuário?", "Como os usuários podem ser identificados?", "Quais são as preferências dos usuários?", "Que ações os usuários têm realizado?" e "Quais são os estados atuais dos usuários?".

Como operacionalizações a essas perguntas sugerimos identificar os usuários, perceber (ser consciente de) as preferências dos usuários e perceber as ações dos usuários. Para identificação de usuários sugerimos como alternativas o uso de autenticação por dispositivo pessoal, autenticação biométrica e autenticação por senha. A autenticação biométrica pode ser operacionalizada por face, impressão palmar, íris, voz, impressão digital, DNA, geometria da mão ou veias da mão.

Para operacionalização da consciência das preferências dos usuários sugerimos duas alternativas: solicitar que os usuários informem suas preferências e inferir as preferências dos usuários. Como alternativas para operacionalização da consciência das ações dos usuários, sugerimos rastrear as ações dos usuários e detectar padrões de comportamento. As operacionalizações são apresentadas na Figura 11.

2.7. Uso e evolução do catálogo

Embora não haja uma ordem específica no processo de escolha da operacionalização aplicável a um domínio de problema específico, algumas das questões apresentadas devem ser respondidas com antecedência, a fim de orientar o processo de engenharia de requisitos, desde o levantamento de requisitos até a escolha da alternativa de operacionalização. Questões como "Quais metas o software deve satisfazer?", "Quais dados devem ser monitorados e interpretados?" e "Quais são as alternativas possíveis para a satisfação das metas (e metas flexíveis)?". As respostas a outras questões sugeridas, especialmente as sobre o ambiente físico, localização e identificação de usuários, tais como "O software irá operar em um ambiente interno ou em um lugar aberto?", "Que recursos estão disponíveis?" e "Como os usuários são identificados?", estão relacionados e são limitadas pelas tecnologias disponíveis para implementação do software.

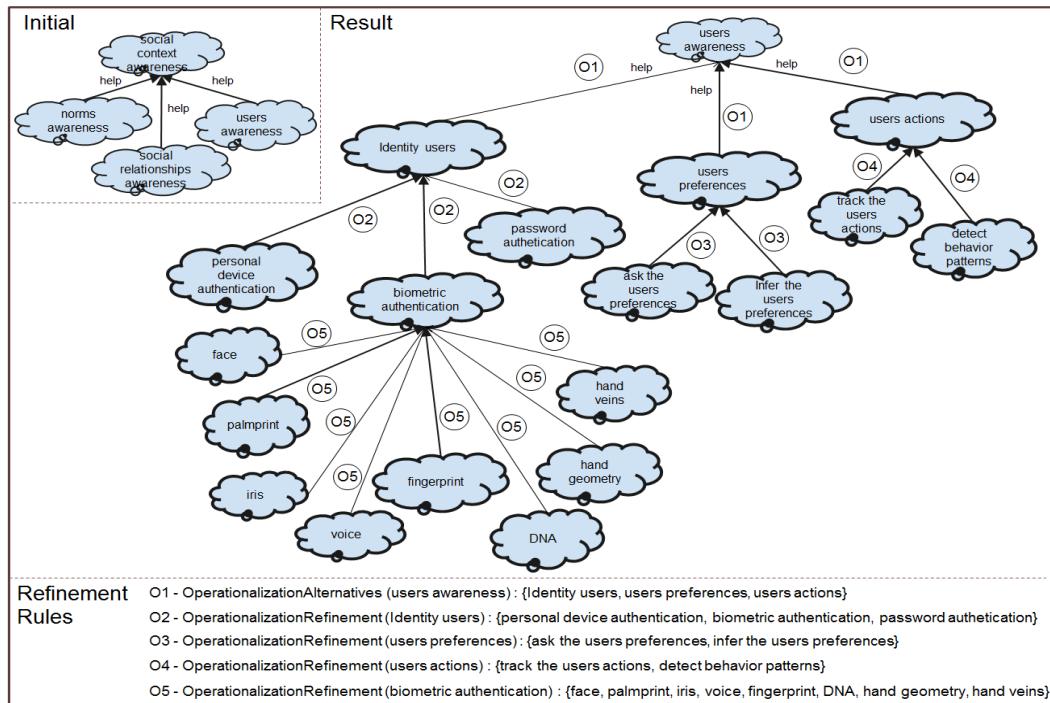


Figura 11 - operacionalizações sugeridas para consciência de usuários

Os padrões para requisitos não funcionais apresentados não foram concebidos para um domínio de problema específico e se propõem a ser de uso geral.

As questões propostas neste trabalho não são exaustivas. Novas questões podem ser adicionadas, seguindo o método GQO. Uma vez que uma questão tenha sido adicionada ao catálogo, mais alternativas de operacionalizações podem surgir e elas também devem ser adicionadas ao catálogo.

Da mesma forma que as perguntas, as operacionalizações sugeridas no catálogo não são exaustivas. Na verdade, outras operacionalizações podem ser adicionadas ao catálogo sempre que for identificada uma nova alternativa para responder a uma pergunta. A adição de novas alternativas ao catálogo de consciência de software não é só esperada, mas é desejada e incentivada. Ao fazer isso, o catálogo pode ser melhorado tornando-se mais completo e útil. Se uma nova alternativa de operacionalização for identificada sem responder diretamente a qualquer pergunta, esta alternativa poderia também ser adicionada ao catálogo. Neste caso, uma análise mais detalhada deve ser realizada para decidir se é ou não necessário acrescentar uma nova pergunta para o catálogo.

Outra maneira de se evoluir o catálogo de consciência de software é mudando o SIG apresentado na Figura 1. Isso pode ser feito por meio da adição de novos subtipos de consciência ou por um arranjo diferente dos subtipos de consciência no SIG. Sempre que ocorrer a adição de um novo subtipo, o método

GQO pode ser acionado levando a adição de novas perguntas, o que por sua vez conduzirá a adição de novas operacionalizações. No caso de um rearranjo dos subtipos no SIG, se um subtipo de consciência é colocado em outra posição no SIG, as perguntas com as respectivas operacionalizações podem ser realocadas juntamente com a meta flexível que representa o subtipo. Em caso de divisão de um subtipo em mais de um, uma análise mais profunda deve ser realizada para decidir quais perguntas e operacionalizações se aplicam a cada novo subtipo. Em caso de fusão de dois ou mais subtipos, uma análise detalhada deve ser realizada para decidir quais perguntas e operacionalizações ainda se aplicam ao subtipo resultante.

Nós não consideramos a exclusão de um subtipo de consciência porque isso iria restringir o conhecimento disponível no catálogo que pode ser reutilizado. Se parte do catálogo não se aplica em alguns casos, mas se aplica em outros casos, é preferível ter esse conhecimento disponível no catálogo.

2.8.

Considerações sobre o catálogo de consciência de software

De modo geral, modelos de requisitos são desenvolvidos para cada sistema, sem oferecer suporte adequado à reutilização de conhecimento sobre requisitos não funcionais. Engenheiros de software, no entanto, poderiam se beneficiar de um conjunto de "potenciais soluções" para metas de qualidade (requisitos não funcionais). Uma maneira eficaz de fazer isso é através do desenvolvimento de catálogos que listam possíveis operacionalizações de requisitos não funcionais, sendo esta uma das contribuições apresentadas em nosso trabalho.

A elaboração de um catálogo de meta flexível (representando um requisito não funcional) é uma atividade complexa. A primeira dificuldade é fundamental: não há consenso na comunidade de engenharia de software sobre o significado de muitas das qualidades de software. Isso tem sido observado por alguns autores, que tentaram definições mais precisas. Duboc et al., por exemplo, argumenta que a escalabilidade é uma qualidade que é mal entendida e comumente confundida com desempenho (Duboc et al., 2012).

O catálogo para consciência de software proposto não é a resposta final para lidar com a consciência no contexto de software. No entanto, é um primeiro passo e pode evoluir para tornar-se um catálogo mais completo e robusto para

alcançar o objetivo de ser uma ferramenta útil no desenvolvimento de software que tenha consciência como requisito não funcional.

Resumo do capítulo

Neste capítulo apresentamos a versão inicial de um catálogo para o NFR consciência de software. A construção do catálogo de consciência de software foi baseada principalmente em três trabalhos:

- i) NFR framework, que nos guiou no primeiro passo do processo de elaboração do catálogo de consciência de software, refinando o requisito não funcional em subtipos com um menor nível de abstração e, consequentemente, mais perto de alternativas de operacionalização;
- ii) método GQO, que é uma abordagem que ajuda na identificação e proposição de alternativas de operacionalização para os subtipos de NFR refinados;
- iii) nós organizamos o catálogo como padrões para requisitos não funcionais (*NFR Patterns*) que são mais adequados para a reutilização do conhecimento NFR.

Ao final do capítulo apresentamos uma breve discussão sobre o uso e evolução do catálogo e algumas considerações sobre o mesmo.

Nos próximos capítulos focaremos na implementação do requisito de consciência de software.

3

Modelagem do requisito de consciência de software

Neste capítulo é discutida e proposta uma abordagem para a modelagem do requisito de consciência. A questão central discutida neste capítulo é como representar o requisito de consciência em modelos que possam ser verificados e validados e que guiem os passos subsequentes no desenvolvimento do software. Usamos como base uma abordagem orientada a metas (Goal Oriented Requirement Engineering - GORE) na qual as abstrações necessárias para representar o requisito de consciência são adicionadas a modelos i^* (i -estrela). Uma vez produzidos e analisados, estes modelos enriquecidos com as abstrações de consciência, podem ser descritos em versão estendida de iStarML, uma linguagem de marcação para descrição de modelos i^* acrescida de elementos que representem as abstrações do requisito de consciência.

Consciência é um requisito fundamental para softwares que necessitem se autoadaptar em algum grau. A autoadaptação provê ao software a capacidade de lidar com mudanças no ambiente em que o software estiver inserido. O requisito de consciência, por sua vez, provê ao software a capacidade de perceber o que acontece no ambiente, “entender” como o ambiente muda, e de que modo as mudanças no ambiente afetam seu próprio funcionamento. Consciência também é fundamental para sistemas autônomos. De acordo com IBM (IBM Autonomic Computing White Paper, 2005), sistemas autônomos possuem a capacidade de se autogerenciar e executar suas tarefas através da escolha apropriada das ações a serem tomadas de acordo com uma ou mais situações que eles percebem no ambiente. São sistemas que percebem seu ambiente operacional, modelam seus comportamentos nesse ambiente, e tomam ações para mudar o ambiente ou o seu comportamento.

Vários pesquisadores consideram os termos autônomo (de *autonomic computing*) e autoadaptativo (*self-adaptive*) como sinônimos (Souza, 2012). Apesar da semelhança, Salehie e Tahvildari destacam algumas diferenças e consideram que o termo autônomo refere-se a um contexto mais amplo, a manipulação de todas as camadas da arquitetura do sistema (a partir de aplicações para hardware), enquanto que a autoadaptativo tem menos cobertura

- restrito principalmente a aplicações e middleware - e, assim, caindo sob o guarda-chuva da computação autônoma.

Vitor Souza também destaca como diferença o fato de a computação autônoma ser motivada pelo custo de manutenção de sistemas que possuem complexidade interna elevada (por exemplo, compiladores, sistemas de gerenciamento de banco de dados), enquanto a autoadaptação é motivada pela necessidade de implantação de sistemas de software em contextos sociais, abertos, com um alto grau de incerteza (Souza, 2012). Além disso, a pesquisa em computação autônoma se concentra em soluções de arquitetura para automatizar tarefas de manutenção, enquanto sistemas autoadaptativos geralmente estão mais preocupados com requisitos dos usuários, restrições e premissas do ambiente. Neste sentido, a pesquisa em sistemas autoadaptativos teria um alcance mais amplo, considerando-se todo o processo de desenvolvimento de software a partir de requisitos até a operação.

De um modo geral, a abordagem para operacionalização, tanto de sistemas autoadaptativos quanto de sistemas autônomos, passa por algum mecanismo com funções de monitoração e análise do contexto em que o software opera e ao qual deve se adaptar - mais adiante apresentaremos uma definição mais precisa de contexto.

A IBM propôs, em (IBM Autonomic Computing White Paper, 2005), uma arquitetura com implementações de *laços de controle* (*control loops*) em diversos níveis para com as funções de monitorar, analisar, planejar e executar (MAPE), alavancando o conhecimento do ambiente. No modelo de arquitetura proposto, no nível mais baixo se encontram os recursos (componentes de hardware ou software) a serem gerenciados. Os *touchpoints*, que estão no nível imediatamente acima dos recursos, são interfaces que implementam comportamentos de sensores e atuadores dos recursos. Nos níveis acima dos *touchpoints*, encontram-se os gerenciadores autônomos, que usam laços de controle (*control loops*) que implementam as funções monitorar, analisar, planejar e executar (MAPE).

As funções de monitoração e análise estão intrinsecamente ligadas ao requisito de consciência e aparecem em outras abordagens, como na tese de consciência de requisitos (*requirement awareness*) de Vitor Souza (Souza, 2012).

Em seu trabalho, Vitor Souza propõe o uso de laços de retroalimentação (*feedback loops*) como mecanismo de controle para monitorar os requisitos do software e disparar a adaptação ou a necessidade de evolução dos requisitos -

os loops de feedback fornecemos meios através dos quais sistemas adaptativos são capazes de monitorar indicadores importantes e, se estes indicadores mostrarem que o sistema não está funcionando corretamente, tomar a uma ação corretiva apropriada, a fim de se adaptar. Para tanto, usa uma abordagem de engenharia de requisitos orientada a meta na qual os requisitos são representados em modelos intencionais. A estes modelos, são acrescentados critérios de satisfação dos requisitos (indicadores) especificados em Object-Constraint Language (OCL). Essa especificação passa então a ser monitorada via *feedback loops* durante a execução do sistema, e dependendo do resultado da monitoração podem ser disparadas ações de adaptação ou evolução dos requisitos. Vale destacar que na tese de Vitor Souza, os requisitos (que são as entidades sobre as quais se deseja ter consciência) também são submetidos a funções de monitoração e análise através de *feedback loops*.

Outro trabalho importante é a tese sobre computação ubíqua de Albrecht Schmidt (Schmidt, 2002), cujo foco é sobre consciência de contexto (*context awareness*), no qual o autor propõe a criação de sistemas de sensoriamento para prover contexto a uma entidade, permitindo assim explorar o conhecimento de domínio que se torna disponível sobre aquela entidade - uma entidade podendo ser um lugar, um artefato, um assunto, um dispositivo, um pedido, um outro contexto, ou um grupo desses.

3.1. Abordagem orientada a meta

Nos últimos anos, a popularidade de abordagens de engenharia de requisitos orientadas a meta têm aumentado, tendo essas abordagens sido reconhecidas como úteis (Van Lamsweerde, 2000; Yu e Mylopoulos, 1998). A principal razão para isso é a pouca adequação das abordagens de análise de sistemas tradicionais em tratar sistemas de software cada vez mais complexos (Lapouchnian, 2005). Suposições incorretas sobre o ambiente de um sistema de software são conhecidas por ser um dos problemas responsáveis por muitos erros em especificações de requisitos. Os requisitos não funcionais também são, em geral, deixados de fora das especificações de requisitos. Além disso, técnicas de modelagem e análise tradicionais, exceto as derivadas de modelagem de processos, não permitem configurações de sistemas alternativos, onde mais ou menos funcionalidade é automatizada, ou diferentes atribuições de responsabilidade são exploradas para serem representadas e comparadas. A

engenharia de requisitos orientada a meta (*GORE: Goal Oriented Requirements Engineering*) tenta resolver estes problemas (Lapouchnian, 2005).

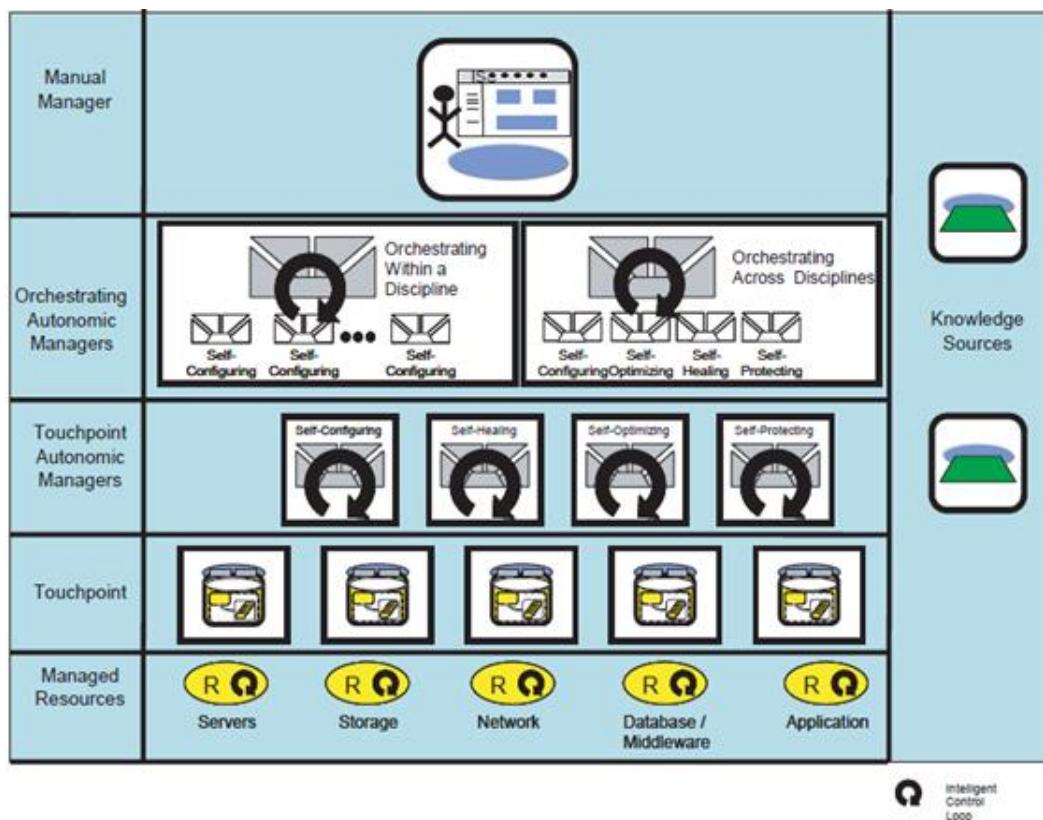


Figura 12 - Arquitetura de referência da IBM - computação autônoma

Enquanto a análise de sistemas tradicional foca em *quais* características (ex. atividades e entidades) um sistema irá suportar, as abordagens orientadas a meta focam no *porquê* sistemas são construídos provendo motivação e *rationale* para justificar os requisitos de software (Anton, 1996). A abordagem orientada a meta captura o motivo dos dados e das funções estarem presentes no software, e se são suficientes para atingir os objetivos de alto nível que surgem naturalmente no processo de engenharia de requisitos (Liu e Yu, 2004). A incorporação de representações explícitas de metas em modelos de requisitos fornece um critério de completude dos requisitos, ou seja, os requisitos podem ser julgados como completos se são suficientes para estabelecer as metas que eles estão refinando (Liu e Yu, 2004).

Outro ponto importante é que a modelagem orientada a meta é particularmente adequada para introdução nos modelos de requisitos do software os elementos necessários à representação do requisito de consciência. A variabilidade presente nos modelos, decorrente da possibilidade de se ter

diferentes alternativas para se alcançar uma determinada meta, é fundamental para representar as possibilidades de adaptação do software.

3.2. Engenharia de software orientada a agentes

Embora não haja um consenso universal sobre o que exatamente constitui um agente, um número crescente de pesquisadores considera a seguinte caracterização útil (Jennings, 1999):

Um agente é um sistema de computador encapsulado que está situado em algum ambiente, e que é capaz de atuar de forma flexível e autónoma neste ambiente, a fim de alcançar os objetivos para os quais foi projetado.

Jennings destaca certo número de pontos sobre esta definição que exigem mais explicações. Agentes são:

- (i) entidades claramente identificáveis, com limites e interfaces bem definidas, e que resolvem problemas;
- (ii) situados (embarcados) em um ambiente - eles recebem entradas relacionadas ao estado do ambiente através de seus sensores e agem sobre o meio ambiente através de seus atuadores²;
- (iii) projetados para preencher determinados papéis - possuem metas particulares a alcançar, que podem ser tanto explícita ou implicitamente representadas dentro dos agentes;
- (iv) autônomos, eles têm controle sobre seu estado interno e sobre o seu próprio comportamento;
- (v) capazes de exibir comportamento flexível (dependente do contexto) para resolução de problemas - eles precisam ser reativos - capazes de responder em tempo adequado às mudanças que ocorrem em seu ambiente, a fim de satisfazer os objetivos para os quais foram projetados) e proativos (aptos a adotarem de forma oportuna novas metas e tomar a iniciativa para satisfazer os objetivos para os quais foram projetados (Wooldridge e Jennings, 1995).

² Tipicamente cada agente tem uma visão parcial do ambiente (podendo ou não haver uma sobreposição com a visão de outros agentes) e uma esfera limitada de influência através da qual ele pode alterar o ambiente.

Ao adotar uma visão do mundo orientada a agentes, logo se torna evidente que um único agente é insuficiente. A maioria dos problemas exige ou envolve vários agentes: para representar a natureza descentralizada do problema, os múltiplos pontos de controle, as várias perspectivas, ou a competição de interesses. Além disso, os agentes terão de interagir uns com os outros, seja para atingir os seus objetivos individuais, ou então para gerenciar as dependências decorrentes de estarem situados em um ambiente em comum. Essas interações variam da interoperabilidade semântica simples (capacidade de trocar mensagens compreensíveis, por meio de interações do tipo cliente-servidor tradicionais, a capacidade de solicitar que uma determinada ação seja executada), para interações sociais mais ricas (a capacidade de cooperar, coordenar e negociar sobre um curso de ação). Qualquer que seja a natureza do processo social, no entanto, há dois pontos que qualitativamente diferenciam as interações entre agentes daquelas que ocorrem em outros paradigmas de engenharia de software. Em primeiro lugar, as interações orientadas a agentes geralmente ocorrem através de uma linguagem de comunicação de agentes de alto nível (declarativa), tipicamente baseada na teoria dos atos de fala. Consequentemente, as interações são geralmente realizadas ao nível de conhecimento (Yu, 1995): em termos de que metas devem ser seguidas, em que momento e por quem (em comparação com invocação de métodos ou chamadas de função que operam em um nível puramente sintático). Em segundo lugar, como agentes são solucionadores de problema com comportamento flexível, operando em um ambiente sobre o qual têm controle e capacidade de observação apenas parciais, as interações também precisam ser tratadas de forma flexível. Assim, os agentes precisam do aparato computacional para tomar decisões dependentes do contexto sobre a natureza e o escopo das suas interações e para iniciar (e responder a) interações que não foram necessariamente previstas em tempo de desenho (Wooldridge, 1995).

Em suma, a adoção de uma abordagem de engenharia de software orientada a agentes significa a decomposição do problema em vários, interativos e autônomos componentes (agentes) que têm objetivos específicos para alcançar.

Este tipo de abordagem, em virtude de suas características descritas acima, é mais adequado ao desenvolvimento de software autônomo ou alto-adaptativo, para os quais consciência de software é um requisito essencial.

3.3. Modelagem com i* (i-estrela)

Em nossa abordagem, usamos para modelagem dos requisitos o *framework* i* (i-estrela), descrito a seguir. i* (Yu, 1995) é bastante adequado para modelagem de requisitos orientada a meta e também a modelagem de requisitos orientada a agentes. Em i*, a decomposição do sistema é feita por atores, que podem ser especializados em agentes de software. Além das relações intencionais de dependência entre os atores/agentes, é elaborado um modelo interno de cada ator/agente a partir de suas metas individuais, com as possíveis alternativas para a satisfação dessas metas. Além disso, em i* é possível representar requisitos não funcionais como elementos de primeira grandeza inserindo-os nos modelos e considerando sua *satisfação a contento* na escolha das alternativas.

Por satisfazer plenamente "os requisitos" para introdução do requisito de consciência nos modelos de software (orientação a agente e orientação a meta), e pela nossa familiaridade³ com ela, escolhemos i* para modelagem de requisitos. Contudo, a mesma abordagem poderia ser feita com outra linguagem/framework que atendesse aos requisitos tal como KAOS (Van Lamsweerde, 2001), com as devidas adaptações.

O *framework* i* modela contextos organizacionais baseado nos relacionamentos de dependência entre os atores e possibilita a compreensão das razões internas dos atores, uma vez que as mesmas são expressas explicitamente, auxiliando na escolha de alternativas durante a etapa de modelagem do software. Atores são entidades ativas que efetuam ações para alcançar metas através do exercício de suas habilidades e conhecimentos, podendo ter dependências intencionais entre si. Uma dependência intencional ocorre quando o elemento da dependência está, de alguma forma, relacionado a alguma meta de um dos atores.

i*, como proposto originalmente por Yu, usa dois modelos: o modelo *Strategic Dependency* (SD) e o modelo *Strategic Rationale* (SR), descritos a seguir.

³ O grupo de engenharia de requisitos da PUC-Rio, liderado pelo Prof. Julio Cesar Sampaio do Prado Leite, vem trabalhando com i* há alguns anos, produzindo trabalhos como (Oliveira et al., 2008; Oliveira et al., 2010; Oliveira et al., 2011; Cunha, 2007; Serrano e Leite, 2011)

3.3.1. i* - Modelo SD

O modelo SD é usado para mapear a rede de dependências entre os atores organizacionais. O modelo SD é composto por um conjunto de nós que representam os atores (agentes, posições ou papéis) e elos que representam as dependências entre os atores. Uma dependência representa um acordo entre dois atores, onde um ator (“*depende*” ou “*dependente*”) depende de um outro ator (“*dependee*” ou “*de quem se depende*”) para que uma meta (*goal*) seja alcançada, uma tarefa seja executada, um recurso seja disponibilizado ou uma meta flexível (*softgoal*) seja razoavelmente satisfeita. Metas, metas flexíveis, tarefas e recursos que fazem parte das relações de dependência entre os atores são chamados de *elementos de dependência (dependum)* e caracterizam os quatro tipos de dependências possíveis, descritos a seguir:

- Dependência por meta: ocorre quando o *depende* depende do *dependee* para que certo estado do mundo seja alcançado. Ao *dependee* é dada à liberdade de escolher como fazê-lo. Com uma dependência por meta, o *depende* ganha a habilidade de assumir que a condição ou estado do mundo será alcançado, mas torna o *depende* vulnerável, pois o *dependee* pode falhar em realizar tal condição.
- Dependência por tarefa: ocorre quando um ator (o *depende*) depende de outro (o *dependee*) para que este outro desempenhe uma tarefa. Uma dependência por tarefa especifica “como” a tarefa deve ser desempenhada, mas não especifica o “porquê”. O *depende* é vulnerável, pois o *dependee* pode falhar em desempenhar a tarefa. A especificação de uma tarefa deve ser vista mais como uma restrição do que como o conhecimento prévio adequado (*knowhow*) para realização da tarefa.
- Dependência por recurso: ocorre quando um ator (o *depende*) depende de outro (o *dependee*) para que uma entidade (física ou computacional) seja disponibilizada. Com o estabelecimento deste tipo de dependências, o *depende* ganha a habilidade de usar a entidade como um recurso, ficando ao mesmo tempo vulnerável, pois a entidade pode tornar-se indisponível.

– Dependência por meta flexível: ocorre quando um ator (*o depender*) depende de outro (*o dependee*) para que este desempenhe alguma tarefa para que uma meta flexível seja “satisfeita a contento” ou “razoavelmente satisfeita”, isto é, seja satisfeita a um nível considerado aceitável. A expressão "satisfeita a contento" aqui é utilizada como uma tradução do termo em inglês *satisfice*, introduzido por Herbert Simon nos anos 1950. O conceito de meta flexível é usado para designar metas cujos critérios para sua satisfação não estão claramente definidos *a priori*, enquanto o conceito de meta rígida, chamadas apenas metas, está baseado numa noção clara em relação a sua satisfação, do tipo sim ou não - metas (rígidas) são usadas para representar determinados estados em que o mundo está ou não. (Esta subjetividade quanto a sua satisfação é inerente às metas flexíveis, e aos requisitos não funcionais - por se tratarem de requisitos de qualidade, os critérios para sua satisfação não estão claramente definidos *a priori*.) Numa dependência por meta flexível não há um acordo prévio entre os atores (*depende*r e *dependee*) sobre o que constitui a satisfação da meta (flexível). O significado da meta flexível é especificado em termos dos métodos que são escolhidos no curso da busca para alcançar a meta. Assim com em uma dependência por meta, o *depende*r ganha a habilidade de assumir que a condição ou estado do mundo será alcançado, mas torna-se vulnerável, pois o *dependee* pode falhar em realizar tal condição. Diferentemente do que ocorre com as metas, as condições para que as metas flexíveis sejam alcançadas são elaboradas ao passo que as tarefas são desempenhadas.

Estes quatro tipos de dependência também caracterizam como as decisões de processo (escolha de alternativas; sequência de passos para execução de uma tarefa) recaem em cada lado da dependência. Em uma dependência por meta, o *dependee* tem liberdade para tomar as decisões necessárias para que a meta seja satisfeita. Em uma dependência por tarefa o *depende*r toma as decisões. Em uma dependência por recurso a questão da decisão não vem à tona, por ser o recurso considerado um produto final de algum processo de ação-deliberação, não havendo assim nenhuma decisão em aberto a ser considerada. Em uma decisão por meta flexível, o *depende*r toma a decisão final, mas faz isso beneficiado pelo conhecimento prévio adequado

(*knowhow*) do *dependee*. A Figura 13 ilustra como são representados os quatro tipos de relação em um modelo SD.

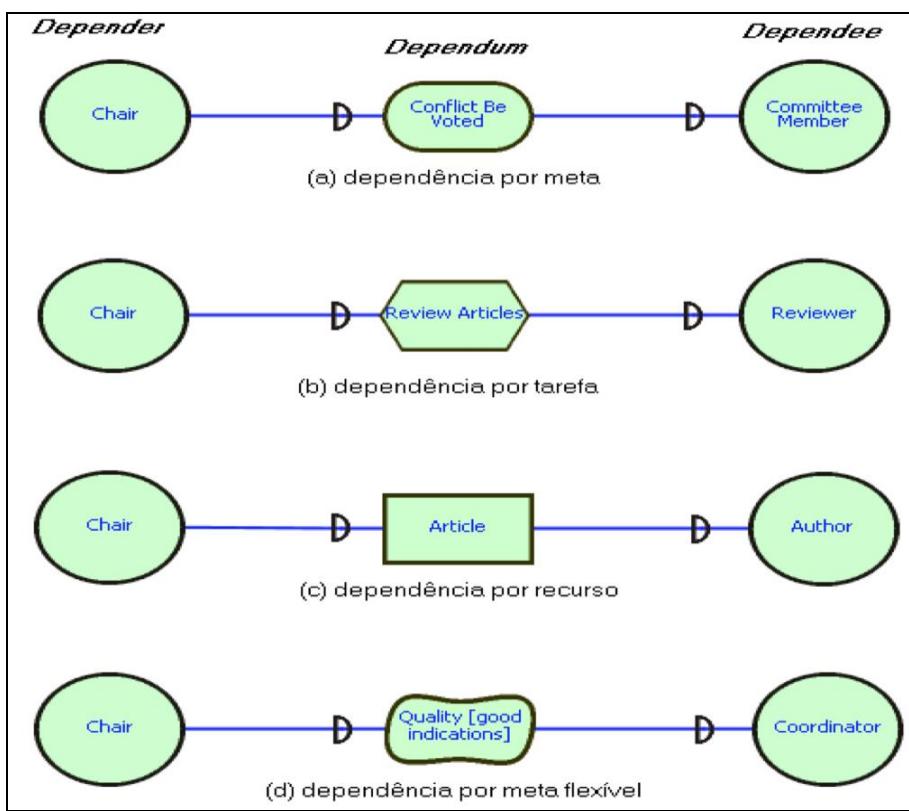


Figura 13 - Tipos de dependência em um modelo SD

3.3.2. i* - Modelo SR

O modelo SR tem como objetivo representar as estratégias internas de cada ator. O modelo SR provê uma descrição intencional do processo em termos dos elementos do processo e das decisões e escolhas por trás dele. O modelo SR permite representar explicitamente o entendimento detalhado do raciocínio (*rationale*) dos atores do processo. Estes modelos são elaborados de modo a expressarem o processo pelo qual: as metas são alcançadas, as tarefas são elaboradas, os recursos são disponibilizados e as metas flexíveis são refinadas (decompostas) e operacionalizadas. Isto é feito pela representação dos relacionamentos intencionais que são internos aos atores através de relacionamentos “meios-fim” que relacionam elementos de processo, provendo uma representação explícita do ‘porquê’, do ‘como’ e de alternativas.

O modelo SR é um grafo, com alguns tipos de nós e elos que proveem uma estrutura de representação para expressar explicitamente as razões por

trás do processo. Há quatro tipos de nós, idênticos aos tipos de *dependum* em um modelo SD: meta, tarefa, recurso e meta flexível. Há duas classes principais de elos: elos de decomposição de tarefa e elos “meios-fim”.

Elos de decomposição de tarefa – Uma tarefa (cuja noção está associada a ‘como fazer alguma coisa’) é modelada em termos de sua decomposição em suas subcomponentes, que podem ser: metas, tarefas, recursos e/ou metas flexíveis (os quatro tipos de nós).

Uma **meta** é uma condição ou estados de desejos no mundo que o ator deseja alcançar, expressa como uma assertiva na linguagem de representação. Como a meta deve ser alcançada não é especificado, possibilitando a consideração de várias alternativas.

Uma **tarefa** especifica um modo particular de fazer alguma coisa. Quando uma tarefa é especificada como subtarefa (parte) de outra tarefa (todo) ela restringe esta outra tarefa (todo) a um curso de ação em particular, especificado pela tarefa subtarefa.

Um **recurso** é uma entidade (física ou informacional) que não é considerada problemática pelo ator. A principal característica é se está disponível (e por quem foi disponibilizado no caso de uma dependência externa).

Uma **meta flexível** é uma condição ou estado no mundo que o ator deseja alcançar, mas diferentemente de uma meta (rígida), o critério para a condição ser alcançada não é precisamente definido *a priori*, estando sujeito à interpretação. Quando uma meta flexível é um componente em uma decomposição de tarefa, ela serve como uma meta de qualidade para aquela tarefa, guiando (ou restringindo) a seleção entre as alternativas para a decomposição da tarefa.

Elos Meios-Fim – Elos do tipo meios-fim indicam relacionamentos entre um “fim”, que pode ser: uma meta a ser alcançada, uma tarefa a ser desempenhada, um recurso a ser produzido, ou uma meta flexível a ser razoavelmente satisfeita (*satisficed*); e “meios” alternativos para alcançá-lo. Os meios são usualmente expressos na forma de tarefas, uma vez que a noção de tarefa está associada a ‘como fazer alguma coisa’. Na notação gráfica do i*, uma cabeça de seta aponta dos meios para o fim. Os tipos de elos meios-fim⁴ estão descritos a seguir:

⁴ Embora o tipo de elo se chame meios-fim (*means-end*), Yu os descreve de forma invertida: fim-meios. Visando facilitar a compreensão, optamos por descrevê-los de forma direta, diferentemente de Yu.

- Elo Tarefa-Meta – em um elo tarefa-metá o “fim” é especificado como uma meta e o “meio” é especificado como uma tarefa. Esta tarefa especifica “como” através de sua decomposição em seus componentes.
- Elo Tarefa-Recurso - neste tipo de elo o “fim” é especificado como um recurso e o “meio” é especificado como uma tarefa.
- Elo Tarefa-Meta Flexível - neste tipo de elo o “fim” é especificado como uma meta flexível e o “meio” é especificado como uma tarefa.
- Elo Meta Flexível–Meta Flexível – nos elos do tipo Meta Flexível–Meta Flexível tanto o “fim” quanto o “meio” são especificados como metas flexíveis. Este tipo de elo possibilita o desenvolvimento de uma hierarquia meios-fim de metas flexíveis, até que eventualmente algumas metas flexíveis são operacionalizadas por tarefas (via elos Tarefa-Meta Flexível). Esta hierarquia é particularmente interessante para o refinamento de metas flexíveis, onde uma meta flexível com nível de abstração alto é refinada sucessivamente até que se consigam metas flexíveis mais fáceis de operacionalizar.

Yu menciona ainda em (Yu, 1995) a possibilidade de uso de outros tipos de elos meios-fim como meta-meta, embora este tipo de elo não apareça em nenhum dos exemplos apresentados.

Os elos meios-fim que envolvem metas flexíveis possuem um atributo extra para indicar o tipo de contribuição. A contribuição individual de um meio representa o grau de contribuição desse meio para o fim. *MAKE* representa a situação positiva em que o desenvolvedor está suficientemente confiante para pensar que o meio em particular é "suficientemente bom" para a satisfação ("make") do fim. *BREAK* é utilizado quando o desenvolvedor está confiante de que o meio irá impedir ("quebrar") a satisfação do fim. Contribuições parciais são representadas por *HELP* ("+") nos casos positivos e *HURT* ("−") para casos negativos. *SOME+* representa alguma contribuição positiva, *HELP* ou *MAKE*, enquanto *SOME-* representa uma contribuição negativa, seja *HURT* ou *BREAK*.

Os elos de contribuição desempenham um papel importante na escolha de alternativas, especialmente nos casos em que, por alguma restrição, apenas uma alternativa possa ser escolhida para satisfação de um determinado fim.

3.3.3.

i* - Modelo SA (Strategic Actor)

O modelo SA (Leite et al., 2007) é usado especificamente para modelar os atores em i*. A adoção do modelo SA auxilia no entendimento dos atores, e seus relacionamentos, do ponto de vista estrutural. Embora Yu não tenha proposto um modelo específico com esta finalidade, o modelo SA usa os mesmos conceitos presentes em (Yu, 1995): agente, posição e papel como refinamento dos atores, e os relacionamentos entre eles. Estes conceitos estão descritos a seguir, conforme definido por Yu:

Autor: “Um ator é uma entidade ativa que desempenha ações para alcançar suas metas através do uso de seu conhecimento”. “O termo ator é usado para se referir genericamente a qualquer unidade a qual se possa atribuir dependências intencionais.” (Yu, 1995).

Posição: “Uma posição está em um nível intermediário de abstração entre um papel e um agente. É um conjunto de papéis tipicamente desempenhado”.

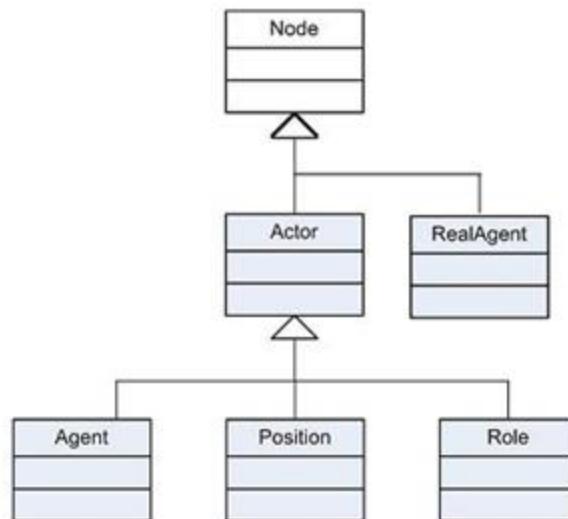


Figura 14 - Diagrama UML - Ator e seus refinamentos (Leite et al., 2007)

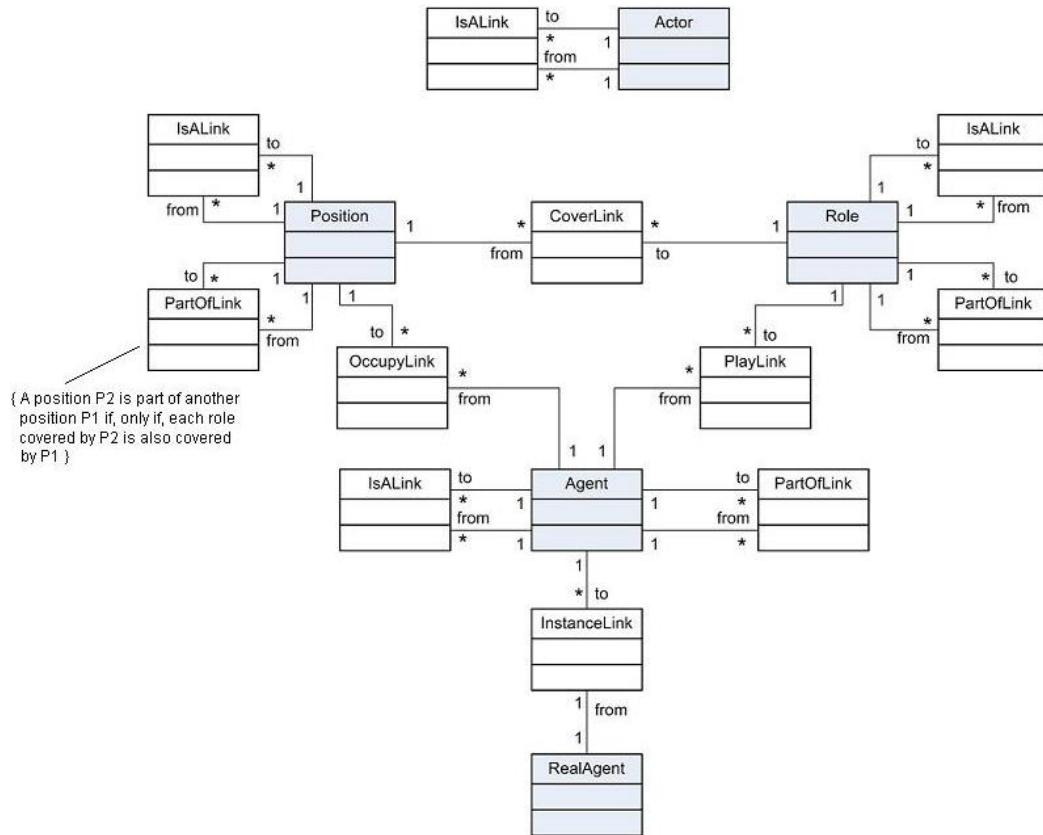


Figura 15 - Diagrama UML - Relações entre atores (Leite et al., 2007)

Papel: “Um papel é uma caracterização abstrata do comportamento de um ator social em algum contexto ou domínio especializado. Suas características podem facilmente ser transferidas para outros atores sociais. Dependências são associadas com papéis quando estas dependências se aplicam independentemente de quem esteja desempenhando o papel”.

Agente: “Um agente é um ator com manifestações físicas, concretas, tal qual um ser humano. O termo agente é usado no lugar de pessoa a fim de generalizar, podendo se referir tanto a agentes humanos quanto a agentes artificiais (hardware/software). Um agente tem dependências que se aplicam independentemente de que papel ele está desempenhando. Estas características não podem ser tipicamente transferidas para outros indivíduos, como por exemplo, suas habilidades, suas experiências, suas limitações físicas”. Em virtude do relacionamento de instanciação entre agentes, foi proposto em (Leite et al., 2007) um novo tipo de ator: o agente real. O agente real é mais específico que o agente (mais genérico), podendo ser identificado unicamente, como por exemplo, uma pessoa específica ou um software específico.

Os relacionamentos entre estes conceitos conforme descritos por Leite et al. baseado nas definições e exemplos de Yu, são apresentados resumidamente a seguir:

Ocupa - relacionamento de agente para posição – um agente pode ocupar mais de uma posição e uma posição pode ser ocupada por mais de um agente.

Desempenha – relacionamento de agente para papel – um agente pode desempenhar mais de um papel e um papel pode ser desempenhado por mais de um agente.

Cobre – relacionamento de posição para papel – uma posição pode cobrir mais de um papel e um papel pode ser coberto por mais de uma posição.

Parte de – relacionamento de posição para posição; de papel para papel; de agente para agente – posição, papel e agente podem ter mais de uma subparte, assim como podem ser subparte de mais de uma posição, papel e agente respectivamente. Há uma restrição para este relacionamento entre posições. Uma posição é parte de outra se todos os papéis desta posição também são cobertos pela outra posição.

É um – relacionamento de ator para ator; de posição para posição; de papel para papel; de agente para agente – ator, posição, papel e agente podem ser especializados por mais de um ator, posição, papel e agente respectivamente.

Instancia – relacionamento de agente real para agente – um agente real instancia um agente; um agente pode ser instanciado por mais de um agente real.

As Figuras 14 e 15 apresentam diagramas de classe UML com refinamentos de atores e suas relações.

3.4.

Representação descritiva dos modelos i* em iStarML

iStarML é uma especificação textual em formato compatível com o XML para representar diagramas i*, definida em (Cares et al., 2007), criada com o intuito de: (i) ter um formato de arquivo para intercâmbio de diagramas entre vários tipos de ferramentas de software i* específicas, tais como análise de metas, desenho, editores, cálculo de métricas; (ii) ter uma maneira comum de representar as diferenças e semelhanças entre as variações de i* existentes; (iii) ter uma representação comum para repositório de padrões i*; e (iv) tirar vantagens do formato XML para comunicação via Internet e também para uso de ferramentas XML gerais.

Utilizaremos a especificação de iStarML, apresentada a seguir para adicionar novos elementos a i* que permitam a modelagem de aspectos relacionados ao requisito de consciência de software.

Na especificação de iStarML serão usados os símbolos descritos na Tabela 1.

<i>Italic blue string</i>	significa um conceito da linguagem (no lugar dos símbolos tradicionais da BNF “<“ e “>“)
::=	significa uma definição da linguagem
[]	significa uma estrutura opcional da linguagem, 0 ou 1 vez
{ }	significa que a estrutura da linguagem pode ser repetida 0 ou mais vezes
()	grupo de estruturas da linguagem
	significa separação opcional

Tabela 1 - símbolos da BNF estendida utilizados

Os conceitos principais de i* e as tags iStarML correspondentes são apresentados na Tabela 2. Em seguida, são apresentadas tags complementares de iStarML na Tabela 3. A sintaxe de um arquivo iStarML válido é apresentada na Tabela 4.

Abstract core concept	Meanings and examples of core Specializations	Tag
Actor	An actor represents an entity which may be an organization, a unit of an organization, a single human or an autonomous piece of software. Also it	<actor>

	can represent abstractions over actors such as roles and positions.	
Intentional element	An intentional element is an entity which allows to relate different actors conforming a social network or, also, to express the internal rationality of an actor. Broadly used types of intentional elements are: goal, softgoal, resource, and task.	<ielement>
Dependency	A dependency is a relationship which represents the explicit dependency of an actor (depender) respect to the other actor (dependee). The dependency is expressed with respect to an intentional element.	<dependency> <dependee> <depender>
Boundary	A boundary represents a group of intentional elements. The common type of boundary is the actor's boundary which represents the vision of an omnipresent objective observer with respect to the actor's scope. However other boundary types can also be used.	<boundary>
Intentional element link	An intentional element link represents an nary relationship among intentional elements (either in the actor's boundary or outside). Broadly used types of intentional element link are decomposition, means-end and contribution. Related concepts such as routines or capabilities can be also represented using this relationship	<ielmentLink>
Actor association link	An actor relationship is a relationship between two actors. Broadly used types of actor relationships are is_a, is_part_of, instance_of (INS), plays, occupies and covers.	<actorLink>

Tabela 2 - Conceitos de i* e tags iStarML (Cares et al., 2007)

Additional Concept	Tag	Meaning
i* markup language file	<istarml>	The main tag of the iStarML
Diagram	<diagram>	A diagram is a particular i* diagram
Graphic expression	<graphic>	Represent some graphic properties of a particular diagram or diagram element.

Tabela 3 - Tags complementares de iStarML

<i>istarmlFile</i> ::=	<iistarml version="1.0"> <i>diagramTag</i> { <i>diagramTag</i> }</iistarml>
<i>diagramTag</i> ::=	<diagram <i>basicAtts</i> [author= <i>string</i>] { <i>extraAtt</i> } > [<i>graphic-diagram</i>] { [<i>actorTag</i>] [<i>ielementExTag</i>] } </diagram>
<i>extraAtt</i> ::=	<i>attributeName</i> = <i>attributeValue</i>
<i>basicAtts</i> ::=	[id=" <i>string</i> "] name=" <i>string</i> " id=" <i>string</i> " [name=" <i>string</i> "]

Tabela 4 - Sintaxe de iStarML / <iistarml> syntax

<i>actorTag</i> ::=	<actor <i>basicAtts</i> [<i>typeAtt</i>] { <i>extraAtt</i> } > [<i>graphic-node</i>] { <i>actorLinkTag</i> } [<i>boundaryTag</i>] </actor> <actor <i>basicAtts</i> [<i>typeAtt</i>] { <i>extraAtt</i> } /> <actor aref=" <i>string</i> " /> <actor aref=" <i>string</i> "> [<i>graphic-node</i>] </actor>
<i>typeAtt</i> ::=	type=" <i>actorType</i> "
<i>actorType</i> ::=	<i>basicActorType</i> <i>string</i>
<i>basicActorType</i> ::=	agent role position

Tabela 5 - Sintaxe de ator / <actor> syntax

<i>ielementTag</i> ::=	<ielement <i>ieAtts</i> > [<i>graphic-node</i>] { <i>ielementLinkTag</i> } </ielement> <ielement <i>ieAtts</i> /> <ielement iref=" <i>string</i> "/> <ielement iref=" <i>string</i> "> [<i>graphic-node</i>] </ielement>
<i>ielementExTag</i> ::=	<ielement <i>ieAtts</i> > [<i>graphic-node</i>] [<i>dependencyTag</i>] { <i>ielementLinkTag</i> } </ielement> <i>ielementTag</i>
<i>ieAtts</i> ::=	<i>basicAtts</i> type=" <i>itype</i> " [state=" <i>istate</i> "] { <i>extraAtt</i> }
<i>itype</i> ::=	<i>basic-itype</i> <i>string</i>
<i>basic-itype</i> ::=	goal softgoal task resource
<i>istate</i> ::=	undecided satisfied weakly satisfied denied weakly denied <i>string</i>

Tabela 6 - Sintaxe de elementos intencionais / <ielement> syntax

A Tabela 5 apresenta a sintaxe para atores em iStarML. Na Tabela 6, é apresentada a sintaxe para os elementos intencionais em iStarML. A Tabela 7 apresenta a sintaxe para a fronteira dos atores. Na Tabela 8 é apresentada a sintaxe para os elos entre os elementos intencionais. A Tabela 9 apresenta a sintaxe para dependência entre atores.

<i>boundaryTag</i> ::=	<boundary [type="string"]> [graphic-path] {[ielementTag] [actorTag]} </boundary>
------------------------	--

Tabela 7 - Sintaxe para fronteira dos atores / <boundary> syntax

<i>ielementLinkTag</i> ::=	<ielementLink linkAtts> [graphic-path] ielementTag {ielementTag} </ielementLink>
<i>linkAtts</i> ::=	type = "decomposition" [value=("and" "or")] type="means-end" [value="string"] type="contribution" [value="contribution-value"]
<i>contribution-value</i> ::=	+ - sup sub ++ -- break hurt some- some+ unknown equal help make and or

Tabela 8 - Sintaxe para elos entre elementos / <ielementLink> syntax

<i>dependencyTag</i> ::=	<dependency>dependertag {dependertag} dependeeTag {dependeeTag} </dependency>
<i>dependertag</i> ::=	<dependertag [iref="string"] aref="string" [value="dep-type"] /> <dependertag [iref="string"] aref="string" [value="dep-type"] > [graphic-path] </dependertag>
<i>dependeeTag</i> ::=	<dependee [iref="string"] aref="string" [value="dep-type"] /> <dependee [iref="string"] aref="string" [value="dep-type"] > [graphic-path] </dependee>
<i>dep-type</i> ::=	open committed critical delegation permission trust owner string

Tabela 9 - Sintaxe para dependência entre atores / <dependency> syntax

Finalmente a Tabela 10 apresenta a sintaxe para elos entre atores.

<i>actorLinkTag ::=</i>	<actorLink type=" <i>actorLink-type</i> " aref=" <i>string</i> "> [<i>graphic-path</i>] </actorLink> <actorLink type=" <i>actorLink-type</i> " aref=" <i>string</i> ">
<i>actorLink-type ::=</i>	is_part_of is_a instance_of plays covers occupies <i>string</i>

Tabela 10 - Sintaxe para elos entre atores / <actorLink> syntax

3.4.1.

Acrescentando abstrações de consciência aos modelos de requisitos

Nossa proposta é acrescentar, aos modelos i*, abstrações que ajudem o software a perceber o ambiente com suas mudanças inerentes, relacionando estas novas abstrações aos demais elementos do modelo que determinam o comportamento do software. As abstrações serão **situação** e **contexto**, como definidos por Schmidt em (Schmidt, 2002).

De acordo com Schmidt, **contexto** é "um mecanismo para descrever situações por suas características definidoras e agrupá-los em uma unidade" ou em outras palavras: "um contexto é uma descrição da situação atual em um nível abstrato, que pode ser comparado com situações previamente especificadas". Segundo ele, "descrição constitui-se de uma série de condições que podem ser avaliadas como verdadeiras ou falsas, possivelmente com um grau de certeza atribuído". Uma **situação**⁵ é o estado do mundo em um determinado momento ou durante um intervalo de tempo em um determinado ambiente.

Ainda segundo Schmidt, as seguintes propriedades de contexto são fundamentais:

- Cada contexto é ancorado em uma entidade.
- A consciência do contexto está sempre relacionada a uma entidade.

Uma entidade poderia ser um lugar, um artefato, um assunto, um dispositivo, uma aplicação, um outro contexto, ou um grupo destes.

⁵ Adaptamos a definição de situação de Schmidt para torná-la mais abrangente. A definição original era: "um estado do mundo *real*, um determinado momento ou intervalo de tempo, em um determinado local". Esta definição difere da definição de Zorman [87] para situação (de ocorrência de um cenário): um determinado momento no tempo e um contexto geográfico.

Em nosso trabalho, consideramos como entidades sobre as quais o contexto pode ser ancorado os principais elementos de i*: atores (inclusive suas especializações: agentes, papéis e posições), elementos intencionais (metas, metas flexíveis, tarefas e recursos) e dependências.

Para que possamos então representar as situações de um determinado contexto em modelos i*, precisamos estender o *framework* acrescentando o elemento *Context* aos elementos intencionais originais do *framework*. Apresentamos a seguir mais detalhes sobre essa extensão.

3.5. Extensão do framework i*

Para representar o requisito de consciência nos modelos i*, além de introduzir as abstrações de situação e contexto, relacionando-as com as demais abstrações existentes em i*, precisamos também representar como o requisito de consciência pode ser operacionalizado, guiando assim a implementação. Assim, propomos extensões que visam:

- Representar consciência através de contexto, com suas respectivas situações;
- Permitir a operacionalização (refinamento) do requisito de consciência guiando a implementação do mesmo.

Contextos, com suas respectivas situações, estão diretamente relacionados com o domínio do problema. Nos casos em que o requisito de consciência for importante as situações reais subjacentes à operação do software mudam em alguma medida, demandando software autoadaptação e/ou autonomia em algum nível. Um ponto chave é identificar quais são estas situações que o software deve perceber durante sua operação. Esta identificação pode ser feita a partir das metas dos agentes de software e das alternativas para sua satisfação.

Uma vez identificadas que situações são relevantes em um determinado domínio de problema, as seguintes perguntas, relativas à implementação, precisam ser respondidas:

i - "Que dados devem ser adquiridos?";

ii - "Como obter os dados desejados?";

iii - "Como os dados adquiridos podem ser interpretados?".

Os instrumentos utilizados para a aquisição de dados variam de acordo com o subtipo de consciência de software em questão, assim como os mecanismos utilizados para interpretar estes dados variam de acordo com o tipo dos dados e a quantidade dos dados disponíveis. Além disso, os dados que devem ser adquiridos variam de acordo com o domínio do problema.

A abordagem que propomos para operacionalizar o requisito de consciência é dividida em duas etapas integradas:

- Aquisição de dados, equivalente à função de monitoração no MAPE;
- Interpretação dos dados adquiridos, equivalente à função de análise no MAPE.

Mais precisamente, por interpretação dos dados queremos dizer uma função para mapear as instâncias de contexto adquiridas no processo de aquisição de dados em um conjunto de situações conhecidas, enquanto a aquisição de dados é uma função que dado uma descrição contexto retorna a situação subjacente representado por esta descrição em um determinado instante de tempo.

Propomos um mecanismo de interpretação de dados dividido em dois subpassos sequenciais:

- Identificação situação a partir dos dados adquiridos;
- Avaliação da ação alternativa.

A identificação tem por objetivo identificar a qual das situações de um determinado contexto uma situação percebida, através dos dados adquiridos, pertence, ou seja: em qual situação o agente se encontra em um determinado momento. Em outras palavras, a identificação de situação é uma função que, dada uma situação percebida s e um contexto C , avalia se s pertence C .

Para responder as perguntas "como o comportamento do software é influenciado pela situação de contexto subjacente?" e "como o software pode perceber seu próprio comportamento sob a influência de uma situação de contexto?" é preciso detalhar como software poderia compreender contexto. O objetivo principal da etapa de interpretação de dados é permitir que o agente possa, após perceber a situação subjacente no mundo real e identificar a que

situação de contexto previamente definida essa situação real percebida pertence, (re)agir de acordo. Assim, é necessária a construção de uma "ponte" entre as situações identificadas através dos dados de contexto adquiridos, e as alternativas para a satisfação das metas dos agentes, uma vez que as alternativas representam as possíveis ações que o agente poderia tomar (ou os pontos de variabilidade no comportamento do software). Uma abordagem interessante é definir um conjunto de situações no mundo real, que pode ser capturado através da aquisição de dados, e para as quais software poderia decidir como (re)agir, ou em outras palavras, as situações que o agente poderia raciocinar sobre as alternativas para atingir as suas metas atuais. Esse passo de avaliação da ação alternativa na etapa de interpretação dos dados pode ser automatizado por uma função que, dada uma situação específica pertencente a um contexto e uma meta relacionada a esse contexto, retorna a melhor alternativa de ação que levará à satisfação da meta na situação específica.

De uma maneira mais precisa, descrita a seguir em lógica de predicados, temos:

Entity E: - representa a entidade sobre a qual o contexto é ancorado, podendo ser um ator (agente, papel ou posição), um elemento intencional (meta, metas flexível, tarefa ou recurso) ou uma dependência entre atores.

Context C(E): representa um contexto, caracterizado por um conjunto de situações, sobre uma entidade *E*

ContextDescription(E): $\{ v_1, v_2, \dots, v_n \}$ - um conjunto finito não vazio de variáveis utilizadas para descrever situações em que uma entidade *E* se encontra no mundo real

Domain D(v): um conjunto (finito ou infinito) de possíveis valores para cada variável *v* em *ContextDescription(E)*.

AcquiredData(E): $\{ x_1, x_2, \dots, x_n \}$ - onde x_i é o valor adquirido da variável v_i , $v_i \in \text{ContextDescription}(C)$ e $x_i \in D_i$

DataAquisition(E, t): AcquiredData(E)_t - em que t é o tempo (discreto), e *AcquiredData(E)_t* representa os dados adquiridos da entidade E no instante t

ContextSituations(C(E)) : {s₁, s₂, ..., s_n} – representa o conjunto de situações que pertencem a um contexto $C(E)$

SituationIdentification(C, AcquiredData(E)) : s – função que retorna a situação $s \in ContextSituations(C)$, cujos critérios para cada variável $v_i \in C$ sejam atendidos pelos respectivos valores de x_i de $AcquiredData(E)$. Caso contrário, retorna vazio.

$G(C)$: representa a meta relacionada ao contexto C

AlternativesActions(G): {a₁ v a₂ v ... v a_n} - - a_i é uma alternativa para satisfação de G .

AlternativeActionChoice(s, G(C)): a_i - - onde $a_i \in AlternativesActions(G(C))$ e a_i é a melhor alternativa que irá levar a satisfação de G na situação s .

Para viabilizar essa abordagem, é preciso representar nos modelos i*:

- as situações de contexto;
- a ligação das situações de contexto com as entidades nas quais estão ancoradas;
- a ligação das situações de contexto com as alternativas à satisfação das metas relacionadas às situações.

Para tanto, acrescentamos em iStarML os seguintes elementos:

- O tipo de elemento intencional para contexto - a serem representadas em iStarML como um novo tipo (*contextawareness*) na tag `<ielement>`.
- O elo entre o elemento de contexto e os meios alternativos para se alcançar a meta relacionada ao contexto - representado em iStarML por um novo tipo de elo entre os elementos intencionais. Chamamos esse novo tipo de elo de argumentação – um conceito oriundo do NFR Framework, de (Chung et al., 2000), utilizado para registrar o racional na escolha de uma alternativa.

Assim, acrescentamos a linha abaixo ao final da Tabela 2 - Conceitos principais de linguagens de modelagem baseadas em i* e as tags iStarML correspondentes.

ContextAwareness	A context is a set of situation at a given time or over a period of time at a given environment. A situation is a state of the world. Context awareness is a special task (mean) to <i>satisfy</i> the awareness NFR (end)	<ielement type="conte xtawareness " />
------------------	--	---

As demais alterações necessárias, juntamente com a sintaxe em iStarML, são apresentadas a seguir (em fonte de cor vermelha). A Tabela 11 apresenta a sintaxe para ielementTag acrescida de *context-awareness*. A Tabela 12 apresenta a sintaxe para ielementLinkTag acrescida de *argumentation*.

Em virtude das alterações introduzidas alterarem, ainda que pontualmente, a gramática original de iStarML, consideramos que essa nova gramática define uma nova linguagem que chamaremos de iStarML estendida.

O conjunto de situações pertencentes ao contexto é representado através dos elos de argumentação do elemento de contexto para as alternativas (meio em um elo meios-fim) que as situações impactam. Tipicamente, uma situação impacta positivamente o meio através do qual a meta será satisfeita na ocorrência desta situação, modificando o grau de contribuição dos meios positivamente nesses casos. É possível também que a ocorrência de uma determinada situação impacte negativamente uma alternativa, sinalizando que nesse caso a alternativa não deve ser escolhida.

<i>ielementTag</i> ::=	<ielement <i>ieAtts</i> [graphic-node] {ielementLinkTag} </ielement> <ielement <i>ieAtts</i> > <ielement iref="string"/> <ielement iref="string"> [graphic-node] </ielement>
<i>ielementExTag</i> ::=	<ielement <i>ieAtts</i> >[graphic-node] [dependencyTag] {ielementLinkTag} </ielement> <i>ielementTag</i>
<i>ieAtts</i> ::=	<i>basicAtts</i> type="itype" [state="istate"] {extraAtt}
<i>itype</i> ::=	<i>basic-itype</i> string
<i>basic-itype</i> ::=	goal softgoal task resource context-awareness
<i>istate</i> ::=	undecided satisfied weakly satisfied denied weakly denied string

Tabela 11 - Sintaxe de elementos intencionais estendida / <ielement> syntax

<i>ielementLinkTag</i> ::=	<ielementLink <i>linkAtts</i> [<i>graphic-path</i>] <i>ielementTag</i> { <i>ielementTag</i> } </ielementLink>
<i>linkAtts</i> ::=	type = “decomposition” [value=(“and” “or” ⁶)] type=“means-end” [value=“ <i>string</i> ”] type=“contribution” [value=“ <i>contribution-value</i> ”] type=“argumentation” situation-name=“<i>string</i>” [value=“<i>contribution-value</i>”] type=“ <i>string</i> ” [value=“ <i>string</i> ”]
<i>contribution-value</i> ::=	+ - sup sub ++ -- break hurt some- some+ unknown equal help make and or

Tabela 12 - Sintaxe para elos entre elementos intencionais estendida

Assim, através de elos de argumentação é possível representar tantas as situações de um contexto quanto o impacto que estas situações têm na escolha de alternativas para satisfação das metas as quais o contexto está relacionado.

Como nossa abordagem é dividida nos passos aquisição de dados e interpretação dos dados, propomos uma construção particular para o novo elemento adicionado a i*: decompor o elemento de consciência de contexto (*context-awareness*) nas subtarefas aquisição de dados e interpretação de dados.

O conjunto com a meta flexível de consciência (fim) juntamente com o elemento de consciência de contexto (meio) e as subtarefas para aquisição de dados e interpretação de dados, formam uma estrutura canônica conhecida como *SRConstruct* idealizado por Oliveira em (Oliveira et al., 2008). Um *SRConstruct* estabelece uma estratégia para a satisfação de uma meta que pode ser reusada em outros casos. Podemos afirmar que um *SRConstruct* constitui um padrão de desenho para modelos i*. Assim o conjunto que propomos define um padrão de desenho para operacionalização do requisito de consciência em modelos i*. A Figura 16 apresenta o *SRConstruct* para a meta flexível de consciência, destacado na cor azul. As situações são representadas por elos de argumentação, destacados na cor vermelha, cujos rótulos são precedidos de um sinal de exclamação (“!”).

⁶ Em i* como definido em (Yu, 1995) o elo de decomposição de tarefas é do tipo “and”.

3.6.

Considerações sobre a modelagem do requisito de consciência

Um ponto chave para a engenharia do requisito de consciência é definir o conjunto de situações que pertencem a um determinado contexto. Para algumas classes de problemas, a identificação das situações e o impacto do contexto para satisfação das metas pode ser direta e explícita. Há problemas em que a manutenção de metas (estados desejados do mundo) consiste em monitorar alguns sinais no ambiente e (re)agir em conformidade com as mudanças. Nessas classes de problemas, o comportamento correspondente em resposta à situação de contexto subjacente pode ser definido de uma forma determinística: baseado em valores de variáveis monitoradas é possível decidir qual a melhor ação a ser tomada em todos os casos (todas as combinações de variáveis).

Por exemplo, considere um ambiente inteligente, onde com uma porta automática que abre quando as pessoas se aproximam da entrada/saída. Depois de um curto intervalo, por exemplo, 10 segundos, se não houver uma pessoa perto da porta, ela deve ser fechada.

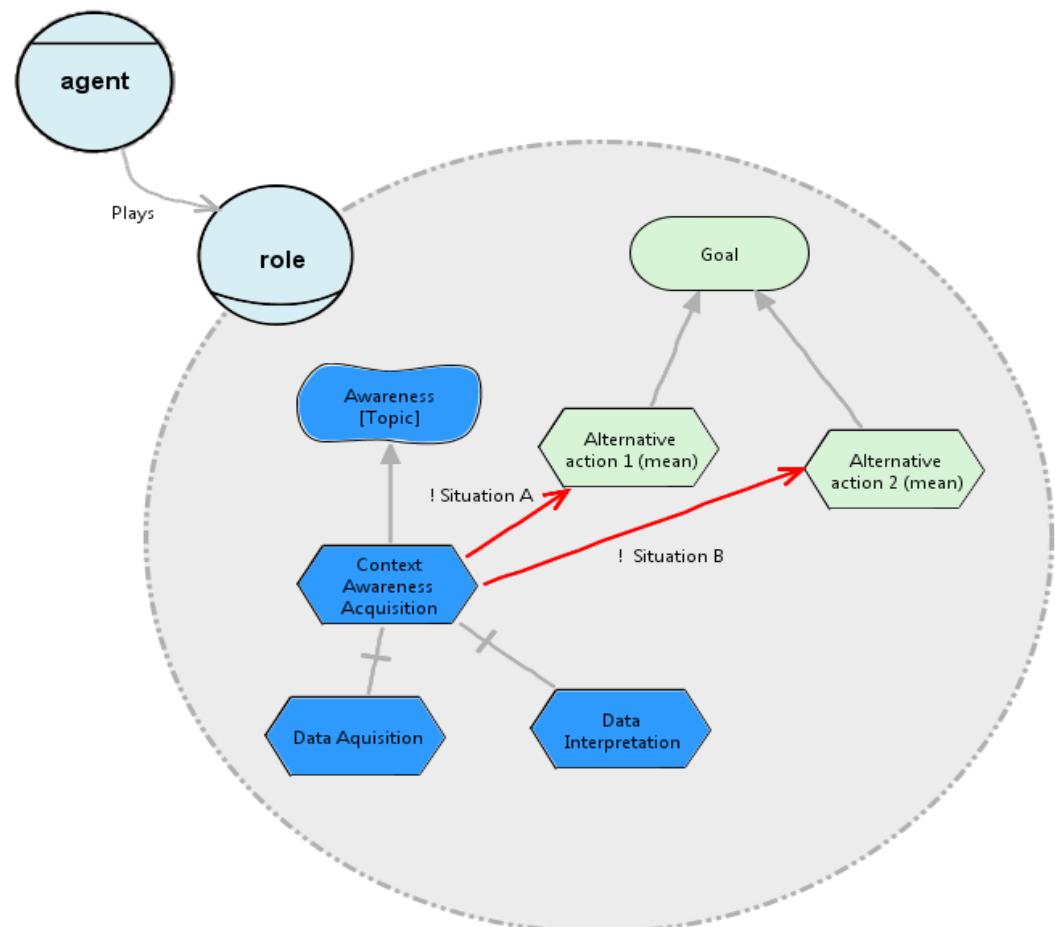


Figura 16 - SRConstruct para a meta flexível de consciência

As duas situações possíveis nesse caso são "presença de pessoas" e "ausência de pessoas". Adicionalmente, o agente que controla a porta pode eventualmente receber alertas do sistema de prevenção de incêndios informando a provável presença de fogo no recinto. Neste caso, o agente que controla a porta deve abri-la para que as pessoas que se encontram dentro da sala possam deixar o recinto em segurança. A Figura 17 apresenta o modelo i* estendido para esse problema.

As situações "presença de pessoa(s)" e "ausência de pessoa(s)" são mutuamente exclusivas. Mas, a situação "provável presença de fogo" é ortogonal a estas duas e poderia acontecer simultaneamente com ambas. Uma opção é combinar as variáveis (e as situações) a fim de obter situações mutuamente excludentes a fim de facilitar à avaliação de alternativas.

A especificação de um contexto pode ser descrita através do modelo apresentado a seguir.

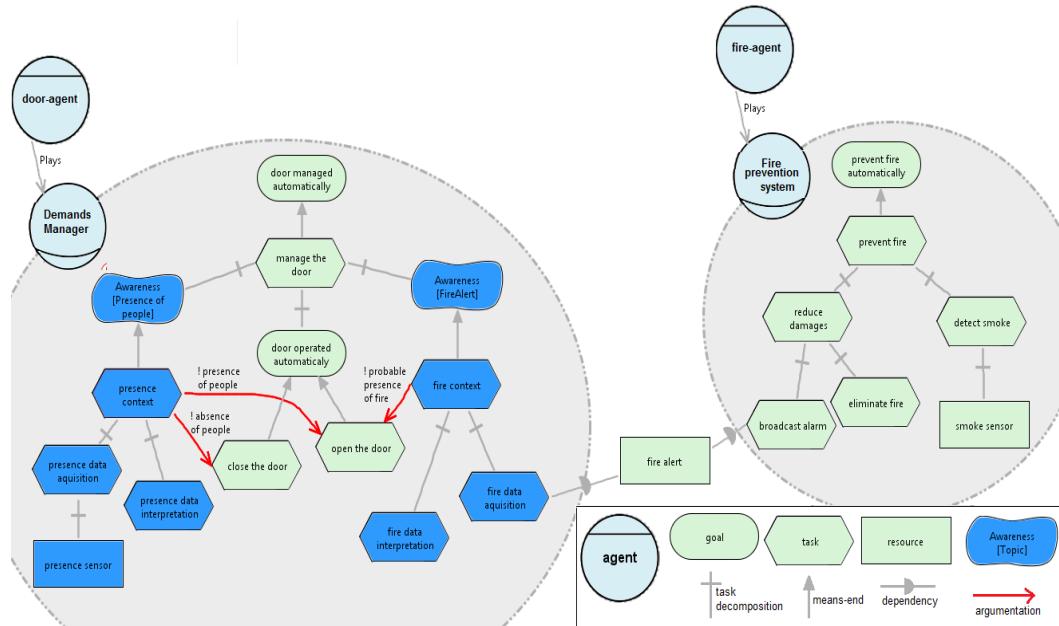


Figura 17 - modelo i* estendido para problema da porta automática

Awareness requirement name <i><the name should follow the rule: Awareness [Topic]></i>		
Topic description:	<i><Brief description of the topic (problem domain) related to the awareness requirement></i>	
Goal:	<i><The goal impacted by the context></i>	
Awareness subtype:	<i><The awareness subtype (from awareness catalog) which is related to this requirement></i>	
Suggested operationalizations:	<i><Suggested operationalizations to this requirement. Some of them can be found in awareness catalog ></i>	
Alternative actions:	<i><The alternative means to achieve the goal impacted by the context></i>	
Entity:	<i><The entity in which the context awareness element is anchored (what the context is about)></i>	
Source of entity data:	<i><The source from where the entity data will be acquired></i>	
ContextDescription	<i><List of variables that enables the situations identification></i>	
Domains of variable <i>< domain definition section for variables in context description</i>		
Variable name	Domain	
Context situations specification <i><specification section for the context situations></i>		
Situation name	Especification	
Alternative action choice <i><specification of relations among situations and alternative actions></i>		
Situation	Alternative action	Impact

Tabela 13 - Modelo para especificação de consciência

Os modelos preenchidos para o exemplo do sistema para o controle automático da abertura/fechamento da porta, citado anteriormente, são apresentados a seguir.

Awareness[Presence of People]		
Topic description:	The agent should open the door whenever there are people near the door. In order to do that, the agent needs to be aware of the presence of people near the door and act according to his perception.	
Goal:	Door managed automatically	
Awareness subtype:	Physical environment	
Suggested operationalizations:	Use of presence sensors	
Alternative actions:	Open the door; Close the door	
Entity:	People using the room.	
Source of entity data:	People presence sensor	
ContextDescription	{peoplePresence}	
Domains of variable		
Variable name	Domain	
peoplePresence	{"PRESENCE", "ABSENCE"}	
Context situations especification		
Situation name	Specification	
PresenceOfPeople	peoplePresence=="PRESENCE"	
AbsenceOfPeople	peoplePresence=="ABSENCE"	
Alternative action choice		
Situation	Alternative action	Impact
Presence of people	OpenTheDoor	MAKE (strongly positive)
Presence of people	CloseTheDoor	BREAK (strongly negative)
Absence of people	CloseTheDoor	MAKE (strongly positive)
Absence of People	OpenTheDoor	BREAK (strongly negative)

Awareness[Fire Alert]		
Topic description:	The agent should open the door whenever there is a fire alert. In order to do that, the agent needs to be aware of the existence of fire alerts broadcasted by the Fire System.	
Goal:	Door managed automatically	
Awareness subtype:	Physical environment; Dependency	
Suggested operationalizations:	Use of smoke detection sensors; Communicate with fire prevention system	
Alternative actions:	Open the door; Close the door	
Entity:	Fire in the room	
Source of entity data:	Fire system	
ContextDescription	{fireAlert}	
Domains of variable		
Variable name	Domain	
fireAlert	{"FIRE"}	
Context situations especification		
Situation name	Specification	
Probable presence of fire	\exists fireAlert	
No fire detected	not (\exists fireAlert)	
Alternative action choice		
Situation	Alternative action	Impact
Probable presence of fire	OpenTheDoor	MAKE (strongly positive)
Probable presence of fire	CloseTheDoor	BREAK (strongly negative)
No fire detected	OpenTheDoor	No impact (neutral)
No fire detected	CloseTheDoor	No impact (neutral)

Em seguida é apresentada a descrição do modelo em iStarML, com os elementos de contexto e seus respectivos elos destacados em vermelho.

```
<istarml version="2.0">
<diagram>
    <actor type="role" id="1" name="DoorSystem">
        <boundary>
            <ielement id="1.1" type="task" name="CloseTheDoor"
                basic="true"/>
            <ielement id="1.2" type="task" name="OpenTheDoor"
                basic="true"/>
            <ielement id="1.3" type="goal"
                name="DoorOperatedAutomatically" >
                <ielementLink type="means-end"
                    run-mode="unique">
                    <ielement iref="1.1"/>
                    <ielement iref="1.2"/>
                </ielementLink>
            </ielement>
            <ielement id="1.4" type="resource"
                name="PresenceSensor" basic="true"/>
            <ielement id="1.5" type="task"
                name="presence_data_aquisition">
                <ielementLink type="decomposition"
                    run-mode="sequential" >
                    <ielement iref="1.4"/>
                </ielementLink>
            </ielement>
            <ielement id="1.6" type="task"
                name="PresenceDataInterpretation" basic="true"/>
            <ielement id="1.7" type="context-awareness"
                name="PresenceContext">
                <ielementLink type="decomposition"
                    run-mode="sequential">
                    <ielement iref="1.5"/>
                    <ielement iref="1.6"/>
                </ielementLink>
                <ielementLink type="argumentation">
                    <ielement iref="1.1" situation-name=
                        "absence of people"/>
                    <ielement iref="1.2" situation-name=
                        "presence of people"/>
                </ielementLink>
            </ielement>
            <ielement id="107" type="softgoal"
                name="Awareness[PresenceOfPeople]" >
                <ielementLink type="means-end"
                    run-mode="unique">
                    <ielement iref="1.7"/>
                </ielementLink>
            </ielement>
            <ielement id="1.8" type="task" name="FireDataAquisition"
                basic="true"/>
            <ielement id="1.9" type="task"
```

```

        name="FireDataInterpretation" basic="true"/>
<ielement id="1.10" type="context-awareness"
name="FireContext" >
    <ielementLink type="decomposition"
run-mode="sequential" >
        <ielement iref="1.8"/>
        <ielement iref="1.9"/>
    </ielementLink>
    <ielementLink type="argumentation">
        <ielement iref="1.2" situation-
name="probable presence of fire"/>
    </ielementLink>
</ielement>
<ielement id="110" type="softgoal"
name="Awareness[FireAlert]">
    <ielementLink type="means-end"
run-mode="unique">
        <ielement iref="1.10"/>
    </ielementLink>
</ielement>
<ielement id="1.11" type="task"
name="manage_the_door">
    <ielementLink type="decomposition"
run-mode= "sequential">
        <ielement iref="107"/>
        <ielement iref="110"/>
        <ielement iref="1.3"/>
    </ielementLink>
</ielement>
<ielement id="1.12" type="goal" name=
"DoorManagedAutomatically" main="true">
    <ielementLink type="means-end"
run-mode="unique">
        <ielement iref="1.11"/>
    </ielementLink>
</ielement>
</boundary>
</actor>

<actor type="role" id="2" name="Fire prevention system">
    <boundary>
        <ielement id="2.1" type="resource" name="SmokeSensor"
basic="true" />
        <ielement id="2.2" type="task" name="detect_smoke" >
            <ielementLink type="decomposition"
run-mode= "sequential">
                <ielement iref="2.1"/>
            </ielementLink>
        </ielement>
        <ielement id="2.3" type="task" name="BroadcastAlarm"
basic="true"/>
        <ielement id="2.4" type="task" name="EliminateFire"
basic="true"/>
        <ielement id="2.5" type="task" name="reduce_damages">
    </boundary>

```

```

<ielementLink type="decomposition" run-mode=
"sequential">
    <ielement iref="2.3"/>
    <ielement iref="2.4"/>
</ielementLink>
</ielement>
<ielement id="2.6" type="task" name="prevent_fire">
    <ielementLink type="decomposition"
run-mode= "sequential">
        <ielement iref="2.2"/>
        <ielement iref="2.5"/>
    </ielementLink>
</ielement>
<ielement id="2.7" type="goal"
name="PreventFireAutomatically"
main="true">
    <ielementLink type="means-end"
run-mode="unique">
        <ielement iref="2.6"/>
    </ielementLink>
</ielement>
</boundary>
</actor>

<ielement id="3" type="resource" name="FireAlert" waiting_time="3500">
    <dependency>
        <dependee aref="1" /> <!-- "fire data acquisition" -->
        <dependee iref="2.3" aref="2" /> <!-- "broadcast alarm" -->
    </dependency>
</ielement>

<actor id="4" type="agent" name="door-agent">
    <actorLink type="plays" aref="1"/>
</actor>

<actor id="5" type="agent" name="fire-agent">
    <actorLink type="plays" aref="2"/>
</actor>

</diagram>
</istarml>

```

É particularmente interessante notar que, Schmidt declara que "em certos casos, a descrição pode consistir de uma única condição complexa que não é, necessariamente, legível - por exemplo: uma rede neural artificial". A etapa de identificação de situação pode ser realizada incorporando aprendizagem ao comportamento do software. Basicamente, existem três cenários:

- nenhuma aprendizagem após o desenvolvimento ser concluído;
- fase de aprendizagem dedicada durante o uso;
- aprendizagem contínua.

O exemplo apresentado anteriormente se enquadra no cenário mais simples, onde nenhum aprendizado é necessário ao software. Mais adiante, apresentaremos exemplos para os cenários mais complexos.

3.7.

Processo de definição do requisito de consciência

De acordo com Leite, o processo de engenharia de requisitos é composto por quatro macro atividades: elicitação, modelagem, análise e gerência (Leite, 2006). A atividade de elicitação tem como principal objetivo a aquisição de informações (e requisitos), oriundas do “Universo de Informações”. A atividade de modelagem tem por objetivo documentar os requisitos através de modelos. A atividade de análise tem por objetivo a verificação e validação dos requisitos modelados. A atividade de gerência é desempenhada paralelamente a estas três atividades, com o objetivo de evoluir os requisitos de forma controlada e garantir a rastreabilidade entre os requisitos. Chamaremos de “definição de Requisitos” a composição das atividades de elicitação, modelagem e análise. A Figura 18 apresenta um diagrama *Structured Analysis and Design Technique – SADT* (Marca e McGowan, 1987) com as etapas de um processo para a definição do requisito de consciência. Este processo é ortogonal a qualquer processo de requisitos.

A etapa de elicitação (1) tem por objetivo elaborar uma lista dos contextos a serem percebidos pelo agente, com o nome do contexto, a descrição do problema e a meta relacionada ao contexto com suas respectivas alternativas de ação. Nesta etapa, o catálogo de consciência de software pode ser consultado para auxiliar na elicitação dos contextos.

A etapa seguinte (2) tem por objetivo a identificação das situações pertinentes a cada contexto elicitado na etapa anterior. Caso necessário pode-se retornar a etapa anterior para complementar a elicitação, com o auxílio das situações identificadas.

Uma vez identificadas, as situações são especificadas na etapa seguinte (3), onde para cada situação é definido um nome, a descrição do contexto que permitirá identificar a ocorrência da situação, as variáveis da descrição de contexto com seus respectivos domínios e o impacto da situação nas alternativas de ação da meta relacionada. Caso necessário pode-se retornar a etapa anterior para complementar a identificação de situações, com auxílio das especificações das mesmas.

A etapa de modelagem do contexto (4) pode ocorrer simultaneamente à etapa de especificação, pois são complementares. Na etapa de modelagem são adicionados os elementos de contexto. Cada elemento de contexto é ligado aos meios (de elos meios-fim para a meta do contexto) que representam as ações alternativas através de elos de argumentação. Cada elo de argumentação representa uma situação específica, e pode ter o valor de contribuição da situação descrito no elo de argumentação. Como as etapas de especificação e modelagem são complementares, e podem ocorrer simultaneamente, o modelo elaborado é usado para auxiliar a especificação assim como a especificação é usada para auxiliar a elaboração do modelo.

Após as etapas de especificação e modelagem, vem a etapa de análise (5), onde especificação e modelo são verificados e validados. Nesta etapa é verificado se o modelo está de acordo com as regras e boas práticas de modelagem em i*, e se a especificação contém todos os elementos do modelo para especificação de consciência apresentado na Tabela 13, e se modelo e especificação estão consistentes entre si. Ambos, modelo e especificação, precisam também ser validados junto a um especialista do domínio do problema. Adicionalmente, podem ser identificadas e disparadas nessa etapa necessidades de evolução do catálogo de consciência de software.

Finalmente, na etapa de implementação (6), modelo e especificação são usados na implementação de um sistema multiagente (MAS - MultiAgent System), onde o requisito de consciência será incorporado ao comportamento dos agentes. Nos capítulos seguintes apresentaremos mais detalhes sobre a implementação do requisito de consciência.

3.8.

Heurísticas para uso do catálogo na definição do requisito de consciência

A seguir, apresentamos na Tabela 14 um conjunto de heurísticas apresentadas na forma de perguntas que, juntamente com o catálogo apresentado anteriormente, julgamos ser úteis na definição do requisito de consciência de software.

As respostas às perguntas apresentadas na Tabela 14 auxiliam na definição do requisito de consciência de software. Essas perguntas sintetizam pontos que devem ser levados em consideração durante o processo e podem

também ser úteis como uma lista de verificação do requisito, especialmente para uso na atividade de análise.

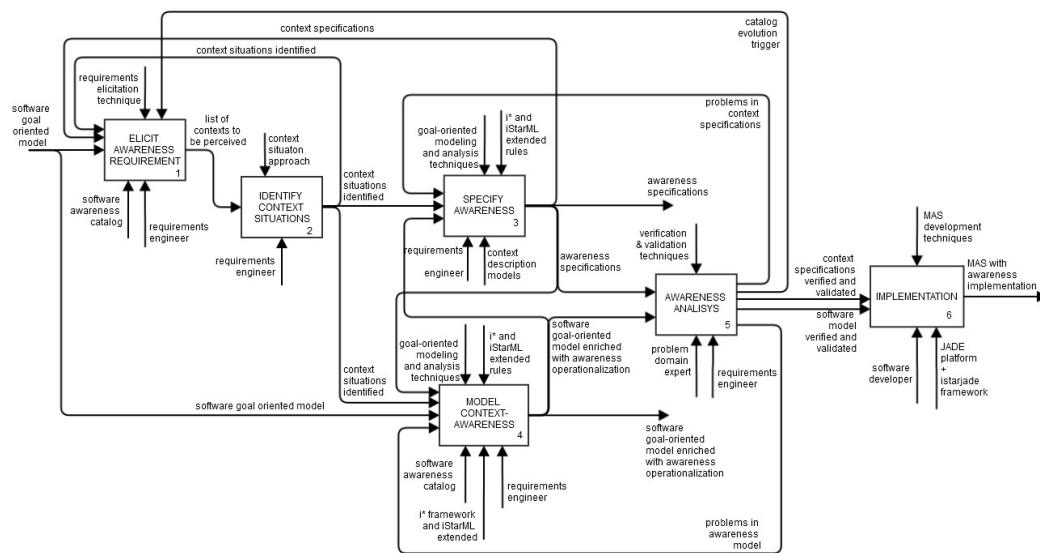


Figura 18 - Visão geral das etapas para definição do requisito de consciência

Etapa	Questão	Racional
Quanto à elição de requisito de consciência	Quais metas o software terá de satisfazer?	As respostas a essas perguntas ajudam a eliciar as necessidades de adaptação do comportamento do software.
	Quais as alternativas para satisfazer as metas?	
	Durante sua operação, a que contextos o software terá que se adaptar?	As respostas a esse grupo de perguntas ajudam a eliciar que propriedades o software deve perceber. Essas propriedades correspondem aos subtipos de consciência apresentados nos catálogo, onde podem ser encontradas sugestões de operacionalizações
	O software terá de se adaptar a mudanças no ambiente físico em que irá operar?	
	A localização é importante para a operação do software?	
	O software terá de se adaptar ao usuário?	
Quanto à identificação de situações de contexto	Quais situações podem ocorrer em determinado contexto?	
	Como as situações mudam?	
	Como as situações impactam na satisfação das metas?	
	Como as mudanças de situação podem ser percebidas/detectadas pelo software?	As respostas a essas perguntas ajudam a detalhar como contextos podem ser percebidos
Quanto à modelagem da consciência	Que dados precisam ser adquiridos/monitorados para que se possa perceber a situação em que o software se encontra?	As respostas ajudam a construir o modelo intencional com o requisito de consciência relacionado às metas do software
	Como esses dados podem ser interpretados?	

	Qual alternativa (à satisfação das metas) é mais adequada em cada situação?	
Quanto à especificação	Como as situações podem ser percebidas/identificadas individualmente a partir dos dados obtidos?	As respostas, juntamente com o modelo para especificação proposto, ajudam a elaborar uma especificação do requisito de consciência
	Qual o impacto individual de cada situação para as alternativas das metas a que está relacionada?	
	As situações podem ocorrer simultaneamente? Em caso afirmativo, qual tem maior impacto na satisfação das metas?	
Quanto à análise do requisito de consciência	Foram elicitados os contextos nos quais o software irá operar?	As respostas a essas perguntas indicam se a especificação está boa o suficiente para guiar a implementação ou se alguma das etapas anteriores precisa ser refeita
	Para cada contexto, foram elicitadas as situações que podem vir a ocorrer?	
	Foram definidos os dados necessários para que o software perceber a situação em que se encontra?	
	Foi definido como os dados devem ser interpretados?	
	Foi definido como o software deve avaliar as alternativas à satisfação de suas metas em cada situação que o software venha a se encontrar?	

Tabela 14 - Heurísticas para auxiliar a definição do requisito de consciência

Resumo do capítulo

Neste capítulo discutimos as abordagens de engenharia de requisitos mais adequadas para tratar o requisito de consciência de software e apresentamos as justificativas para nossa opção por uma abordagem orientada a meta e orientada a agentes. Apresentamos também i*, *framework* escolhido como base para modelagem e análise orientada a meta e agentes, juntamente com a linguagem iStarML que o descreve. Introduzimos nessa linguagem as abstrações necessárias para representar contexto e suas situações, elementos que julgamos necessários em nossa abordagem do requisito de consciência de software, relacionando esses novos elementos ao modelo de comportamento do software a ser desenvolvido e apresentamos, juntamente com a abordagem proposta para sua operacionalização, um modelo para especificação do requisito de consciência.

4

Framework istarjade

Neste capítulo apresentamos o framework *istarjade* usado para implementação do requisito de consciência. O framework foi desenvolvido em JADE, uma plataforma de software para implementação de sistemas multiagente (*MultiAgent System - MAS*). A ideia principal é que os requisitos sejam capturados em modelos *i** e descritos em arquivos *iStarML* que serão mapeados automaticamente pelo framework para sua implementação em JADE. Neste capítulo apresentamos, de forma não muito sucinta, mas necessária para compreensão do framework, os conceitos principais de JADE e seu funcionamento, o framework que desenvolvemos com o mapeamento dos elementos de *iStarML* para JADE e o funcionamento deste framework.

Optamos por usar JADE como plataforma para implementação dos modelos desenvolvidos em *iStarML* por JADE ser conforme a especificação da FIPA (The Foundation for Intelligent Physical Agents) e amplamente utilizada. Nossa opção por implementar diretamente em JADE e não em JADEX, deveu-se aos seguintes motivos:

- embora *i** sirva perfeitamente para representar modelos de agentes que sejam compatíveis com o modelo BDI, é possível e válido ter um modelo em *i** que não siga essa abordagem;
- em *i** não existe o conceito de plano - a estratégia para resolução de problemas em *i** é baseada fortemente na variabilidade de alternativas, representada pelos elos meios-fim, e na decomposição de tarefas.

Por isso, acreditamos que o mapeamento de *i** diretamente para JADE nos ajuda a contornar esses pontos: não seria preciso criar o conceito de plano nos modelos *i**, e os mecanismos de variabilidade de alternativas e decomposição de tarefas poderiam ser implementados diretamente em comportamentos JADE, como apresentaremos a seguir.

4.1. JADE

JADE (Java Agent DEvelopment Framework) é um *framework* totalmente implementado na linguagem Java. JADE simplifica a implementação de sistemas multiagente através de um *middleware* que é conforme com as especificações FIPA e através de um conjunto de ferramentas gráficas para suporte as fases de depuração de código e implantação (Bellifemine et al., 1999; Bellifemine et al., 2003). A plataforma de agentes pode ser distribuída por diferentes máquinas (que nem sempre precisam compartilhar o mesmo sistema operacional) e a configuração pode ser controlada via interface gráfica do usuário (GUI) remota. A configuração pode inclusive ser modificada em tempo de execução, movendo agentes de uma máquina para outra, como e quando for necessário. A seguir, com base em (Bellifemine et al., 2010; Giovanni, 2009), descrevemos os principais conceitos de JADE.

JADE segue o modelo padrão de uma plataforma de agente, tal como definido pelo FIPA, representado pelas Figuras 19 e 20. O Sistema de Gestão de Agente (AMS - Agent Management System) é o agente que exerce o controle de supervisão sobre o acesso e uso da Plataforma Agent. Apenas um AMS irá existir em uma única plataforma.

O AMS fornece o serviço de página-branca e de ciclo de vida, mantendo um diretório de identificadores de agentes (AID) e estado do agente. Cada agente deve se registrar com um AMS para obter um AID válido.

O Diretório Facilitador (DF) é o agente que fornece o serviço padrão de páginas amarelas na plataforma. O sistema de transporte de mensagens, também chamado de *Agent Communication Channel* (ACC), é o componente de software que controla toda a troca de mensagens dentro da plataforma, incluindo mensagens de/para plataformas remotas.

JADE é totalmente conforme a esta arquitetura de referência e, quando uma plataforma JADE é lançada, o AMS e DF são criados imediatamente. Além disso, o serviço de mensagens (que implementa o componente ACC) é sempre ativado para permitir a comunicação baseada em mensagens.

A plataforma de agentes pode ser dividida em vários hosts. Normalmente (mas não necessariamente) apenas uma aplicação Java e, portanto, apenas uma *Java Virtual Machine* (JVM), é executada em cada host. Cada JVM é um *container* básico de agentes que fornece um ambiente completo de tempo de execução para a execução do agente e permite a execução de vários agentes

simultaneamente no mesmo host. O *container* principal é o *container* onde o AMS e DF “moram”. Os outros *containers* conectam-se ao *container* principal, fornecendo um ambiente completo de tempo de execução para a execução de qualquer conjunto de agentes em JADE.

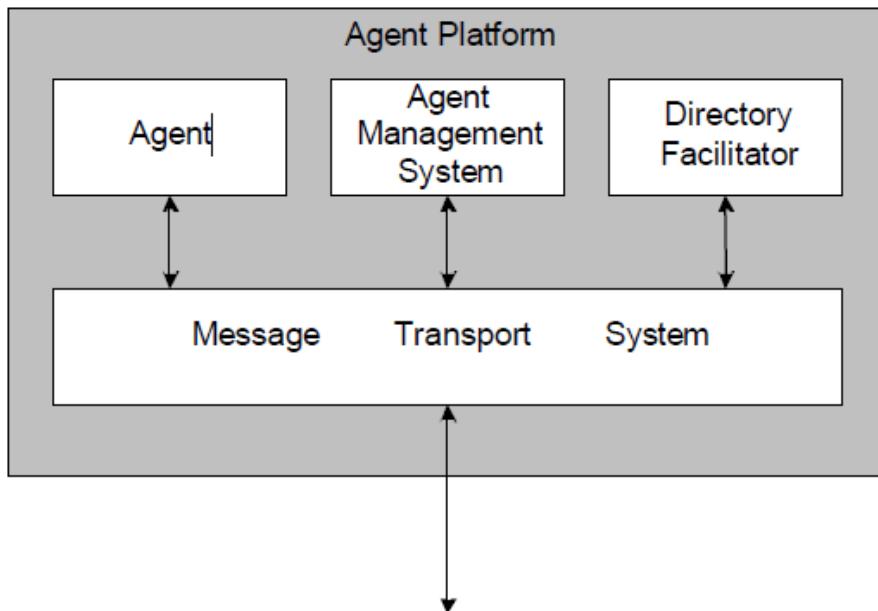


Figura 19 - Arquitetura plataforma de agentes FIPA (Bellifemine et al., 2010)

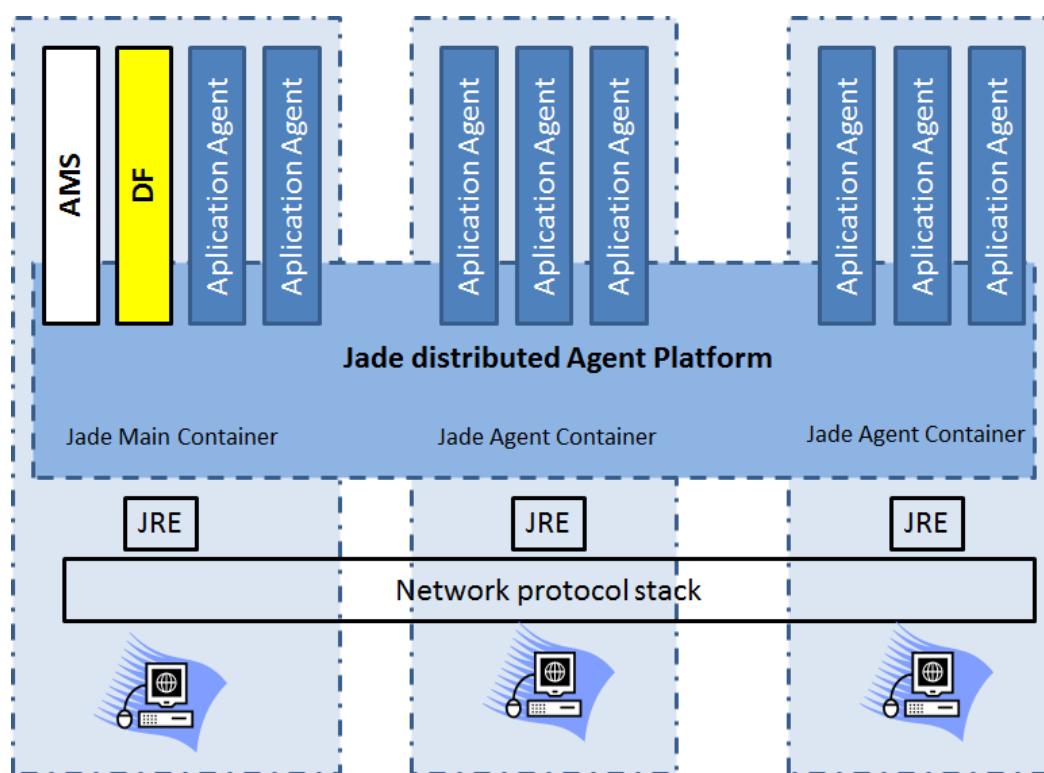


Figura 20 - Arquitetura de agentes detalhada FIPA (Bellifemine et al., 2010)

4.1.1.

Agentes em JADE

A classe *Agent* representa uma classe base comum para os agentes definidos pelo usuário. Portanto, do ponto de vista do programador, um agente JADE é simplesmente uma instância de uma classe Java definida pelo usuário que estende a classe *Agent base*. Isto implica na herança de características para realizar as interações básicas com a plataforma de agentes (registro, configuração, gerenciamento remoto) e um conjunto básico de métodos que podem ser chamados para implementar o comportamento personalizado do agente (por exemplo, enviar/receber mensagens, usar os protocolos de interação padrão, registrar-se em vários domínios). O modelo computacional de um agente é multitarefa, onde tarefas (ou comportamentos) são executadas concorrentemente. Cada funcionalidade/serviço prestado por um agente deve ser implementado como um ou mais comportamentos. Um agendador, interno à classe *Agent base* e escondido do programador, gerencia automaticamente o agendamento de comportamentos.

Um agente JADE pode estar em um dos vários estados, de acordo com o ciclo de vida de agente na especificação FIPA, representados na Figura 21 e detalhados a seguir:

- INICIADO : o objeto agente é construído, mas não se registrou ainda no AMS, não tem um nome, nem endereço e não pode se comunicar com outros agentes.
- ATIVO : o objeto agente está registrado no AMS, tem nome e endereço regular e pode acessar todos os vários recursos de Jade.
- SUSPENSO : o objeto agente está parado. A sua *thread* interna está suspensa e nenhum comportamento do agente está sendo executado.
- ESPERANDO : o objeto agente está bloqueado, esperando por algo. A sua *thread* interna está dormindo em um monitor Java e irá acordar quando alguma condição for atendida (normalmente quando uma mensagem chega).
- DELETADO: o Agente está definitivamente morto. A *thread* interna terminou a sua execução e o agente não está mais registrado no AMS.

TRÂNSITO: um agente móvel entra nesse estado enquanto está migrando para o novo local. O sistema continua a armazenar as mensagens (em um *buffer*) que serão então enviadas para o seu novo local.

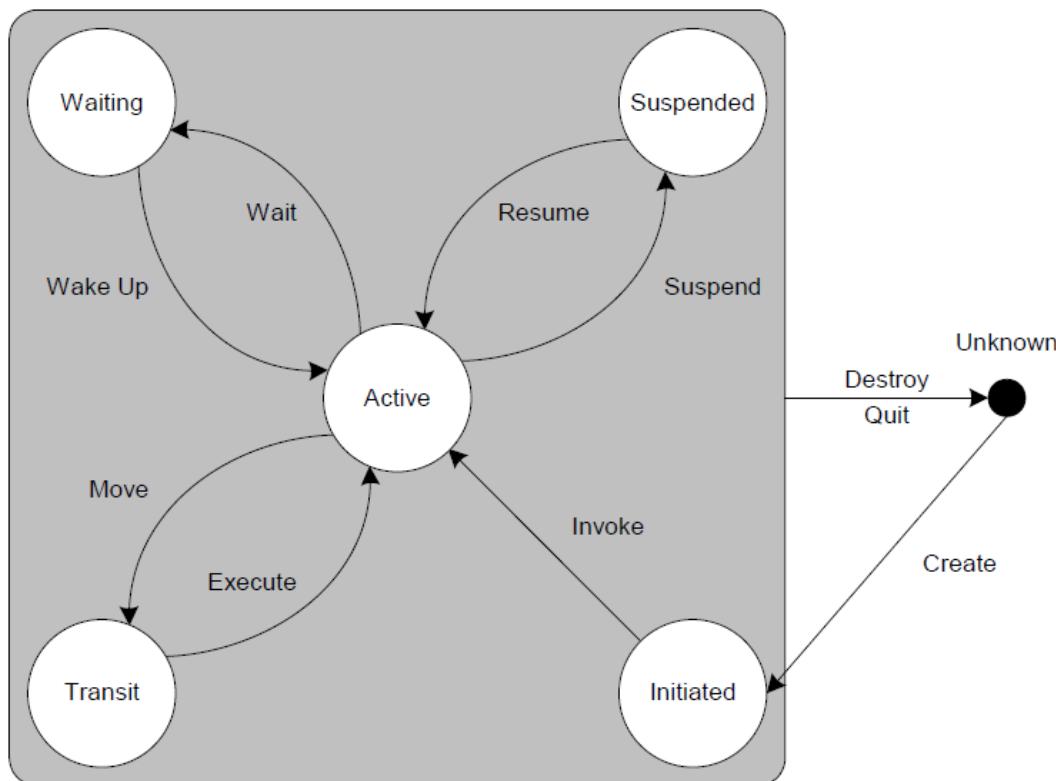


Figura 21 - Ciclo de vida de um Agente FIPA (Bellifemine et al., 2010)

A classe *Agent* fornece métodos públicos para realizar transições entre os vários estados, estes métodos levam seus nomes a partir de uma transição adequada na Máquina de Estado Finita mostrada na Figura 21 da especificação FIPA para gestão de agentes.

4.1.2. Comportamento dos agentes em JADE

O trabalho real que um agente tem que fazer é normalmente realizado através de "comportamentos". Um comportamento representa uma tarefa que um agente pode realizar e é implementado como um objeto de uma classe que estende *jade.core.behaviours.Behaviour*. Para fazer um agente executar a tarefa implementada por um objeto de comportamento é suficiente adicionar o comportamento ao agente por meio do método *addBehaviour()* da classe do agente. Comportamentos podem ser adicionados em qualquer momento: quando um agente é iniciado (no método *setup()*) ou a partir de outros comportamentos. Cada classe estendendo a classe *Behaviour* deve implementar o método *action()*, que, na verdade, define as operações a serem executadas quando o comportamento está em execução e o método *done()* (retorna um valor booleano), que especifica se um comportamento foi concluído ou não, e se

deve ser removido do conjunto de comportamentos que um agente está realizando. A Figura 22 ilustra o caminho de execução de uma *thread* de agente.

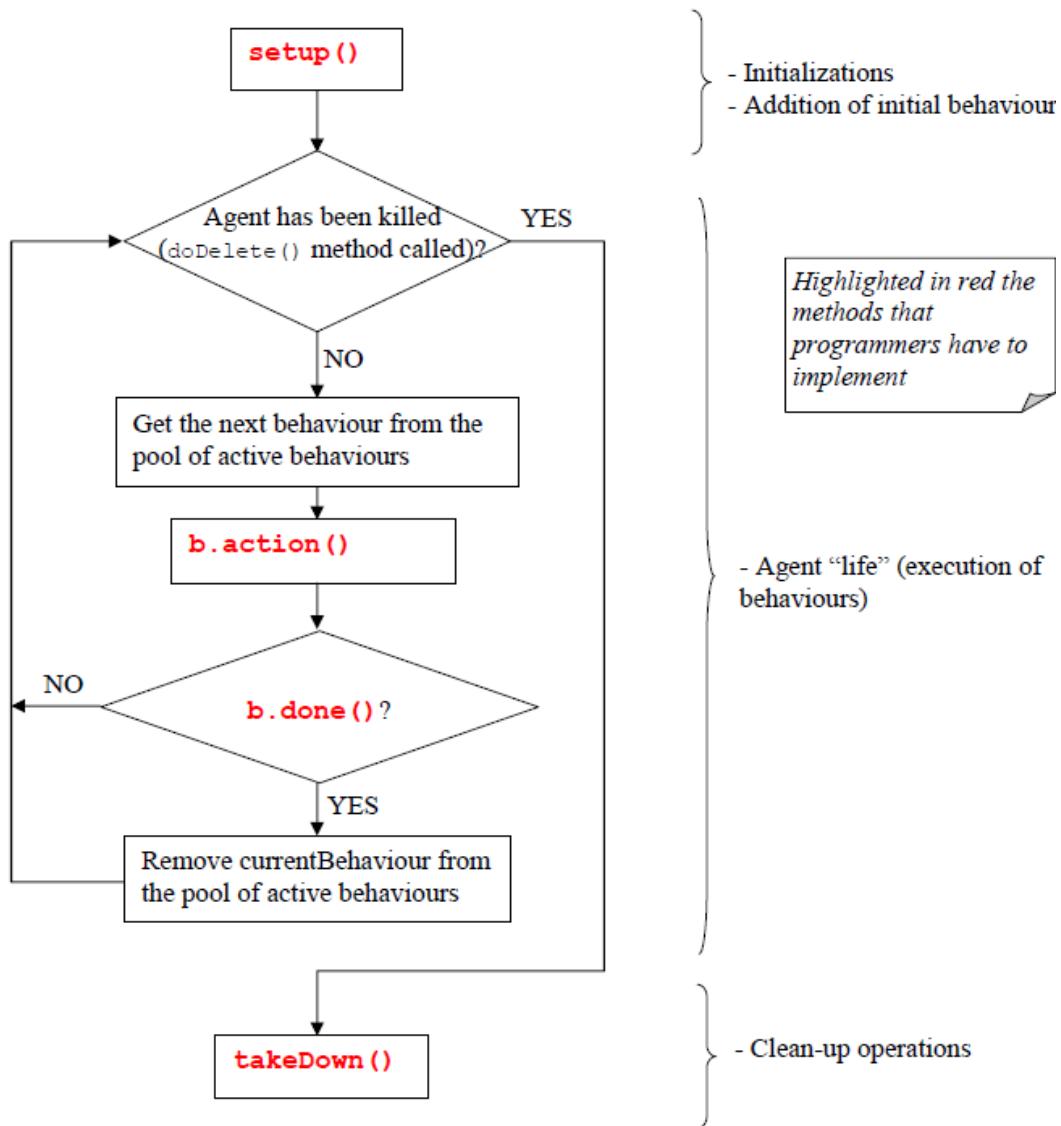


Figura 22 - Caminho de execução *thread* de Agente (Bellifemine et al., 2010)

Um agente pode executar vários comportamentos simultaneamente. No entanto, é importante notar que a programação de comportamentos em um agente não é preemptiva (como para *thread* Java), mas cooperativa. Isto significa que, quando um comportamento está agendado para execução seu método *action()* é chamado e é executado até que ele retorne. Por isso, é o programador que define quando um agente troca da execução de um comportamento para a execução do próximo. Embora exigindo um pequeno esforço adicional para os programadores, esta abordagem tem várias vantagens, a saber:

- Permite ter uma única thread Java por agente (isto é muito importante, especialmente em ambientes com recursos limitados, tais como telefones celulares).
- Fornece melhor desempenho uma vez que a troca de comportamento é extremamente mais rápida do que a troca de thread Java.
- Elimina todos os problemas de sincronização entre comportamentos simultâneos, acessando os mesmos recursos já que todos os comportamentos são executados pelo mesmo thread Java.

Quando uma troca de comportamento ocorre o status de um agente não inclui qualquer informação empilhada e por isso é possível fazer um *snapshot* do mesmo. Isso torna possível a implementação de importantes características avançadas, como por exemplo, salvar o status de um agente em um armazenamento persistente para a retomada mais tarde (persistência de agente) ou transferência para outro *container* para a execução remota (mobilidade de agente).

Comportamentos de agente podem ser descritos como máquinas de estados finitas, mantendo seu estado com um todo em suas variáveis de instância. Lidar com os comportamentos de agente complexos (como protocolos de interação entre agentes) usando variáveis de estado explícitas pode ser complicado, por isso JADE também suporta uma técnica de composição para construir comportamentos mais complexos. O *framework* fornece subclasses de *Behaviour* prontas para uso, que podem conter subcomportamentos e executá-los de acordo com alguma política. Por exemplo, uma classe *SequentialBehaviour* é fornecida, que executa os seus subcomportamentos um após o outro para cada invocação de *action()*.

A Figura 23, a seguir, é um diagrama de classes UML anotado para os comportamentos em JADE.

A classe abstrata *Behaviour* fornece uma base para modelagem de tarefas de agente, e define a base para a programação do comportamento, pois permite transições de estado (ou seja, iniciar, bloquear e reiniciar um objeto de comportamento Java).

Um comportamento bloqueado pode continuar a execução quando uma das seguintes condições ocorre: uma mensagem de ACL é recebida pelo agente a que este comportamento pertence; um tempo limite associado a este

comportamento por uma chamada anterior a *block()* expira; ou o método *restart()* é chamado explicitamente sobre esse comportamento.

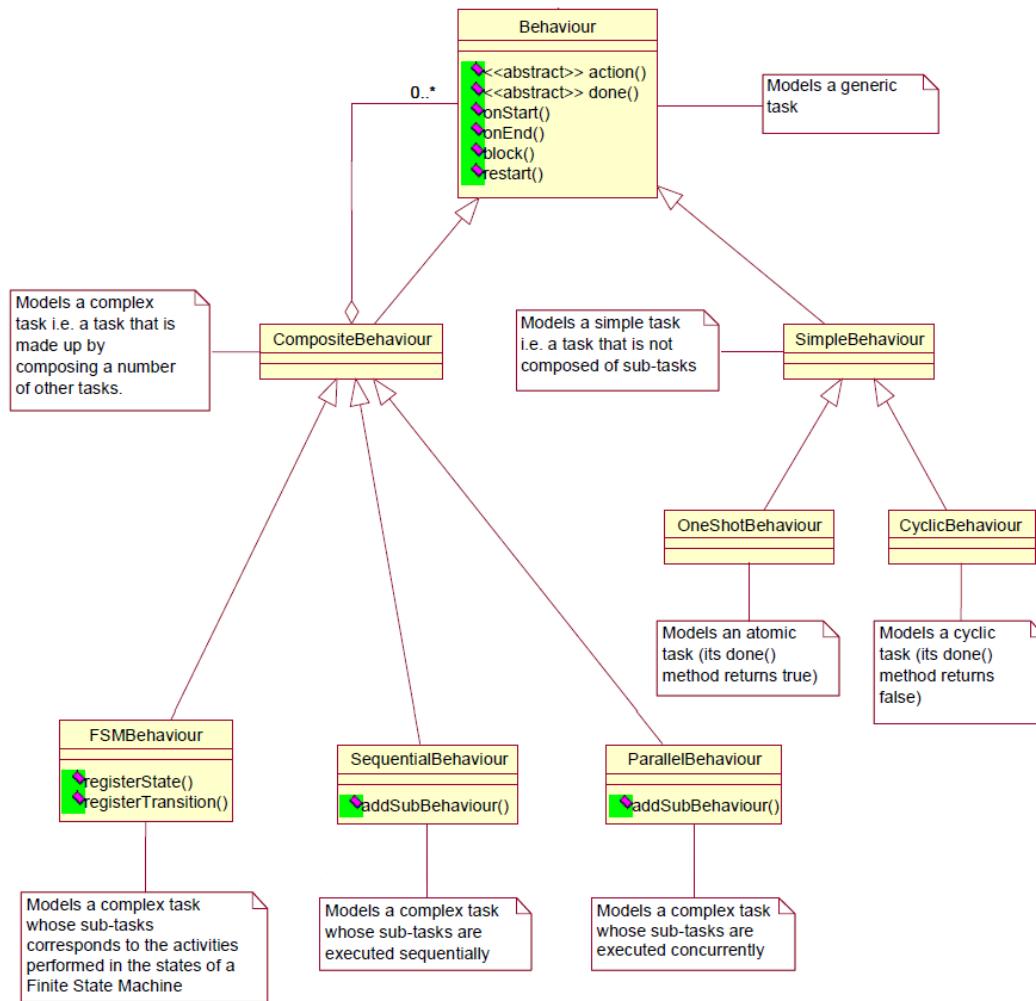


Figura 23 - Modelo UML da hierarquia *Behaviour* (Bellifemine et al., 2010)

A classe *Behaviour* também fornece dois métodos abstratos, chamados *onStart()* e

onEnd(). Estes métodos podem ser substituídos por subclasses definidas pelo usuário quando algumas ações estão para serem executadas antes e depois da execução do comportamento - *onEnd()* retorna um inteiro que representa um valor de término para o comportamento.

A classe *SimpleBehaviour* modela comportamentos atômicos simples. Seu método *reset()* não faz nada por *default*, mas pode ser sobreescrito (*overridden*) por uma subclass definida pelo usuário.

A classe *OneShotBehaviour* modela comportamentos atômicos que devem ser executados apenas uma vez e não podem ser bloqueados. Portanto, seu método *done()* sempre retorna *true*.

A classe *CyclicBehaviour* modela comportamentos atômicos que devem ser executados continuamente. Portanto, seu método *done()* sempre retorna *false*.

A classe *CompositeBehaviour* modela comportamentos que são feitos pela composição de um número de outros comportamentos (filhos). Portanto, as reais operações decorrentes da execução deste comportamento não são definidas nele mesmo, mas dentro de seus filhos enquanto *CompositeBehaviour* apenas toma conta dos filhos fazendo o agendamento de acordo com uma determinada política. Em particular a classe *CompositeBehaviour* apenas fornece uma interface comum para o agendamento dos filhos, mas não define uma política de agendamento. Esta política de agendamento deve ser definida por uma subclasse (*SequentialBehaviour*, *ParallelBehaviour* e *FSMBehaviour*).

SequentialBehaviour é uma subclasse de *CompositeBehaviour* que executa seus subcomportamentos sequencialmente e termina quando todos os seus subcomportamentos tiverem terminado. Esta classe deve ser usada quando uma tarefa complexa puder ser expressa com uma sequência atômica de passos (ex: fazer alguma computação, então receber uma mensagem, e então fazer alguma outra computação).

ParallelBehaviour é a subclasse de *CompositeBehaviour* que executa seus sub comportamentos simultaneamente e termina quando uma condição particular em seus sub comportamentos for atendida. Constantes designadas no construtor desta classe são fornecidas ao criar uma instância de *ParallelBehaviour* que pode terminar quando todos os seus subcomportamentos terminarem, quando qualquer um entre os seus subcomportamentos terminar, ou quando um número *N* definido pelo desenvolvedor de seus subcomportamentos terminar. Esta classe deve ser usada quando uma tarefa complexa puder ser expressa como uma coleção de operações alternativas paralelas, com algum tipo de condição de terminação nas subtarefas geradas.

FSMBehaviour é a subclasse de *CompositeBehaviour* que executa seus filhos de acordo com uma máquina de estados finita (*Finite State Machine*) definida pelo usuário. Mais detalhadamente, cada filho representa a atividade a ser realizada dentro de um estado da *FSM* e o usuário pode definir as transições entre os estados da *FSM*. Quando um filho correspondente ao estado *S_i* for concluído, o seu valor de término (como retornado pelo método *onEnd()*) é utilizado para selecionar a transição a ser acionada e um novo estado *S_j* é alcançado. Na próxima rodada o filho correspondente ao estado *S_i* será

executado. Alguns dos filhos de um *FSMBehaviour* podem ser registrados como estados finais. O *FSMBehaviour* termina após a execução de uma destes filhos.

4.2.

Mapeamento de elementos básicos de i* para JADE

A Figura 24 apresenta novamente o modelo i* para o sistema multiagente para controle da porta automática, com marcações numéricas para auxiliar o entendimento do mapeamento entre os elementos de i* para JADE explicado a seguir.

Como JADE é implementado em Java, nosso *framework* foi desenvolvido na mesma linguagem. As Figura 25 e 26 abaixo apresentam diagramas de classe UML dos principais elementos de i* implementados no *framework* *iStarJade*.

A classe *IstarJadeAgent* é uma especialização da classe *jade.core.Agent* e implementa a interface *IAgent* (que representa agentes em i*) e possui um comportamento composto do tipo máquina de estados finita (*FSMBehaviour*).

Para os elementos intencionais de i* (*goal*, *softgoal*, *task* e *resource* em inglês) foram criadas as interfaces Java *IGoal*, *ISoftgoal*, *ITask* e *IResource*. Essas interfaces são implementadas respectivamente pelas classes Java *Goal*, *Softgoal*, *Task* e *Resource*. Essas classes, por sua vez, são especializadas pelas classes básicas *BasicGoal*, *BasicSoftgoal*, *BasicTask* e *BasicResource*.

Os elementos básicos devem ter uma classe que implemente a interface *IBasicElement*, que possui o método *getBehaviour()*: *jade.core.Behaviour* e que será carregada via Java Reflection - a API Reflection (McManis, 1997) é comumente usada por programas que requerem a habilidade para examinar ou modificar em tempo de execução o comportamento de aplicações rodando na *Java Virtual Machine* (JVM). De forma análoga, os elementos principais, tipicamente metas (classe *goal*), devem ter uma classe que implemente a interface *IMainGoal* que também possui o método *getBehaviour()*: *jade.core.Behaviour* e que será carregada via Java Reflection.

A Figura 26 apresenta o mapeamento dos elos entre elementos intencionais (*ielementLink*) e os comportamentos que são instanciados automaticamente nos agentes (*IStarJadeAgent*).

Os elos *means-end* (que representam a seleção de alternativas – “ou” lógico) são mapeados para comportamentos *MeansEndUniqueBehaviour* ou *MeansEndParallelBehaviour*. Estes comportamentos determinam a alternativa a

ser escolhida. Uma vez escolhida a alternativa, o comportamento correspondente ao elemento é instanciado. Os elos *decomposition* (que representam decomposição de tarefas - “e” lógico) são mapeados em comportamentos *SequentialTaskBehaviour* ou *SequentialParallelBehaviour*. Os elos sequenciais impõem uma ordem na execução dos subelementos – que é a ordem que os elos para esses subelementos aparecem no arquivo iStarML estendido.

A Tabela 15 apresenta a correspondência entre os componentes (numerados) do modelo i* e as classes do framework iStarjade. Os números são os mesmos usados nas figuras 24, 25 e 26 para auxiliar o entendimento.

Ainda na Figura 26, destacamos as classes *Context*, que especializa a classe *Task*, e *ArgumentationLink* (cor amarela no diagrama). Estas classes correspondem ao elemento Contexto e ao elo de argumentação respectivamente.

Componente em i*	Tag iStarML	Classe/atributo em iStarjade
1 – meta principal	<ielement type="goal" main="true" />	MainGoal
2 – elo meios-fim (escolha única)	<ielementlink type="means-end" run-mode="unique" />	MeansEndUniqueBehaviour
3 – tarefa	<ielement type="task" />	Task
4 – elo de decomposição de tarefa (sequencial)	<ielementlink type="task-decomposition" run-mode="sequential" />	TaskDecompositionBehaviour
5 – meta flexível	<ielement type="softgoal"/>	Softgoal
6 – context	<ielement type="context-awareness"/>	ContextAwareness
7 – recurso básico	<ielement type="resource" basic="true" />	BasicResource
8 – tarefa básica	<ielement type="task" basic="true" />	BasicTask
9 – elo de argumentação	<ielementlink type="argumentation" situation-name="string" />	ArgumentationLink
10 - meta	<ielement type="goal" />	Goal
11 – depender	<dependency> <depender iref="string"/> </dependency>	IDependency.dependerElement
12 – resource	<ielement type="resource"/>	Resource
13 – dependee	<dependency> <dependee iref="string" /> </dependency>	IDependency.dependeeElement
14 – agente	<actor type="agent" />	IstarJadeAgent
15 – elo desempenha	<actorlink type="plays" />	IAgent.getPlays(): List<IRole>
16 – papel	<actor type="role" />	Role

Tabela 15 – Mapeamento entre elementos de i*, iStarML e Jade

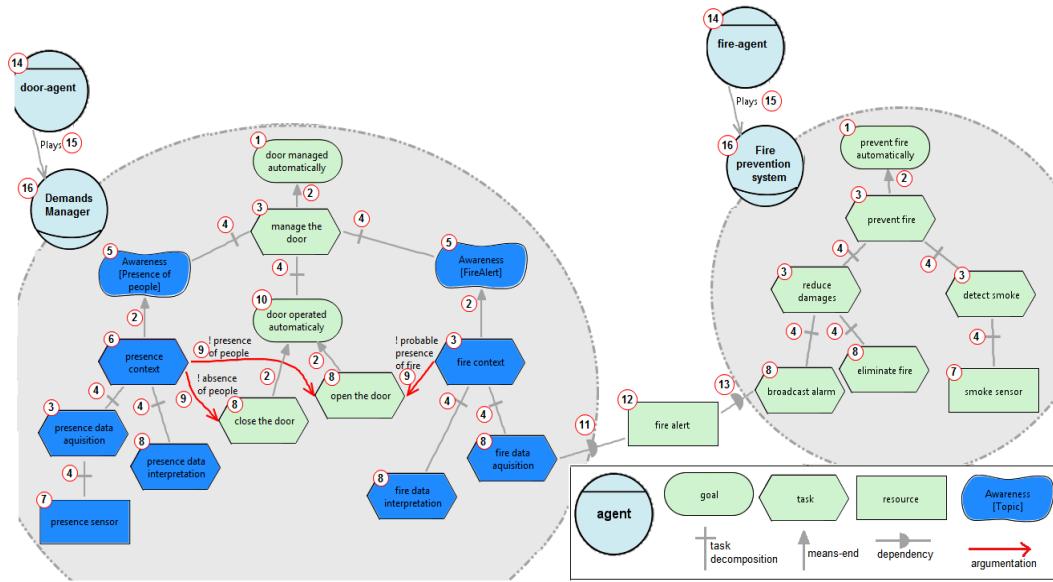


Figura 24 - Diagrama do SMA para controle de porta

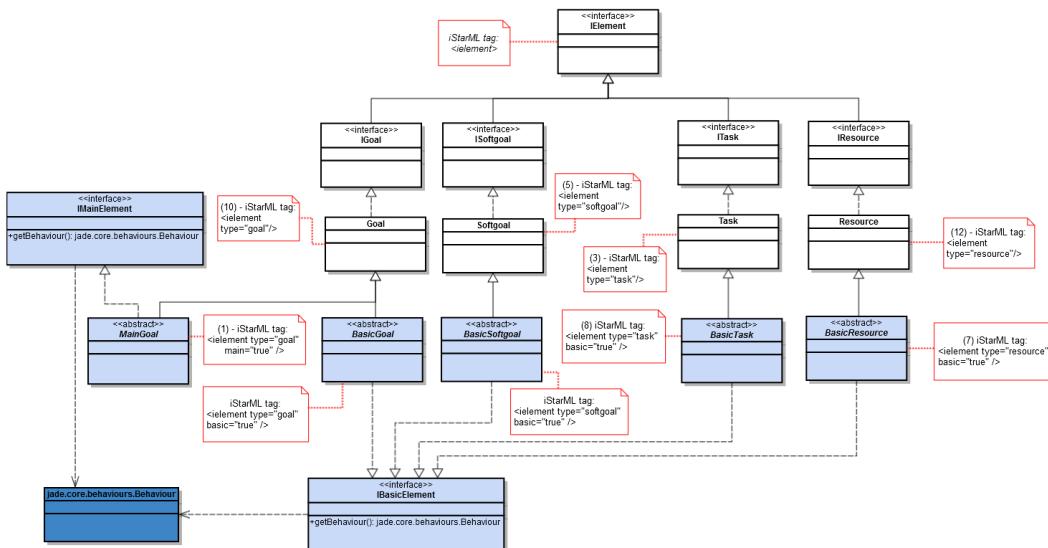


Figura 25 - Diagrama de classe para os elementos intencionais

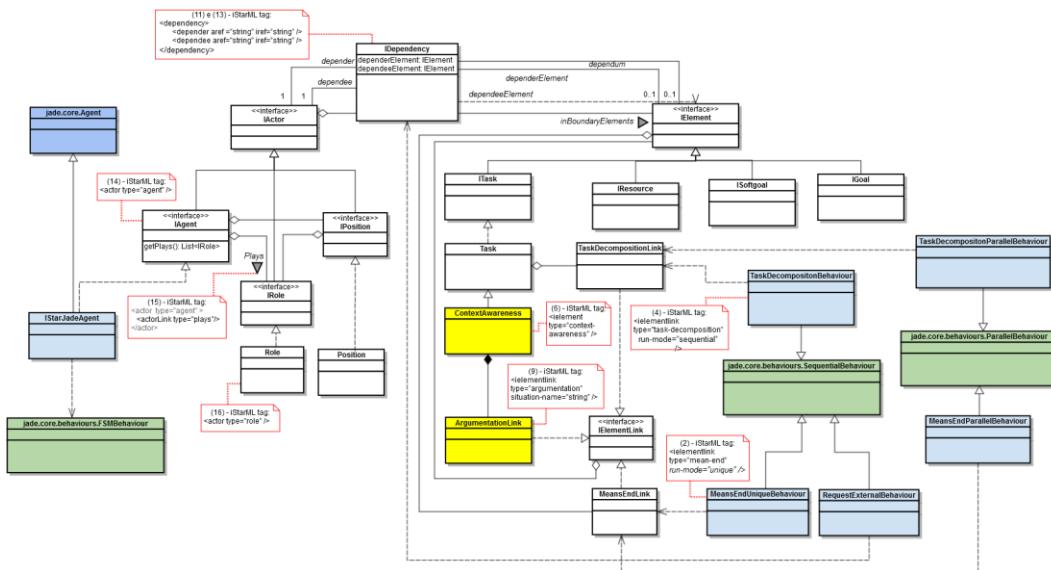


Figura 26 - Diagrama de classe UML, com atores, elos e dependência

4.2.1. Acréscimo de atributos em iStarML

Para a implementação do *framework istarjade* adicionamos alguns atributos na descrição dos modelos em iStarML que não existem correspondentes em i*, além dos apresentados no capítulo de modelagem, para direcionar alguns aspectos necessários à implementação.

Em i*, quando há uma decomposição de tarefa não há nenhuma indicação da ordem de execução dos subelementos da tarefa decomposta. Na resolução de inúmeros problemas práticos, uma subparte de uma tarefa gera um resultado intermediário que servirá de entrada para a subparte seguinte, impondo uma ordem na execução das subpartes da tarefa. Para outros problemas, as subpartes podem ser executadas paralelamente, sem prejuízo para o resultado final. Para sinalizar o modo de execução das subpartes lançamos mão do atributo *run-mode* que pode ter um dos dois valores: “*sequential*” ou “*parallel*”. Este atributo definirá o comportamento a ser carregado para execução pelo agente. Caso o atributo *run-mode* tenha o valor “*sequential*” os comportamentos correspondentes às subtarefas são alocados em uma instância da classe *TaskDecomposition* (que é uma especialização da classe *SequentialBehaviour* de JADE). Caso o atributo *run-mode* tenha o valor “*parallel*”, no caso do elo de decomposição de tarefas os comportamentos correspondentes às subtarefas são alocados em uma instância da classe *TaskDecompositionParallel* (que é uma especialização da classe *ParallelBehaviour* de JADE).

Também acrescentamos um novo atributo aos elos meios-fim. Um refinamento através dos elos meios-fim representa alternativas (meios) para que se possa alcançar o fim desejado. Ocorre que há casos em que todas as alternativas podem ser tentadas (de forma paralela ou não) e há casos que por alguma restrição, em geral por causa do consumo de algum recurso que não pode ser feito de forma concorrente nem simultânea (por exemplo: quantidade de dinheiro limitada para aquisição de algum bem), é necessário escolher uma dentre as alternativas existentes. Assim, acrescentamos a possibilidade de ter o valor “*unique*” para o atributo *run-mode* a ser utilizado nos elos meios-fim. Caso o atributo *run-mode* tenha o valor “*unique*”, apenas o comportamento correspondente a um dos meios será selecionado e alocado em uma instância da classe *MeansEndUniqueBehaviour* (que é uma especialização da classe *SequentialBehaviour* de JADE). A escolha do meio é baseada no valor de contribuição desse meio para o fim designado no elo meios-fim. Caso o atributo *run-mode* tenha o valor “*parallel*”, os comportamentos correspondente a todos os meios serão alocados em uma instância da classe *MeansEndParallelBehaviour* (que é uma especialização da classe *ParallelBehaviour* de JADE).

A Tabela 16 apresenta a especificação de iStarML com o acréscimo deste atributo.

<i>ielementLinkTag</i> ::=	<pre><ielementLink <i>linkAtts</i> [<i>runAtts</i>]> [graphic-path] <i>ielementTag</i> {<i>ielementTag</i>} </ielementLink></pre>
<i>linkAtts</i> ::=	<pre>type = “decomposition” [value=(“and” “or”)] type=“means-end” [value=“<i>string</i>”] type=“contribution” [value=“<i>contribution-value</i>”] type=“<i>string</i>” [value=“<i>string</i>”]</pre>
<i>contribution-value</i> ::=	<pre>+ - sup sub ++ -- break hurt some- some+ unknown equal help make and or</pre>
<i>runAtts</i> ::=	<pre>run-mode=“sequential” run-mode=“parallel” run-mode=“unique”</pre>

Tabela 16 - Sintaxe para elos entre elementos intencionais modificada

Além destes atributos, acrescentamos mais dois atributos às *tags* dos elementos intencionais (*<ielement>*) para direcionar outro aspecto da implementação em nosso *framework*, que é permitir a criação dinâmica de

comportamentos dos agentes. Para isto, precisamos distinguir os elementos que teriam os seus comportamentos carregados dinamicamente via Java Reflection. Dois tipos de elementos podem ter seus comportamentos carregados dinamicamente:

- elementos básicos (nós folha no racional de ator - nós que não são compostos nem são fins de elos meios-fim, nem dependem de nenhum elemento externo) – para distinguir os elementos básicos na descrição em iStarML adicionamos o atributo *basic="true"*.
- elementos principais (nós raiz no racional do ator – em geral estes nós representam as metas que um ator tem, numa abordagem BDI, embora em i* esse nó também possa ser uma tarefa ou recurso) – para distinguir os elementos principais na descrição em iStarML adicionamos o atributo *main="true"*.

Com isso, os comportamentos dos agentes no *framework* serão de uma destas categorias:

- Comportamentos principais – carregados via Java Reflection, geralmente estão relacionados às metas principais do agente. São os comportamentos iniciais dos agentes, a partir dos quais os demais comportamentos serão adicionados após sua finalização.
- Comportamentos de elos meios-fim – são comportamentos compostos e determinam quais comportamentos relativos aos meios serão carregados para execução pelos agentes, podendo ser do tipo escolha de um único meio para execução, ou de vários meios executando paralelamente.
- Comportamentos de decomposição de tarefas – também são comportamentos compostos que determinam como os comportamentos relativos às subtarefas serão carregados para execução pelos agentes, podendo ser de forma sequencial ou paralela.

- Comportamentos básicos – também carregados via Java Reflection, representam os comportamentos que serão adicionados aos comportamentos compostos dos elos meios-fim e dos elos de decomposição de tarefas.

A seguir, descrevemos como acontece a adição de comportamentos ao se iniciar um agente em istarjade.

4.3.

Criação de agentes e adição de comportamentos

O framework iStarJade tem uma classe de agente chamada *Loader* que especializa a classe *jade.core.Agent*, que tem a capacidade de criar agentes da classe *IstarJadeAgent* a partir de um modelo i* descrito em iStarML. Este agente da classe *Loader* obtém uma instância da classe *IstarModelLoader* (classe responsável por fazer a varredura do arquivo iStarML e criar os elementos, do diagrama SR, de cada agente) e passa o caminho do arquivo como parâmetro para o método que iniciará o procedimento de varredura do arquivo, criação e inicialização dos agentes. Esta primeira etapa é ilustrada nas Figuras I9, e I10, apresentadas a seguir.

Na Figura 27 temos a configuração dos parâmetros para execução de istarjade no Eclipse (IDE para JAVA). Na caixa de texto para os argumentos do programa (*Program arguments*), o parâmetro -agents loader_agente:laoder.Loader tem o objetivo de criar um agente de nome loader_agent do tipo loader.Loader.

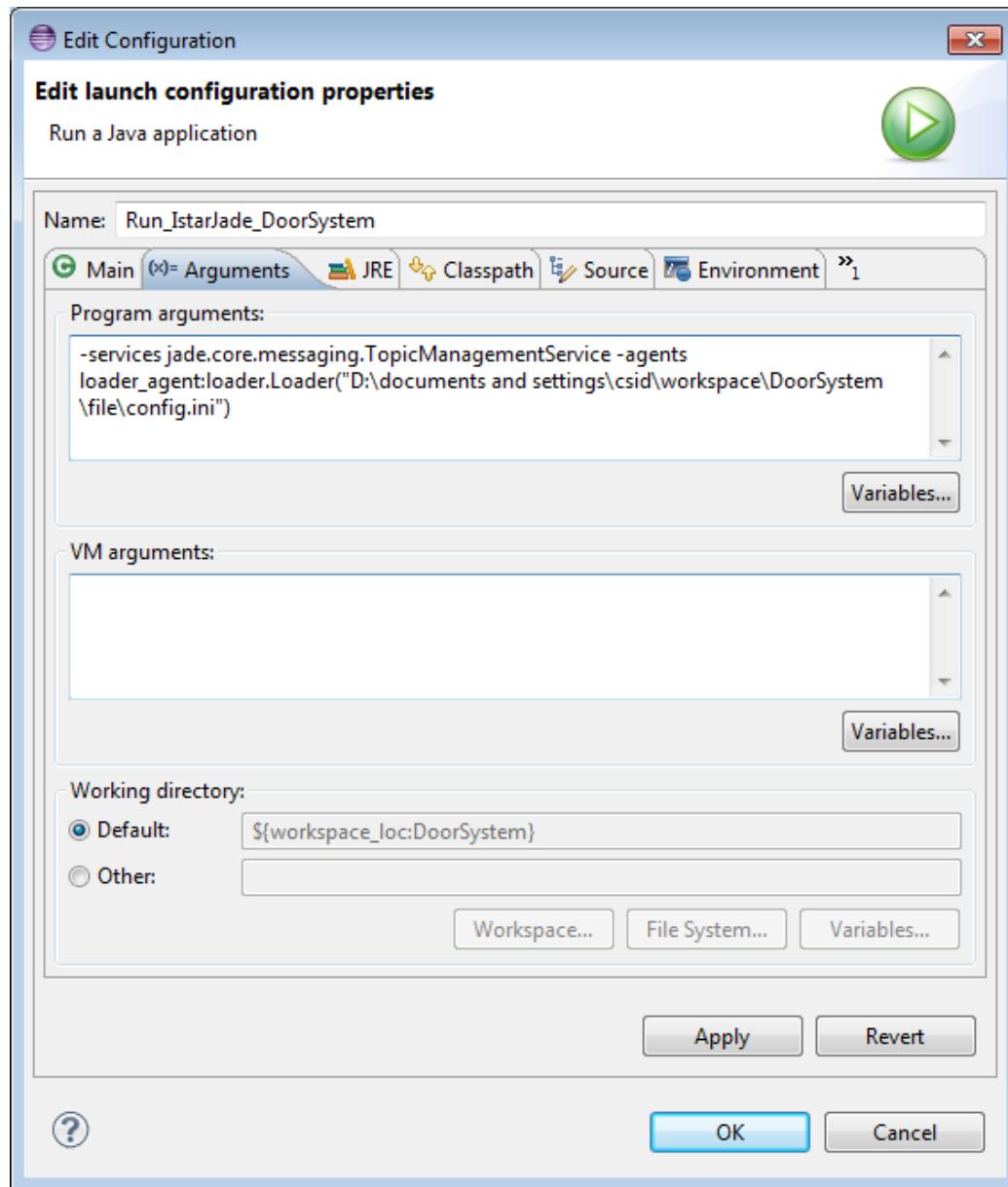


Figura 27 - Parâmetros para execução do framework istarjade

A Figura 28, a seguir, apresenta a GUI de JADE com o agente *loader_agent* criado juntamente com os agentes *amf*, *dfe* e *rma*.

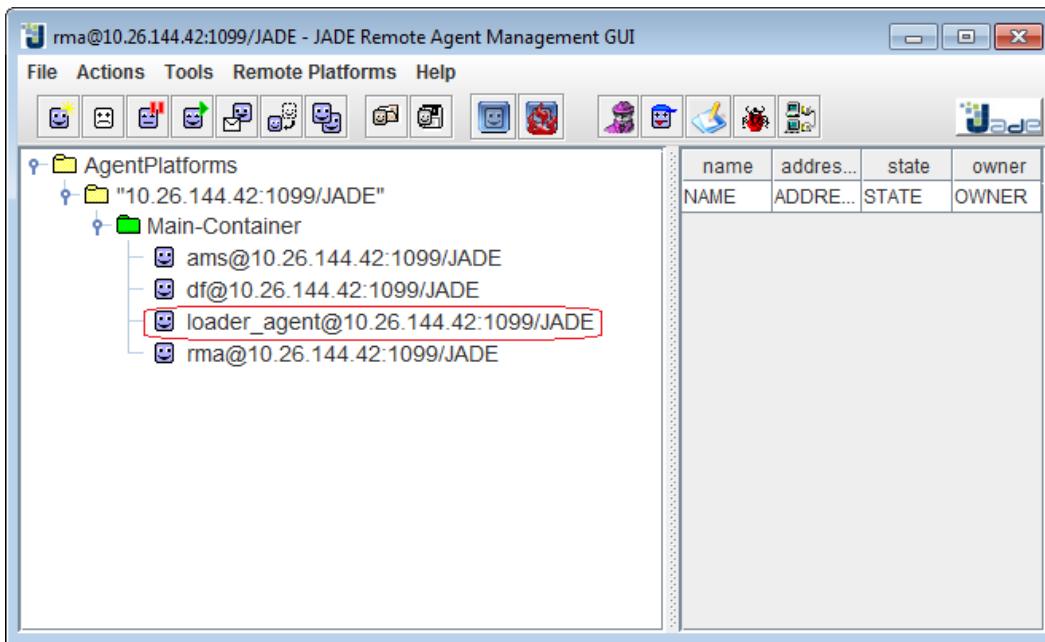


Figura 28 - GUI de JADE após criação do agente loader_agent

Uma descrição sucinta da criação dos agentes é apresentada a seguir:

i – é feito um *parsing* no arquivo iStarML, obtendo uma lista de atores e os elos de ligação entre atores – ou seja, são obtidos os diversos tipos de atores que podem aparecer dentro de um par de *tags* <diagram> de iStarML, com seus respectivos elos de ligação. A Figura 29, a seguir, apresenta um método da classe IstarModelLoader, obtendo os elementos mais externos no arquivo iStarML.

ii – para cada ator, é criada uma lista com os elementos internos a fronteira do ator, juntamente com os elos entre elementos intencionais – ou seja, são obtidos os elementos que podem aparecer dentro de um par de *tags* <actor> de iStarML. Neste ponto, o modelo i* com o racional de cada ator e suas relações de dependência foi obtido do arquivo iStarML e foram criados os objetos correspondentes de classes do framework . A Figura 30 e Figura 31, a seguir, mostram trechos de código onde os elementos básicos e elementos principais são carregados.

iii – é criada uma lista de elementos de dependência existentes no diagrama, a partir da qual as dependências são adicionadas aos atores. Se houver referência ao elemento de dependência (atributo *iref* no arquivo iStarML), a dependência é acrescentada ao elemento especificado na referência. A Figura 32 apresenta trecho de código referente a esta etapa.

iv – para cada agente no diagrama, são criados os agentes *IstarJadeAgent* e na inicialização (*setup()*) de cada agente os seguinte procedimentos são executados:

- a) Cada agente que atua como *dependee* em uma relação de dependência, seja diretamente, seja desempenhando um papel ou ocupando uma posição, registra-se no DF (páginas amarelas), tendo os elementos de dependência como serviços que este agente presta;
- b) Cada agente que atua como *depende* em uma relação de dependência, seja diretamente, seja desempenhando um papel ou ocupando uma posição, registra-se no DF (páginas amarelas) como ouvinte de mensagens sobre os serviços relacionados aos elementos de dependência dos quais este agente depende;
- c) São identificados os elementos principais (geralmente metas) no *rationale* de cada ator. As classes destes elementos principais são adicionadas aos respectivos agentes via Java *Reflection*. Tanto o comportamento principal, quanto os comportamentos dos demais elementos vão sendo adicionados ao comportamento composto do tipo máquina de estados finita (*FSMBehaviour*) que o agente *IStarJadeAgent* possui.
- d) A partir dos elementos principais, os comportamentos referentes aos elementos do racional de cada ator vão sendo adicionados recursivamente aos agentes até que se chegue a um elemento básico (nó folha no modelo SR).
- e) As classes destes elementos básicos são adicionadas aos respectivos agentes via Java *Reflection*.
- f) Finalmente, os comportamentos dos elementos principais são iniciados.

A Figura 33 mostra o trecho de código do método *setup()* da classe *IstarJadeAgent*, onde os passos a), b), c), d) e e) descritos acima.

```

109 /**
110 * Method for obtain first level elements from iStarML diagram
111 * @param _pathToFile: path to iStarML file
112 */
113 private void loadIstarMLModel(String _pathToFile) throws Exception {
114     File istarMLFile;
115     try {
116         istarMLFile = new File(_pathToFile);
117         DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
118         DocumentBuilder db = dbf.newDocumentBuilder();
119         Document doc = db.parse(istarMLFile);
120         doc.getDocumentElement().normalize();
121         // Obtaining elements from iStarML diagram first level
122         NodeList ndListfile= doc.getDocumentElement().getChildNodes();
123         for (int i = 0; i < ndListfile.getLength(); i++) {
124             Node ndfile = ndListfile.item(i);
125             if (ndfile.getNodeType() == Node.ELEMENT_NODE) {
126                 NodeList ndLst = ndfile.getChildNodes();
127                 // Obtaining ielements (dependum elements in strategic dependencies)
128                 for (int j = 0; j < ndLst.getLength(); j++) {
129                     Node elmntNode = ndLst.item(j);
130                     if (elmntNode.getNodeType() == Node.ELEMENT_NODE) {
131                         Element elmnt = (Element)elmntNode;
132                         String elementType = elmnt.getTagName();
133                         if (elementType.equals("actor")){
134                             loadActor(elmnt);
135                         }
136                     }
137                 }
138                 // Obtaining ielements (dependum elements in strategic dependencies)
139                 for (int m = 0; m < ndLst.getLength(); m++) {
140                     Node elmntNode = ndLst.item(m);
141                     if (elmntNode.getNodeType() == Node.ELEMENT_NODE) {
142                         Element elmnt = (Element)elmntNode;
143                         if (elmnt.getTagName().equals("ielement")){
144                             loadExternalElements(elmnt);
145                         }
146                     }
147                 }
148             }
149             for (int i = 0; i < ndListfile.getLength(); i++) {
150                 Node ndfile = ndListfile.item(i);
151                 if (ndfile.getNodeType() == Node.ELEMENT_NODE) {
152                     NodeList ndLst = ndfile.getChildNodes();
153                     // Obtaining actor links
154                     for (int j = 0; j < ndLst.getLength(); j++) {
155                         Node elmntNode = ndLst.item(j);
156                         if (elmntNode.getNodeType() == Node.ELEMENT_NODE) {
157                             Element elmnt = (Element)elmntNode;
158                             String elementType = elmnt.getTagName();
159                             if (elementType.equals("actor")){
160                                 loadActorElementLinks(elmnt);
161                             }
162                         }
163                     }
164                 }
165             }
166         }
167     } catch (Exception e) {
168         e.printStackTrace();
169     }
170 }
171 }
172 }
```

Figura 29 - Método *loadIstarModel()* da classe IstarModelLoader

```

338 /**
339 * Method for load actors SR elements
340 * @param subElmntNode: iStarML element
341 * @param actor: generic i* actor
342 */
343 private void loadInBoundaryElements(Element subElmntNode, IActor actor) throws Exception{
344     IElement internalElmnt=null;
345     // checking if it is a basic element
346     if (subElmntNode.getAttribute("basic").equals("true")){
347         Class<?> elementClass;
348         // checking if the class has to be loaded from a url
349         if ((subElmntNode.getAttribute("urlClass")!="")&&(subElmntNode.getAttribute("behaviourClass")!="")){
350             URL[] urls = new URL[]{ new URL(subElmntNode.getAttribute("urlClass")) };
351             ClassLoader loader = new URLClassLoader(urls);
352             // getting the basic class from url, by JAVA Reflection
353             elementClass = loader.loadClass(subElmntNode.getAttribute("behaviourClass"));
354         }
355         else
356             // getting the basic class the project, by JAVA Reflection
357             elementClass = getBasicClass(subElmntNode.getAttribute("name"));
358
359         // checking if it is a basic task
360         if (subElmntNode.getAttribute("type").equals("task")){
361             try {
362                 // assigning a basic task to the internalElement, by JAVA Reflection
363                 internalElmnt = (AbstractBasicTask)elementClass.newInstance();
364             } catch (Exception e) {
365                 throw e;
366             }
367         }
368         // checking if it is a basic resource
369         if (subElmntNode.getAttribute("type").equals("resource")){
370             try {
371                 // assigning a basic resource to the internalElement, by JAVA Reflection
372                 internalElmnt = (AbstractBasicResource)elementClass.newInstance();
373             } catch (Exception e) {
374                 throw e;
375             }
376         }
377         // checking if it is a basic goal
378         if (subElmntNode.getAttribute("type").equals("goal")){
379             try {
380                 // assigning a basic goal to the internalElement, by JAVA Reflection
381                 internalElmnt = (AbstractBasicGoal)elementClass.newInstance();
382             } catch (Exception e) {
383                 throw e;
384             }
385         }
386         // checking if it is a basic softgoal
387         if (subElmntNode.getAttribute("type").equals("softgoal")){
388             try {
389                 // assigning a basic softgoal to the internalElement, by JAVA Reflection
390                 internalElmnt = (AbstractBasicSoftgoal)elementClass.newInstance();
391             } catch (Exception e) {
392                 throw e;
393             }
394         }
395         // checking if it is a basic context-awareness
396         if (subElmntNode.getAttribute("type").equals("context-awareness")){
397             try {
398                 // assigning a basic context-awareness to the internalElement, by JAVA Reflection
399                 internalElmnt = (AbstractBasicContextAwareness)elementClass.newInstance();
400             } catch (Exception e) {
401                 throw e;
402             }
403         }
404     }

```

Figura 30 - Trecho de código - elementos básicos são carregados.

```

427 // checking if it is a goal
428 if (subElmntNode.getAttribute("type").equals("goal")){
429     // checking if it is a main goal
430     if (subElmntNode.getAttribute("main").equals("true")){
431         // checking if the class has to be loaded from a url
432         if ((subElmntNode.getAttribute("urlClass")!="")&&(subElmntNode.getAttribute("behaviourClass")!="")){
433             URL[] urls = new URL[] { new URL(subElmntNode.getAttribute("urlClass")) };
434             ClassLoader loader = new URLClassLoader(urls);
435             Class<?> goalClass = loader.loadClass(subElmntNode.getAttribute("behaviourClass"));
436             // assigning a main goal to the internalElement from url, by JAVA Reflection
437             internalElmnt = (AbstractMainGoal)goalClass.newInstance();
438         }
439     }
440     else {
441         // assigning a main goal from the project, by JAVA Reflection
442         internalElmnt = (AbstractMainGoal)getMainElementClass(subElmntNode.getAttribute("name")).newInstance();
443     }
444 }
445 else {
446     // assigning a regular goal
447     internalElmnt = new Goal(subElmntNode.getAttribute("name"), actor);
448 }
449 // checking if it is a softgoal
450 if (subElmntNode.getAttribute("type").equals("softgoal")){
451     // checking if it is a main softgoal
452     if (subElmntNode.getAttribute("main").equals("true")){
453         if ((subElmntNode.getAttribute("urlClass")!="")&&(subElmntNode.getAttribute("behaviourClass")!="")){
454             URL[] urls = new URL[] { new URL(subElmntNode.getAttribute("urlClass")) };
455             ClassLoader loader = new URLClassLoader(urls);
456             Class<?> softgoalClass = loader.loadClass(subElmntNode.getAttribute("behaviourClass"));
457             // assigning a main softgoal to the internalElement from url, by JAVA Reflection
458             internalElmnt = (AbstractMainSoftgoal)softgoalClass.newInstance();
459         }
460     }
461     else {
462         // assigning a main softgoal from the project, by JAVA Reflection
463         internalElmnt = (AbstractMainSoftgoal)getMainElementClass(subElmntNode.getAttribute("name")).newInstance();
464     }
465 }
466 else {
467     // assigning a regular softgoal
468     internalElmnt = new Softgoal(subElmntNode.getAttribute("name"), actor);
469 }
470 // checking if it is a resource
471 if (subElmntNode.getAttribute("type").equals("resource")){
472     // checking if it is a main resource
473     if (subElmntNode.getAttribute("main").equals("true")){
474         // checking if the class has to be loaded from a url
475         if ((subElmntNode.getAttribute("urlClass")!="")&&(subElmntNode.getAttribute("behaviourClass")!="")){
476             URL[] urls = new URL[] { new URL(subElmntNode.getAttribute("urlClass")) };
477             ClassLoader loader = new URLClassLoader(urls);
478             Class<?> resourceClass = loader.loadClass(subElmntNode.getAttribute("behaviourClass"));
479             // assigning a main resource to the internalElement from url, by JAVA Reflection
480             internalElmnt = (AbstractMainResource)resourceClass.newInstance();
481         }
482     }
483     else {
484         // assigning a main resource from the project, by JAVA Reflection
485         internalElmnt = (AbstractMainResource)getMainElementClass(subElmntNode.getAttribute("name")).newInstance();
486     }
487 }
488 else {
489     // assigning a regular resource
490     internalElmnt = new Resource(subElmntNode.getAttribute("name"), actor);
491 }
492 // checking if it is a task
493 if (subElmntNode.getAttribute("type").equals("task")){
494     // checking if it is a main task
495     if (subElmntNode.getAttribute("main").equals("true")){
496         // checking if the class has to be loaded from a url
497         if ((subElmntNode.getAttribute("urlClass")!="")&&(subElmntNode.getAttribute("behaviourClass")!="")){
498             URL[] urls = new URL[] { new URL(subElmntNode.getAttribute("urlClass")) };
499             ClassLoader loader = new URLClassLoader(urls);
500             Class<?> taskClass = loader.loadClass(subElmntNode.getAttribute("behaviourClass"));
501             // assigning a main task to the internalElement from url, by JAVA Reflection
502             internalElmnt = (AbstractMainTask)taskClass.newInstance();
503         }
504     }
505     else {
506         // assigning a main task from the project, by JAVA Reflection
507         internalElmnt = (AbstractMainTask)getMainElementClass(subElmntNode.getAttribute("name")).newInstance();
508     }
509 }
510 else {
511     // assigning a regular task
512     internalElmnt = new Task(subElmntNode.getAttribute("name"), actor);
513 }
514 // checking if it is a context-awareness
515 if (subElmntNode.getAttribute("type").equals("context-awareness")){
516     // assigning a regular context-awareness
517     internalElmnt = new ContextAwareness(subElmntNode.getAttribute("name"), actor);
518 }

```

Figura 31 - Trecho de código - elementos principais são carregados

```

200  /**
201  * @param elmnt: iStarML element correspondent to dependum
202  * @throws Exception
203  */
204 private void loadExternalElements(Element elmnt) throws Exception {
205     IElement extIElmnt = null;
206     // checking if it is a task dependency
207     if (elmnt.getAttribute("type").equals("task")){
208         Task extTask = new Task(elmnt.getAttribute("name"));
209         extTask.setId(elmnt.getAttribute("id"));
210         extTask.setEstimatedWaitingTime(new Long(elmnt.getAttribute("waiting_time")));
211         if (!elmnt.getAttribute("cost").equals("")){
212             extTask.setCost(new Double(elmnt.getAttribute("cost")));
213         }
214         extIElmnt = extTask;
215     } // checking if it is a resource dependency
216     } else if (elmnt.getAttribute("type").equals("resource")){
217         Resource extResource = new Resource(elmnt.getAttribute("name"));
218         extResource.setId(elmnt.getAttribute("id"));
219         extResource.setEstimatedWaitingTime(new Long(elmnt.getAttribute("waiting_time")));
220         extIElmnt = extResource;
221     } // checking if it is a goal dependency
222     } else if (elmnt.getAttribute("type").equals("goal")){
223         Goal extGoal = new Goal(elmnt.getAttribute("name"));
224         extGoal.setId(elmnt.getAttribute("id"));
225         extGoal.setEstimatedWaitingTime(new Long(elmnt.getAttribute("waiting_time")));
226         extIElmnt = extGoal;
227     } // checking if it is a softgoal dependency
228     } else if (elmnt.getAttribute("type").equals("softgoal")){
229         Softgoal extSoftgoal = new Softgoal(elmnt.getAttribute("name"));
230         extSoftgoal.setId(elmnt.getAttribute("id"));
231         extSoftgoal.setEstimatedWaitingTime(new Long(elmnt.getAttribute("waiting_time")));
232         extIElmnt = extSoftgoal;
233     } // checking if it is a context-awareness dependency
234     } else if (elmnt.getAttribute("type").equals("context-awareness")){
235         IElement topic = null;
236         Iterator<IActor> itActors = actors.values().iterator();
237         while (itActors.hasNext()){
238             IActor act = itActors.next();
239             if (topic==null && elmnt.getAttribute("topicref")!= ""){
240                 topic = act.getInternalElement(elmnt.getAttribute("topicref"));
241             }
242         }
243         ContextAwareness context = new ContextAwareness(elmnt.getAttribute("name"), topic);
244         context.setId(elmnt.getAttribute("id"));
245         context.setEstimatedWaitingTime(new Long(elmnt.getAttribute("waiting_time")));
246         extIElmnt = context;
247     }

```

Figura 32 - Trecho de código - dependências são carregadas

```

174 /**
175 * Method for configure and initialize agents
176 */
177 protected void setup() {
178     Object[] args = getArguments();
179
180     if (args != null && args.length > 0) {
181         IActor actor = (IActor)args[0];
182         configurationPath = (String)args[1];
183         try {
184             // Getting the main elements
185             Iterator<IElement> itMainElmnts = actor.getMainElements().iterator();
186             while (itMainElmnts.hasNext()){
187                 IElement main = (IElement)itMainElmnts.next();
188                 IElement clone = (IElement)main.clone(this);
189             }
190             // Getting the dependencies in which the actor is the dependee
191             Iterator<IElement> itDependees = actor.getDependeeElements().iterator();
192             while (itDependees.hasNext()){
193                 IElement dependeeElmnt = itDependees.next();
194                 this.dependeeElements.add(dependeeElmnt);
195             }
196             // Getting the dependencies in which the actor is the depender
197             Iterator<IElement> itDependers = actor.getDependerElements().iterator();
198             while (itDependers.hasNext()){
199                 IElement dependerElmnt = itDependers.next();
200                 this.dependerElements.add(dependerElmnt);
201             }
202             // Getting positions and roles
203             if (actor.getClass().equals(IstarAgent.class)){
204                 Iterator<IPosition> itPositions = ((IstarAgent)actor).getOccupies().iterator();
205                 while (itPositions.hasNext()){
206                     this.addPosition(itPositions.next());
207                 }
208                 Iterator<IRole> itRoles = ((IstarAgent)actor).getPlays().iterator();
209                 while (itRoles.hasNext()){
210                     this.addRole(itRoles.next());
211                 }
212             }
213
214             // registering services as provider
215             List<ServiceDescription> listSD = actorServiceDescription(actor);
216             addServices(listSD);
217             // registering services as listener
218             addAsListener(actor);
219
220             List<IElement> mainElements = new Vector<IElement>();
221             mainElements.addAll(this.getMainElements());
222
223             // obtaining main behaviours from roles played
224             Iterator<IRole> itRoles = this.plays.iterator();
225             while (itRoles.hasNext()){
226                 IRole role = itRoles.next();
227                 mainElements.addAll(role.getMainElements());
228             }
229
230             // obtaining main behaviours from positions occupied
231             Iterator<IPosition> itPositions = this.getOccupies().iterator();
232             while (itPositions.hasNext()){
233                 IPosition position = itPositions.next();
234                 itRoles = position.getCovers().iterator();
235                 while (itRoles.hasNext()){
236                     IRole role = itRoles.next();
237                     mainElements.addAll(role.getMainElements());
238                 }
239             }
240             // setting the main behaviours
241             setMainBehaviour(mainElements);
242         } catch (Exception e) {
243             System.out.println(this.getName()+" erro-> "+ e.getMessage());
244         }
245     } else {
246         System.err.println("No istaragent specified");
247     }
248 }

```

Figura 33 - Trecho de código - método setup() de IstarJadeAgent

A seguir, apresentamos o rastro do procedimento de criação dos agentes e inicialização do exemplo do sistema de controle automático da porta, cujo modelo encontra-se na Figura I6. As linhas com os elementos relativos ao requisito de consciência estão destacadas em negrito.

Actor DoorSystem type: class istar.impl.Role
DoorSystem - Internal ielement:1.1 name:CloseTheDoor type:class basicelement.CloseTheDoor
DoorSystem - Internal ielement:1.2 name:OpenTheDoor type:class basicelement.OpenTheDoor
DoorSystem - Internal ielement:1.3 name:DoorOperatedAutomatically type:class istar.impl.Goal

DoorOperatedAutomatically - Means-End link sub-elements:

mean: type:class basicelement.CloseTheDoor name:CloseTheDoor
 mean: type:class basicelement.OpenTheDoor name:OpenTheDoor

DoorSystem - Internal ielement:1.4 name:PresenceSensor type:class basicelement.PresenceSensor

DoorSystem - Internal ielement:1.5 name:presence_data_aquisition type:class istar.impl.Task

presence_data_aquisition TaskDecomposition link subelements:

sub-element: type:class basicelement.PresenceSensor id:1.4 name:PresenceSensor

DoorSystem - Internal ielement:1.6 name:PresenceDataInterpretation type:class basicelement.PresenceDataInterpretation

DoorSystem - Internal ielement:1.7 name:PresenceContext type:class istar.impl.ContextAwareness

DoorSystem - Internal ielement:107 name:Awareness[PresenceOfPeople] type:class istar.impl.Softgoal

Awareness[PresenceOfPeople] - Means-End link sub-elements:

mean: type:class istar.impl.ContextAwareness name:PresenceContext

DoorSystem - Internal ielement:1.8 name:FireDataAquisition type:class basicelement.FireDataAquisition

DoorSystem - Internal ielement:1.9 name:FireDataInterpretation type:class basicelement.FireDataInterpretation

DoorSystem - Internal ielement:1.10 name:FireContext type:class istar.impl.ContextAwareness

DoorSystem - Internal ielement:110 name:Awareness[FireAlert] type:class istar.impl.Softgoal

Awareness[FireAlert] - Means-End link sub-elements:

mean: type:class istar.impl.ContextAwareness name:FireContext

DoorSystem - Internal ielement:1.11 name:manage_the_door type:class istar.impl.Task

manage_the_door TaskDecomposition link subelements:

sub-element: type:class istar.impl.Softgoal id:107 name:Awareness[PresenceOfPeople]

sub-element: type:class istar.impl.Softgoal id:110 name:Awareness[FireAlert]

sub-element: type:class istar.impl.Goal id:1.3 name:DoorOperatedAutomatically

DoorSystem1.12 name:DoorManagedAutomatically type:class mainelement.DoorManagedAutomatically

DoorManagedAutomatically - Means-End link sub-elements:

mean: type:class istar.impl.Task name:manage_the_door

Actor door-agent type: class istar.impl.IstarAgent
 Actor door-agent plays Role: DoorSystem

Actor Fire prevention system type: class istar.impl.Role
 Fire prevention system - Internal ielement:2.1 name:SmokeSensor type:class basicelement.SmokeSensor

Fire prevention system - Internal ielement:2.2 name:detect_smoke type:class istar.impl.Task

```

detect_smoke TaskDecomposition link subelements:
  sub-element: type:class basicelement.SmokeSensor id:2.1 name:SmokeSensor
  Fire prevention system - Internal ielement:2.3 name:BroadcastAlarm type:class
  basicelement.BroadcastAlarm
  Fire prevention system - Internal ielement:2.4 name:EliminateFire type:class
  basicelement.EliminateFire
  Fire prevention system - Internal ielement:2.5 name:reduce_damages type:class
  istar.impl.Task
reduce_damages TaskDecomposition link subelements:
  sub-element: type:class basicelement.BroadcastAlarm id:2.3
  name:BroadcastAlarm
  sub-element: type:class basicelement.EliminateFire id:2.4 name:EliminateFire
  Fire prevention system - Internal ielement:2.6 name:prevent_fire type:class
  istar.impl.Task
prevent_fire TaskDecomposition link subelements:
  sub-element: type:class istar.impl.Task id:2.2 name:detect_smoke
  sub-element: type:class istar.impl.Task id:2.5 name:reduce_damages
  Fire prevention system2.7 name:PreventFireAutomatically type:class
  mainelement.PreventFireAutomatically
  PreventFireAutomatically - Means-End link sub-elements:
    mean: type:class istar.impl.Task name:prevent_fire
    Fire prevention system Dependee element: id:3 - dependum: type:class
    istar.impl.Resource name:FireAlert

Actor fire-agent type: class istar.impl.IstarAgent
Actor fire-agent plays Role: Fire prevention system

```

```

//Registro dos agentes no DF
door-agent is registering as listener to the service: istar.impl.Resource:FireAlert
fire-agent is registering as provider to the service: istar.impl.Resource:FireAlert

```

O primeiro ciclo de execução de comportamento do *door-agent*, com as linhas dos comportamentos relativos ao requisito de consciência destacadas em negrito.

```

20140131 18:44:52: door-agent@10.26.144.42:1099/JADE: executing.
Ticket=door-agent@10.26.144.42:1099/JADE_1
20140131 18:44:53: Active has: manage_the_door;
20140131 18:44:53: behaviour name=Active adding subBehaviour
manage_the_door
20140131 18:44:53: manage_the_door has:
Awareness[PresenceOfPeople];
20140131 18:44:53: behaviour name=manage_the_door adding
subBehaviour Awareness[PresenceOfPeople]

```

```
20140131 18:44:53: manage_the_door has:  
Awareness[PresenceOfPeople];Awareness[FireAlert];  
20140131 18:44:53: behaviour name=manage_the_door adding  
subBehaviour Awareness[FireAlert]  
20140131 18:44:53: manage_the_door has:  
Awareness[PresenceOfPeople];Awareness[FireAlert];DoorOperatedAuto  
matically;  
20140131 18:44:53: behaviour name=manage_the_door adding  
subBehaviour DoorOperatedAutomatically  
20140131 18:44:53: Awareness[PresenceOfPeople] has:  
PresenceContext;  
20140131 18:44:53: behaviour name=Awareness[PresenceOfPeople]  
adding subBehaviour PresenceContext  
20140131 18:44:53: PresenceContext has: presence_data_aquisition;  
20140131 18:44:53: behaviour name=PresenceContext adding  
subBehaviour presence_data_aquisition  
20140131 18:44:53: PresenceContext has:  
presence_data_aquisition;PresenceDataInterpretation;  
20140131 18:44:53: behaviour name=PresenceContext adding  
subBehaviour PresenceDataInterpretation  
20140131 18:44:53: presence_data_aquisition has: PresenceSensor;  
20140131 18:44:53: behaviour name=presence_data_aquisition adding  
subBehaviour PresenceSensor  
20140131 18:44:53: Starting PresenceDataInterpretation  
20140131 18:44:53: Awareness[FireAlert] has: FireContext;  
20140131 18:44:53: behaviour name=Awareness[FireAlert] adding  
subBehaviour FireContext  
20140131 18:44:53: FireContext has: FireDataAquisition;  
20140131 18:44:53: behaviour name=FireContext adding subBehaviour  
FireDataAquisition  
20140131 18:44:53: FireContext has:  
FireDataAquisition;FireDataInterpretation;  
20140131 18:44:53: behaviour name=FireContext adding subBehaviour  
FireDataInterpretation  
20140131 18:44:53: Starting FireDataInterpretation
```

```
20140131 18:44:53: DoorOperatedAutomatically has:  
CloseTheDoor;OpenTheDoor;  
20140131 18:44:53: behaviour name=DoorOperatedAutomatically  
adding subBehaviour CloseTheDoor
```

O rastro do primeiro ciclo de execução de comportamento do agente *fire-agent* é apresentado a seguir.

```
20140131 18:44:52: fire-agent@10.26.144.42:1099/JADE: executing.  
Ticket=fire-agent@10.26.144.42:1099/JADE_1  
20140131 18:44:53: Active has: prevent_fire;  
20140131 18:44:53: behaviour name=Active adding subBehaviour  
prevent_fire  
20140131 18:44:53: prevent_fire has: detect_smoke;  
20140131 18:44:53: behaviour name=prevent_fire adding  
subBehaviour detect_smoke  
20140131 18:44:53: prevent_fire has: detect_smoke;reduce_damages;  
20140131 18:44:53: behaviour name=prevent_fire adding  
subBehaviour reduce_damages  
20140131 18:44:53: detect_smoke has: SmokeSensor;  
20140131 18:44:53: behaviour name=detect_smoke adding  
subBehaviour SmokeSensor  
20140131 18:44:53: Simulation time:23/01/2014 10:01:00 >>  
Fire=NO-FIRE  
20140131 18:44:53: reduce_damages has: BroadcastAlarm;  
20140131 18:44:53: behaviour name=reduce_damages adding  
subBehaviour BroadcastAlarm  
20140131 18:44:53: reduce_damages has:  
BroadcastAlarm;EliminateFire;  
20140131 18:44:53: behaviour name=reduce_damages adding  
subBehaviour EliminateFire  
20140131 18:44:53: RESET_BEHAVIOUR state=RUNNING
```

4.4. Consciência e avaliação de alternativas;

O requisito de consciência é efetivamente implementado pelas classes correspondentes às estruturas canônicas destacadas no modelo i*, da Figura 16

(página 90), e seus respectivos comportamentos. A estrutura é composta por um softgoal representando o requisito de consciência que por sua vez é decomposto por um elemento de contexto. Este elemento é decomposto por duas tarefas, uma para aquisição de dados de contexto e outra para interpretação dos dados, e possui elos de argumentação para as alternativas à meta ao qual está relacionado. A implementação das tarefas, tanto de aquisição de dados, quanto de interpretação de dados, é feita através de elementos básicos no framework istarjade com base na especificação do requisito de consciência. As seções de descrição de contexto e domínio das variáveis, especificação das situações de contexto e escolha das alternativas de ação, constantes da especificação do requisito de consciência, têm justamente a finalidade de guiar a implementação desses elementos no framework istarjade.

A elaboração da especificação do requisito de consciência, por sua vez, beneficia-se do uso do catálogo de consciência de software, principalmente pela sugestões de operacionalização para aquisição do contexto. A tarefa de interpretação tende a ser específica de cada problema, uma vez que relaciona o contexto subjacente às metas específicas do software que está sendo desenvolvido.

No exemplo do sistema para controle da porta automática, apresentado na Figura 24 (página 115), a meta flexível de consciência da presença de pessoas (*Awareness[PresenseOfPeople]*) é adicionada ao comportamento do agente. Este comportamento, por sua vez, adiciona recursivamente o elemento de contexto presença de pessoas que representa o meio pelo qual a o requisito de consciência será satisfeita a contento. O contexto de presença de pessoas (*PresenceOfPeopleContext*) por sua vez, tem a subtarefa de aquisição de dados de presença (*PresenceDataAquisition*) que é refinada por decomposição para o recurso básico: sensor de presença (*PresenceSensor*). Por ser um recurso básico, foi desenvolvida uma classe (como especialização da classe *BasicResouce*) que implementa o comportamento desejado. No caso da simulação, a classe *PresenceSensor* lia os dados de presença de um arquivo simulando a presença ou ausência de pessoas. Com a informação sobre a presença ou ausência de pessoas adquirida o comportamento da classe *PresenceSensor* é finalizado e o comportamento da próxima subtarefa na decomposição do contexto é carregado. Como a subtarefa de interpretação dos dados de presença também é uma classe básica, foi desenvolvida a classe *PresenceDataInterpretation* (como especialização da classe *BasicTask*) que implementa o comportamento de interpretação dos dados de presença de

pessoas. Este comportamento avalia os meios abrir a porta (*OpenTheDoor*) e fechar a porta (*CloseTheDoor*), alterando as contribuições destes meios para a satisfação da meta operar a porta automaticamente (*DoorOperatedAutomatically*) de acordo com a situação identificada com base na informação de presença adquirida na subtarefa anterior. Após a finalização do comportamento, com os elos de contribuição dos meios já avaliados e devidamente alterados pela tarefa de interpretação, o comportamento correspondente ao elo meios-fim avalia que meio deve ser escolhido com base na contribuição de cada meio (*CloseTheDoor* e *OpenTheDoor*) para a satisfação da meta, escolhendo o mais adequado de acordo com a situação percebida.

Um procedimento análogo é executado em relação ao requisito de consciência de alerta de fogo (*Awareness[FireAlert]*) e ao contexto de (existência de) fogo (*FireContext*). No caso do contexto de fogo, a aquisição de dados consiste em verificar o recebimento de uma mensagem de alerta do agente *fire-agent* que desempenha o papel do sistema de prevenção de fogo (*FireSystem*). O comportamento para interpretação dos dados de fogo, implementado na classe *FireDataInterpretation* (que estende a classe *BasicTask*), altera a contribuição dos meios (*CloseTheDoor* e *OpenTheDoor*) apenas se uma mensagem de alerta (da provável presença de) fogo foi recebida, o que indica a ocorrência da situação de fogo. Caso contrário, o valor de contribuição dos meios permanece inalterado e a escolha do meio será determinada pelo contexto de presença de pessoas.

4.5. Considerações sobre o framework *istarjade*

iStarJade é um framework experimental construído com o objetivo de auxiliar na validação de casos para pesquisa. A adição dinâmica de comportamentos permite a rápida elaboração de protótipos para validação de sistemas multiagente modelados em i* e descritos em *iStarML*. Em função desta mesma característica de adição dinâmica de comportamentos, *iStarJade* impõe uma forma particular de implementação: os comportamentos específicos de cada caso devem ser modelados em classes de elementos principais, tipicamente metas, onde são estabelecidos os procedimento para deliberação (ativação) de uma meta e para verificação de sua satisfação. Os demais comportamentos dos agentes são implementados em classes de elementos básicos (nós folha no racional de cada agente). Tanto as classes com

comportamentos dos elementos principais, quanto as classes com comportamento dos elementos básicos, são carregadas no *framework* através de *JAVA Reflection*.

A ordem que esses comportamentos vão sendo adicionados ao comportamento da máquina de estados finita de uma agente *IStarJadeAgent* é definida pela ordem da decomposição das tarefas e pelos graus de contribuição dos meios para os fins no elos meios-fim. Esses aspectos precisam ser levados em consideração na elaboração dos modelos i* que serão executados no *framework*.

O *framework* possui um pacote básico chamado *istar* com as interfaces para os elementos e atores de i*, utilizadas para mapear os elementos descritos no arquivo iStarML no procedimento de carga dos agentes. Um pacote com os comportamentos implementados no *framework* chamado *istar.behaviour* com as classes para os comportamentos dos elos de decomposição (sequencial e paralelo) e para os comportamentos dos elos meios-fim (único e paralelo). Um pacote chamado *istar.impl* com as classes que implementam as interfaces dos elementos e atores i* definidas no pacote *istar* e um pacote chamado *istar.agent* com a classe para implementação dos agentes *IStarJadeAgent*.

As Figuras 34, 35 e 36 apresentam exemplos de classes e interfaces Java implementadas no *framework* istarjade, disponível no ambiente de hospedagem de projetos do google code⁷ através do endereço <https://istarjade.googlecode.com/svn/trunk/>

⁷ O Project Hosting on Google Code oferece um ambiente gratuito de desenvolvimento colaborativo para projetos de código aberto

The screenshot shows the 'Source' tab of the iStarJade project on Google Project Hosting. The page displays a list of Java files under the 'svn/trunk/istar' directory. The files are listed in a table with columns for 'Filename', 'Size', 'Rev', 'Date', and 'Author'. All files were last modified on Jan 30 (3 days ago) by user 'herbet'. The 'Filename' column contains links to the source code for each file.

Filename	Size	Rev	Date	Author
IActor.java	892 bytes	r39	Jan 30 (3 days ago)	herbet
IActorLink.java	315 bytes	r39	Jan 30 (3 days ago)	herbet
IAgent.java	477 bytes	r39	Jan 30 (3 days ago)	herbet
IBasicElement.java	277 bytes	r39	Jan 30 (3 days ago)	herbet
IDependency.java	754 bytes	r39	Jan 30 (3 days ago)	herbet
IElement.java	1.8 KB	r39	Jan 30 (3 days ago)	herbet
IElementLink.java	660 bytes	r39	Jan 30 (3 days ago)	herbet
IGoal.java	611 bytes	r39	Jan 30 (3 days ago)	herbet
IPosition.java	314 bytes	r39	Jan 30 (3 days ago)	herbet
IRefinedActor.java	165 bytes	r39	Jan 30 (3 days ago)	herbet
IResource.java	341 bytes	r39	Jan 30 (3 days ago)	herbet
IRole.java	188 bytes	r39	Jan 30 (3 days ago)	herbet
ISoftgoal.java	380 bytes	r39	Jan 30 (3 days ago)	herbet
ITask.java	1.2 KB	r39	Jan 30 (3 days ago)	herbet
ISPartOfLink.java	780 bytes	r39	Jan 30 (3 days ago)	herbet

[Create or upload a new file](#)

Your project is using approximately 1.2 MB out of 4096 MB total quota.
You can [reset this repository](#) so that svn:sync can be used to upload existing code history.

[Terms](#) - [Privacy](#) - [Project Hosting Help](#)
Powered by [Google Project Hosting](#)

Figura 34 - Interfaces definidas no framework istarjade

The screenshot shows the 'Source' tab of the iStarJade project on Google Project Hosting. The page displays a list of Java files under the 'svn/trunk/istar' directory. The files are listed in a table with columns for 'Filename', 'Size', 'Rev', 'Date', and 'Author'. All files were last modified on Jan 30 (3 days ago) by user 'herbet'. The 'Filename' column contains links to the source code for each file.

Filename	Size	Rev	Date	Author
AbstractIstarBehaviour.java	2.9 kB	r39	Jan 30 (3 days ago)	herbet
IstarBehaviour.java	369 bytes	r39	Jan 30 (3 days ago)	herbet
IstarMainBehaviour.java	6.2 kB	r39	Jan 30 (3 days ago)	herbet
IstarParallelBehaviour.java	3.4 kB	r39	Jan 30 (3 days ago)	herbet
MeansEndUniqueBehaviour.java	4.9 kB	r39	Jan 30 (3 days ago)	herbet
RequestExternalBehaviour.java	33.4 kB	r39	Jan 30 (3 days ago)	herbet
RequestExternalElement.java	852 bytes	r39	Jan 30 (3 days ago)	herbet
SequentialTaskBehaviour.java	4.2 kB	r39	Jan 30 (3 days ago)	herbet
TopicMatchExpression.java	676 bytes	r39	Jan 30 (3 days ago)	herbet

[Create or upload a new file](#)

Your project is using approximately 1.2 MB out of 4096 MB total quota.
You can [reset this repository](#) so that svn:sync can be used to upload existing code history.

[Terms](#) - [Privacy](#) - [Project Hosting Help](#)
Powered by [Google Project Hosting](#)

Figura 35 – Classes de comportamentos do framework istarjade

Source path: svn/
Directories

Filename	Size	Rev	Date	Author
AbstractBasicBelief.java	570 bytes	r39	Jan 30 (3 days ago)	herbet
AbstractBasicContextAwareness.java	646 bytes	r39	Jan 30 (3 days ago)	herbet
AbstractBasicGoal.java	427 bytes	r39	Jan 30 (3 days ago)	herbet
AbstractBasicResource.java	443 bytes	r39	Jan 30 (3 days ago)	herbet
AbstractBasicSoftgoal.java	442 bytes	r39	Jan 30 (3 days ago)	herbet
AbstractBasicTask.java	807 bytes	r39	Jan 30 (3 days ago)	herbet
AbstractElement.java	10.4 KB	r39	Jan 30 (3 days ago)	herbet
AbstractElementLink.java	1.8 KB	r39	Jan 30 (3 days ago)	herbet
AbstractMainGoal.java	1.0 KB	r39	Jan 30 (3 days ago)	herbet
AbstractMainResource.java	1.0 KB	r39	Jan 30 (3 days ago)	herbet
AbstractMainSoftgoal.java	1.0 KB	r39	Jan 30 (3 days ago)	herbet
AbstractMainTask.java	1.0 KB	r39	Jan 30 (3 days ago)	herbet
Actor.java	3.9 KB	r39	Jan 30 (3 days ago)	herbet
AndDecomposition.java	655 bytes	r39	Jan 30 (3 days ago)	herbet
Belief.java	2.6 KB	r39	Jan 30 (3 days ago)	herbet
ContextAwareness.java	3.3 KB	r39	Jan 30 (3 days ago)	herbet
ContributionLink.java	1.6 KB	r39	Jan 30 (3 days ago)	herbet

Your project is using approximately 1.2 MB out of 4096 MB total quota.
You can [reset this repository](#) so that svn:sync can be used to upload existing code history.

Terms · Privacy · Project Hosting Help
Powered by [Google Project Hosting](#)

Figura 36 – Algumas classes do framework iStarJade

Resumo do capítulo

Neste capítulo apresentamos a plataforma JADE para desenvolvimento e execução de sistemas multiagente que utilizamos como base para elaboração do framework *iStarJade*, que desenvolvemos para implementação dos sistemas multiagente modelados em *i** e descritos em *iStarML*. Apresentamos o mapeamento entre os elementos de *i** e as classes de *iStarJade*, bem como a dinâmica de funcionamento e a implementação do requisito de consciência neste framework, que utilizamos como base para os casos de validação da tese a serem apresentados no próximo capítulo.

5 Validação Experimental

Neste capítulo são apresentados alguns casos para validação de nossa proposta de abordagem do requisito de consciência. Para os casos, apresentaremos uma breve seção descritiva do problema, os modelos e especificações produzidos, os dados usados para simulação, e o resultado da simulação dos casos implementados no framework iStarjade introduzido no capítulo anterior.

Iniciaremos apresentando o caso do sistema de controle automático de porta que foi usado como exemplo nos capítulos anteriores (modelagem e implementação), restando apresentar os dados da simulação e resultados da execução da simulação em iStarjade. No segundo caso apresentamos um sistema de elevadores para atendimento das demandas de deslocamento dos usuários, com três cenários: apenas um elevador; com dois elevadores; e com três elevadores. Por último, apresentamos um caso com um sistema multiagente para investimento em ações na bolsa de valores.

5.1. **Caso 1 – Sistema de controle automático de porta**

5.1.1. **Descrição do problema:**

No caso do ambiente inteligente com um sistema de controle automático de porta, o agente *door-agent* controla automaticamente a abertura e fechamento da porta da sala, que deve ser aberta sempre que houver a presença de pessoas na proximidade da porta ou que o agente receber um alerta de provável presença de fogo do agente *fire-agent*. Este caso é o mesmo apresentado anteriormente no Capítulo 2 - Modelagem do requisito de consciência de software. A descrição do problema encontra-se na seção 3.6 *Considerações sobre a modelagem do requisito de consciência*, juntamente com o modelo i*, as especificações do requisito de consciência, a descrição do modelo em iStarML e o rastro de criação dos agentes em iStarjade.

A seguir, apresentamos novamente a especificação do requisito de consciência para esse caso.

Awareness[Presence of People]		
Topic description:	The agent should open the door whenever there are people near the door. In order to do that, the agent needs to be aware of the presence of people near the door and act according to his perception.	
Goal:	Door managed automatically	
Awareness subtype:	Physical environment	
Suggested operationalizations:	Use of presence sensors	
Alternative actions:	Open the door; Close the door	
Entity:	People using the room.	
Source of entity data:	People presence sensor	
ContextDescription	{peoplePresence}	
Domains of variable		
Variable name	Domain	
peoplePresence	{"PRESENCE", "ABSENCE"}	
Context situations especification		
Situation name	Specification	
PresenceOfPeople	peoplePresence=="PRESENCE"	
AbsenceOfPeople	peoplePresence=="ABSENCE"	
Alternative action choice		
Situation	Alternative action	Impact
Presence of people	OpenTheDoor	MAKE (strongly positive)
Presence of people	CloseTheDoor	BREAK (strongly negative)
Absence of people	CloseTheDoor	MAKE (strongly positive)
Absence of People	OpenTheDoor	BREAK (strongly negative)

5.1.2.

Dados da simulação:

A Tabela 17, apresentada a seguir, contém os dados usados na simulação deste caso.

Simulation Step	Time	PresenceOfPeople	PresenceOfFire
1	23/01/2014 10:00:00	ABSENCE	NO-FIRE
2	23/01/2014 10:02:00	ABSENCE	NO-FIRE
3	23/01/2014 10:05:00	PRESENCE	NO-FIRE
4	23/01/2014 10:06:00	PRESENCE	NO-FIRE
5	23/01/2014 10:08:00	ABSENCE	NO-FIRE
6	23/01/2014 10:10:00	ABSENCE	FIRE
7	23/01/2014 10:11:00	ABSENCE	NO-FIRE

Tabela 17 – Dados da simulação do caso do SMA para controle de porta

O resultado da execução da simulação pelos agentes é apresentado a seguir. Vale notar que no passo 6 da simulação, após receber um alerta de fogo, *door-agent* avalia novamente as ações (as linhas “*Mean:CloseTheDoor – eval:*” e “*Mean:OpenTheDoor – eval:*”) com base na informação de fogo (“*FIRE:FIRE*”) alterando os valores de contribuição das ações, o que acarreta a mudança da ação a ser executada, descrita na última linha do passo da simulação - inicialmente *CloseTheDoor* possuía valor de contribuição maior e seria a ação escolhida, porém a receber a informação de fogo, as ações são reavaliadas e a ação *OpenTheDoor* recebe um valor de contribuição maior e é escolhida.

Passo 1 da simulação: 23/01/2014 10:01:00

door-agent Simulation time:23/01/2014 10:00:00 >> Presence:ABSENCE

door-agent >>> Starting PresenceDataInterpretation

Presence of people:ABSENCE

=====

Mean:CloseTheDoor - eval:2

Mean:OpenTheDoor - eval:-2

=====

fire-agent Simulation time:23/01/2014 10:00:00 >> Fire:NO-FIRE

door-agentChecking if there a fire alert

door-agent >>> Starting FireDataInterpretation

Presence of fire: NO-FIRE

SimulationStep=1 - SimulationTime=23/01/2014 10:00:00 - door-agent is closing the door

Passo 2 da simulação: 23/01/2014 10:02:00

fire-agent Simulation time:23/01/2014 10:02:00 >> Fire:NO-FIRE

door-agent Simulation time:23/01/2014 10:02:00 >> Presence:ABSENCE

door-agent >>> Starting PresenceDataInterpretation

Presence of people:ABSENCE

=====

Mean:CloseTheDoor - eval:2

Mean:OpenTheDoor - eval:-2

=====

door-agentChecking if there a fire alert

door-agent >>> Starting FireDataInterpretation

Presence of fire: NO-FIRE

=====

SimulationStep=2 - SimulationTime=23/01/2014 10:02:00 - door-agent is closing the door

Passo 3 da simulação: 23/01/2014 10:05:00

fire-agent Simulation time:23/01/2014 10:05:00 >> Fire:NO-FIRE

door-agent Simulation time:23/01/2014 10:05:00 >> Presence:PRESENCE

door-agent >>> Starting PresenceDataInterpretation

Presence of people:PRESENCE

=====

Mean:CloseTheDoor - eval:-2

Mean:OpenTheDoor - eval:2

=====

door-agentChecking if there a fire alert

door-agent >>> Starting FireDataInterpretation

Presence of fire: NO-FIRE

=====

SimulationStep=3 - SimulationTime=23/01/2014 10:05:00 - door-agent is opening the door

Passo 4 da simulação: 23/01/2014 10:06:00

fire-agent Simulation time:23/01/2014 10:06:00 >> Fire:NO-FIRE
door-agent Simulation time:23/01/2014 10:06:00 >> Presence:PRESENCE
door-agent >>> Starting PresenceDataInterpretation
Presence of people:PRESENCE
=====
Mean:CloseTheDoor - eval:-2
Mean:OpenTheDoor - eval:2
=====
door-agentChecking if there a fire alert
door-agent >>> Starting FireDataInterpretation
Presence of fire: NO-FIRE
=====
SimulationStep=4 - SimulationTime=23/01/2014 10:06:00 - door-agent is opening
the door

Passo 5 da simulação: 23/01/2014 10:08:00

fire-agent Simulation time:23/01/2014 10:08:00 >> Fire:NO-FIRE
door-agent Simulation time:23/01/2014 10:08:00 >> Presence:ABSENCE
door-agent >>> Starting PresenceDataInterpretation
Presence of people:ABSENCE
=====
Mean:CloseTheDoor - eval:2
Mean:OpenTheDoor - eval:-2
=====
door-agentChecking if there a fire alert
door-agent >>> Starting FireDataInterpretation
Presence of fire: NO-FIRE
=====
SimulationStep=8 - SimulationTime=23/01/2014 10:08:00 - door-agent is closing
the door

Passo 6 da simulação: 23/01/2014 10:10:00

fire-agent Simulation time:23/01/2014 10:10:00 >> Fire:FIRE
fire-agent is broadcasting a fire alarm (topic: istar.impl.Resource:FireAlert)
fire-agent is taking actions to eliminate fire

door-agent Simulation time:23/01/2014 10:10:00 >> Presence:ABSENCE
door-agent >>> Starting PresenceDataInterpretation
Presence of people:ABSENCE
=====

Mean:CloseTheDoor - eval:2
Mean:OpenTheDoor - eval:-2
=====

door-agentChecking if there a fire alert
door-agent@10.26.144.42:1099/JADE received a fire alert message from agent fireagent@10.26.144.42:1099/JADE with content: FIRE
door-agent >>> Starting FireDataInterpretation
Presence of fire:FIRE
=====

Mean:CloseTheDoor - eval:-2
Mean:OpenTheDoor - eval:2
=====

SimulationStep=10 - SimulationTime=23/01/2014 10:10:00 - door-agent is opening the door

Passo 7 da simulação: 23/01/2014 10:11:00

fire-agent Simulation time:23/01/2014 10:11:00 >> Fire:NO-FIRE
door-agent Simulation time:23/01/2014 10:11:00 >> Presence:ABSENCE
door-agent >>> Starting PresenceDataInterpretation
Presence of people:ABSENCE
=====

Mean:CloseTheDoor - eval:2
Mean:OpenTheDoor - eval:-2
=====

door-agentChecking if there a fire alert
door-agent >>> Starting FireDataInterpretation
Presence of fire: NO-FIRE
=====

SimulationStep=11 - SimulationTime=23/01/2014 10:11:00 - door-agent is closing the door

Este caso é interessante por haver duas metas de flexíveis, representando requisitos de consciência. As duas metas foram operacionalizadas e executadas em cada passo da simulação. O resultado alcançado com a simulação do sistema multiagente implementando o requisito de consciência especificado foi satisfatório.

5.2.

Caso 2 – Sistema de controle de elevadores

5.2.1.

Descrição do problema:

Considere um prédio com um sistema automático para controle dos elevadores. Ao solicitar o uso de um elevador o usuário informa o andar de destino (o andar atual em que o usuário se encontra pode ser facilmente detectado pelo sistema). O sistema então decide qual elevador atenderá a solicitação do usuário.

O modelo i* para o sistema multiagente de controle de elevadores possui dois papéis: gerenciador de demandas (*DemandsManager*) e elevador (*Elevator*). O agente *manager* que desempenha o papel de gerenciador de demandas possui um elemento de contexto *UsersAndElevatorsContext* que obtém a solicitação do usuário, a posição do(s) elevador(es) e decide para qual elevador deve enviar a solicitação de transporte do usuário. O agente que desempenha o papel de elevador (*Elevator*) deve informar o seu agendamento contendo seu andar atual e o andar de destino para o qual está se dirigindo. Caso seja selecionado para atender a solicitação de transporte do usuário, esse agente receberá uma solicitação de transporte do agente *manager* informando o andar atual do usuário e o seu andar de destino.

Para esse caso, desenvolvemos três cenários: com um elevador, com dois elevadores e com três elevadores. A seguir apresentamos o cenário com um elevador.

A descrição em iStarML dos modelos i* de cada cenário, juntamente com o rastro de criação dos agentes, encontra-se no Anexo I.

5.2.2.

Cenário com um elevador (agente elevator1)

O modelo i* para o cenário com um elevador é apresentado na Figura 37, a seguido da especificação do requisito de consciência.

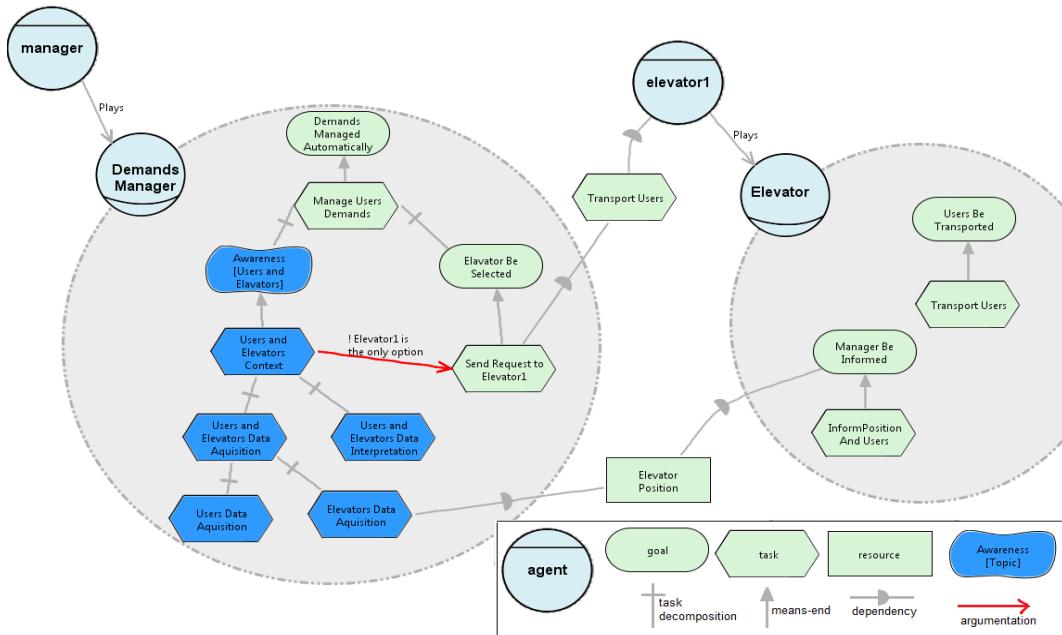


Figura 37 - modelo i* para o cenário com um elevador

A especificação do requisito de consciência é apresentada a seguir.

Awareness[User request and Elevators]	
Topic description:	The agent should be aware of user request (for transport between different floors) and elevators current position and schedules in order to decide which elevator will serve the user request.
Goal:	Users demands be managed automatically
Awareness subtype:	User awareness; Position (elevators)
Suggested operationalizations:	Use system user interface that allow users to inform their requests, including destination; Communicate with automation elevator system
Alternative actions:	Send the request to elevator1
Entity:	Users and elevators
Source of entity data:	User request devices and elevators
ContextDescription	{UserRequest, ElevatorSchedule}
Domains of variable	
Variable name	Domain
UserRequest	{Trip}
ElevatorSchedule	{Trip}
Trip: is a data entity with attributes: number of users (integer); source floor (integer between 1 and 15); destination floor (integer between 1 and 15).	
Total distance (in number of floors to be covered): the distance that the elevator will go to serve the route for which has been set at a given time and the client's request.	
Context situations especification	
Situation name	Specification
Elevator1 is the only option	Whatever be the underlying data acquired, this situation happens
Alternative action choice	

Situation	Alternative action	Impact
Elevator1 is the only option	SendToElevator1	MAKE (strongly positive)

Os dados da simulação de simulação do cenário com um elevador encontram-se na Tabela 18, abaixo.

Simulation Step	Time	UserRequest (from-to)	Elevator1Schedule (from-to)
1	23/01/2014 10:00:00	1-10	1-1
2	23/01/2014 10:02:00	12-5	2-5
3	23/01/2014 10:05:00	15-1	8-2
4	23/01/2014 10:08:00	7-9	10-10
5	23/01/2014 10:10:00	4-13	3-7
6	23/01/2014 10:12:00	11-3	4-4
7	23/01/2014 10:15:00	6-1	1-12

Tabela 18 - Dados para simulação do cenário com um elevador

Os passos 1, 4 e 6 simulam o elevador parado nos andares 1, 10 e 4 respectivamente.

A tarefa de interpretação e avaliação das alternativas é baseada no cálculo da distância total (em número de andares a serem percorridos) que o elevador irá percorrer para atender ao percurso para o qual já está programado em um dado instante e a solicitação do cliente. O valor de contribuição será obtido subtraindo-se de 41⁸ a distância a ser percorrida pelo elevador.

Como nesse primeiro cenário existe apenas um elevador, independente da avaliação, a alternativa escolhida é sempre a mesma: enviar a solicitação para o elevador1.

O rastro da execução da simulação é apresentado a seguir.

Passo 1 da simulação: 23/01/2014 10:00:00

```

Elevator1 Simulation time:23/01/2014 10:00:00 :
Trip [time=23/01/2014 10:00:00, from=1, to=[1], numberOfPassengers=0]
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:00:00, from=1,
to=[1],numberOfPassengers=0]
manager Simulation time:23/01/2014 10:00:00 :
Trip [time=23/01/2014 10:00:00, from=1, to=[10], numberOfPassengers=1]
managerChecking if there is an elevator schedule: 23/01/2014
10:00:00istar.impl.Resource:ElevatorPosition
manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip
[time=23/01/2014 10:00:00, from=1, to=[1], numberOfPassengers=0]}

```

⁸ Usamos a constante 41 por ser o valor máximo de percurso (caso de o elevador estar no andar 14 agendado para ir para o andar 1 e um usuário solicitar ir do andar 15 para o andar 1).

```
manager >>> Starting UsersAndElevatorsDataInterpretation
UsersTrip:Trip [time=23/01/2014 10:00:00, from=1, to=[10],
numberOfPassengers=1]
=====
Mean:SendToElevator1 - eval: 32
=====
SimulationStep=1 - SimulationTime=23/01/2014 10:00:00 - manager is sending
to elevator 1
```

Passo 2 da simulação: 23/01/2014 10:02:00

```
Elevator1 Simulation time:23/01/2014 10:02:00 :
Trip [time=23/01/2014 10:02:00, from=2, to=[5], numberOfPassengers=1]
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:02:00, from=2,
to=[5],numberOfPassengers=1]
=====
manager Simulation time:23/01/2014 10:02:00 :
Trip [time=23/01/2014 10:02:00, from=12, to=[5], numberOfPassengers=1]
managerChecking if there is an elevator schedule: 23/01/2014 10:02:00
istar.impl.Resource:ElevatorPosition
manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip
[time=23/01/2014 10:02:00, from=2, to=[5], numberOfPassengers=1]}
manager >>> Starting UsersAndElevatorsDataInterpretation
UsersTrip:Trip [time=23/01/2014 10:02:00, from=12, to=[5],
numberOfPassengers=1]
=====
Mean: SendToElevator1 - eval: 24
=====
SimulationStep=2 - SimulationTime=23/01/2014 10:02:00 - manager is sending
to elevator 1
```

Passo 3 da simulação: 23/01/2014 10:05:00

```
Elevator1 Simulation time:23/01/2014 10:05:00 :
Trip [time=23/01/2014 10:05:00, from=8, to=[2], numberOfPassengers=3]
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:05:00, from=8,
to=[2], numberOfPassengers=3]
manager Simulation time:23/01/2014 10:05:00 :
Trip [time=23/01/2014 10:05:00, from=15, to=[1], numberOfPassengers=1]
managerChecking if there is an elevator schedule: 23/01/2014 10:05:00-
istar.impl.Resource:ElevatorPosition
manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip
[time=23/01/2014 10:05:00, from=8, to=[2], numberOfPassengers=3]}
manager >>> Starting UsersAndElevatorsDataInterpretation
UsersTrip:Trip [time=23/01/2014 10:05:00, from=15, to=[1],
numberOfPassengers=1]
=====
Mean: SendToElevator1 - eval: 27
=====
SimulationStep=3 - SimulationTime=23/01/2014 10:05:00 - manager is sending
to elevator 1
```

Passo 4 da simulação: 23/01/2014 10:08:00

Elevator1 Simulation time:23/01/2014 10:08:00 :
 Trip [time=23/01/2014 10:08:00, from=10, to=[10], numberOfPassengers=0]
 Elevator1 is sending his schedule: Trip [time=23/01/2014 10:08:00, from=10, to=[10],numberOfPassengers=0]
 manager Simulation time:23/01/2014 10:08:00 :
 Trip [time=23/01/2014 10:08:00, from=7, to=[9], numberOfPassengers=1]
 Elevator1@10.26.144.42:1099/JADE - Initial is executing step=0
 Elevator1 executou Initial onEnd() com result=0
 managerChecking if there is an elevator schedule: 23/01/2014 10:08:00
 istar.impl.Resource:ElevatorPosition
 manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip
 [time=23/01/2014 10:08:00, from=10, to=[10], numberOfPassengers=0]}
 manager >>> Starting UsersAndElevatorsDataInterpretation
 UsersTrip:Trip [time=23/01/2014 10:08:00, from=7, to=[9],
 numberOfPassengers=1]
 ======
 Mean: SendToElevator1 - eval: 36
 ======
 SimulationStep=4 - SimulationTime=23/01/2014 10:08:00 - manager is sending
 to elevator 1

Passo 5 da simulação: 23/01/2014 10:10:00

Elevator1 Simulation time:23/01/2014 10:10:00 :
 Trip [time=23/01/2014 10:10:00, from=3, to=[7], numberOfPassengers=2]
 Elevator1 is sending his schedule: Trip [time=23/01/2014 10:10:00, from=3, to=[7], numberOfPassengers=2]
 manager Simulation time:23/01/2014 10:10:00 :
 Trip [time=23/01/2014 10:10:00, from=4, to=[13], numberOfPassengers=1]
 managerChecking if there is an elevator schedule: 23/01/2014 10:10:00-
 istar.impl.Resource:ElevatorPosition
 manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip
 [time=23/01/2014 10:10:00, from=3, to=[7], numberOfPassengers=2]}
 manager >>> Starting UsersAndElevatorsDataInterpretation
 UsersTrip:Trip [time=23/01/2014 10:10:00, from=4, to=[13],
 numberOfPassengers=1]
 ======
 Mean: SendToElevator1 - eval: 31
 ======
 SimulationStep=5 - SimulationTime=23/01/2014 10:10:00 - manager is sending
 to elevator 1

Passo 6 da simulação: 23/01/2014 10:12:00

Elevator1 Simulation time:23/01/2014 10:12:00 :
 Trip [time=23/01/2014 10:12:00, from=4, to=[4], numberOfPassengers=0]
 Elevator1 is sending his schedule: Trip [time=23/01/2014 10:12:00, from=4, to=[4], numberOfPassengers=0]
 manager Simulation time:23/01/2014 10:12:00 :
 Trip [time=23/01/2014 10:12:00, from=11, to=[3], numberOfPassengers=1]
 managerChecking if there is an elevator schedule: 23/01/2014 10:12:00-
 istar.impl.Resource:ElevatorPosition

```
manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip  
[time=23/01/2014 10:12:00, from=4, to=[4], numberOfPassengers=0]}  
manager >>> Starting UsersAndElevatorsDataInterpretation  
UsersTrip:Trip [time=23/01/2014 10:12:00, from=11, to=[3],  
numberOfPassengers=1]  
=====  
Mean: SendToElevator1 - eval: 26  
=====  
SimulationStep=6 - SimulationTime=23/01/2014 10:12:00 - manager is sending  
to elevator 1
```

Passo 7 da simulação: 23/01/2014 10:15:00

```
Elevator1 Simulation time:23/01/2014 10:15:00 :  
Trip [time=23/01/2014 10:15:00, from=1, to=[12], numberOfPassengers=4]  
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:15:00, from=1,  
to=[12], numberOfPassengers=4]  
manager Simulation time:23/01/2014 10:15:00 :  
Trip [time=23/01/2014 10:15:00, from=6, to=[1], numberOfPassengers=1]  
managerChecking if there is an elevator schedule: 23/01/2014 10:15:00-  
istar.impl.Resource:ElevatorPosition  
manager@10.26.144.42:1099/JADE received 1 message(s): {Elevator1=Trip  
[time=23/01/2014 10:15:00, from=1, to=[12], numberOfPassengers=4]}  
manager >>> Starting UsersAndElevatorsDataInterpretation  
UsersTrip:Trip [time=23/01/2014 10:15:00, from=6, to=[1],  
numberOfPassengers=1]  
=====  
Mean: SendToElevator1 - eval: 19  
=====  
SimulationStep=7 - SimulationTime=23/01/2014 10:15:00 - manager is sending  
to elevator 1
```

5.2.3. Cenário com dois elevadores

No cenário com dois elevadores, a descrição do problema é idêntica a do cenário com um elevador.

O modelo i* para o cenário com dois elevadores é apresentado na Figura 38, seguido da especificação do requisito de consciência, com as diferenças em relação ao cenário com apenas um elevador destacadas em fonte cor vermelha.

Awareness[User request and Elevators]		
Topic description:	The agent should be aware of user request (for transport between different floors) and elevators current position and schedules in order to decide which elevator will serve the user request.	
Goal:	Users demands be managed automatically	
Awareness subtype:	User awareness; Position (elevators)	
Suggested operationalizations:	Use system user interface that allow users to inform their requests, including destination; Communicate with automation elevator system	
Alternative actions:	Send the request to elevator1; Send the request to elevator2	
Entity:	Users and elevators	
Source of entity data:	User request devices and elevators	
ContextDescription	{UserRequest, ElevatorSchedule}	
Domains of variable		
Variable name	Domain	
UserRequest	{Trip}	
ElevatorSchedule	{Trip}	
Trip: is a data entity with attributes: number of users (integer); source floor (integer between 1 and 15); destination floor (integer between 1 and 15).		
Total distance (in number of floors to be covered): the distance that the elevator will go to serve the route for which has been set at a given time and the client's request.		
Context situations especification		
Situation name	Specification	
Elevator1 is the best option	Elevator1 will travel the minor distance to meet the user request	
Elevator2 is the best option	Elevator2 will travel the minor distance to meet the user request	
Alternative action choice		
Situation	Alternative action	Impact
Elevator1 is the best option	SendToElevator1	MAKE (strongly positive)
Elevator2 is the best option	SendToElevator2	MAKE (strongly positive)

Os dados para simulação com dois elevadores são apresentados na Tabela 19 abaixo.

Simulation Step	Time	UserRequest (from-to)	Elevator1 (from-to)	Elevator2 (from-to)
1	23/01/2014 10:00:00	1-10	1-1	1-1
2	23/01/2014 10:02:00	12-5	2-5	1-1
3	23/01/2014 10:05:00	15-1	8-2	1-1
4	23/01/2014 10:08:00	7-9	10-10	1-1
5	23/01/2014 10:10:00	4-13	3-7	1-1
6	23/01/2014 10:12:00	11-3	4-4	1-1
7	23/01/2014 10:15:00	6-1	1-12	1-1

Tabela 19 - Dados para simulação do cenário com dois elevadores

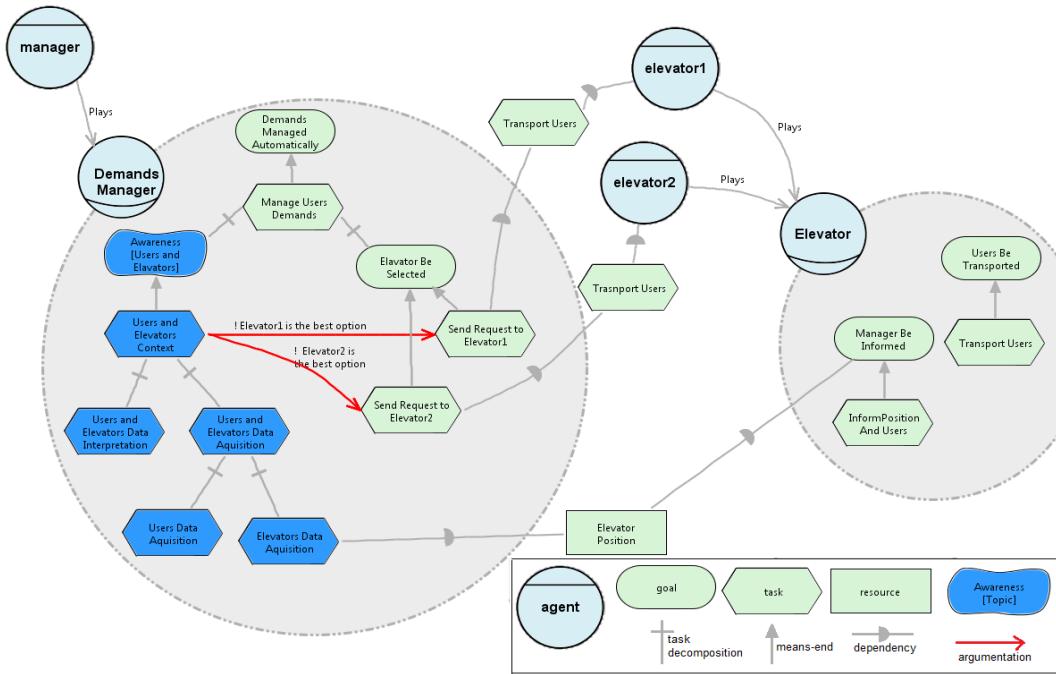


Figura 38 - Modelo i* para o cenário com dois elevadores

Nessa simulação, consideramos em todos os passos o elevador2 está parado no andar 1. Segue o resultado da execução dos passos da simulação.

Passo 1 da simulação: 23/01/2014 10:00:00

Elevator2 Simulation time:23/01/2014 10:00:00 :

Trip [time=23/01/2014 10:00:00, from=1, to=[1], numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:00:00, from=1, to=[1],
numberOfPassengers=0]

Elevator1 Simulation time:23/01/2014 10:00:00 :

Trip [time=23/01/2014 10:00:00, from=1, to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:00:00, from=1, to=[1],
numberOfPassengers=0]

manager Simulation time:23/01/2014 10:00:00 :

managerChecking if there is an elevator schedule: 23/01/2014 10:00:00-

istar.impl.Resource:ElevatorPosition

manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip

[time=23/01/2014 10:00:00, from=1, to=[1], numberOfPassengers=0],

Elevator2=Trip [time=23/01/2014 10:00:00, from=1, to=[1],
numberOfPassengers=0]}

manager >>> Starting UsersAndElevatorsDataInterpretation

UsersTrip:Trip [time=23/01/2014 10:00:00, from=1, to=[10],

numberOfPassengers=1]

=====

Mean:SendToElevator1 - eval: 32

Mean:SendToElevator2 - eval: 32

=====

SimulationStep=1 - SimulationTime=23/01/2014 10:00:00 - manager is sending to elevator 1

No passo 1, como os dois elevadores se encontram parados no mesmo andar, a escolha é aleatória.

Passo 2 da simulação: 23/01/2014 10:02:00

Elevator1 Simulation time:23/01/2014 10:02:00 :

Trip [time=23/01/2014 10:02:00, from=2, to=[5], numberOfPassengers=1]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:02:00, from=2, to=[5],
numberOfPassengers=1]

Elevator2 Simulation time:23/01/2014 10:02:00 :

Trip [time=23/01/2014 10:02:00, from=1, to=[1], numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:02:00, from=1, to=[1],
numberOfPassengers=0]

manager Simulation time:23/01/2014 10:02:00 :

managerChecking if there is an elevator schedule: 23/01/2014 10:02:00-

istar.impl.Resource:ElevatorPosition

manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip
[time=23/01/2014 10:02:00, from=2, to=[5], numberOfPassengers=1],

Elevator2=Trip [time=23/01/2014 10:02:00, from=1, to=[1],
numberOfPassengers=0]}

manager >>> Starting UsersAndElevatorsDataInterpretation

UsersTrip:Trip [time=23/01/2014 10:02:00, from=12, to=[5],
numberOfPassengers=1]

=====

Mean:SendToElevator1 - eval: 24

Mean:SendToElevator2 - eval: 23

=====

SimulationStep=2 - SimulationTime=23/01/2014 10:02:00 - manager is sending to elevator 1

Passo 3 da simulação: 23/01/2014 10:05:00

Elevator2 Simulation time:23/01/2014 10:05:00 :

Trip [time=23/01/2014 10:05:00, from=1, to=[1], numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:05:00, from=1, to=[1],
numberOfPassengers=0]

manager Simulation time:23/01/2014 10:05:00 :

managerChecking if there is an elevator schedule: 23/01/2014 10:05:00

istar.impl.Resource:ElevatorPosition

Elevator1 Simulation time:23/01/2014 10:05:00 :

Trip [time=23/01/2014 10:05:00, from=8, to=[2], numberOfPassengers=3]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:05:00, from=8, to=[2],
numberOfPassengers=3]

manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip

[time=23/01/2014 10:05:00, from=8, to=[2], numberOfPassengers=3],

Elevator2=Trip [time=23/01/2014 10:05:00, from=1, to=[1],
numberOfPassengers=0]}

manager >>> Starting UsersAndElevatorsDataInterpretation

UsersTrip:Trip [time=23/01/2014 10:05:00, from=15, to=[1],

numberOfPassengers=1]

=====

Mean:SendToElevator1 - eval: 8

Mean:SendToElevator2 - eval: 13

=====

SimulationStep=3 - SimulationTime=23/01/2014 10:05:00 - manager is sending to
elevator 2

Passo 4 da simulação: 23/01/2014 10:08:00

Elevator2 Simulation time:23/01/2014 10:08:00 :

Trip [time=23/01/2014 10:08:00, from=1, to=[1], numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:08:00, from=1, to=[1],
numberOfPassengers=0]

manager Simulation time:23/01/2014 10:08:00 :

managerChecking if there is an elevator schedule: 23/01/2014 10:08:00-

istar.impl.Resource:ElevatorPosition

Elevator1 Simulation time:23/01/2014 10:08:00 :

```

Trip [time=23/01/2014 10:08:00, from=10, to=[10], numberOfPassengers=0]
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:08:00, from=10,
to=[10], numberOfPassengers=0]
manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip
[time=23/01/2014 10:08:00, from=10, to=[10], numberOfPassengers=0],
Elevator2=Trip [time=23/01/2014 10:08:00, from=1, to=[1],
numberOfPassengers=0]}
manager >>> Starting UsersAndElevatorsDataInterpretation
UsersTrip:Trip [time=23/01/2014 10:08:00, from=7, to=[9],
numberOfPassengers=1]
=====
Mean:SendToElevator1 - eval: 36
Mean:SendToElevator2 - eval: 33
=====
SimulationStep=4 - SimulationTime=23/01/2014 10:08:00 - manager is sending to
elevator 1

```

Passo 5 da simulação: 23/01/2014 10:10:00

```

Elevator1 Simulation time:23/01/2014 10:10:00 :
Trip [time=23/01/2014 10:10:00, from=3, to=[7], numberOfPassengers=2]
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:10:00, from=3, to=[7],
numberOfPassengers=2]
manager Simulation time:23/01/2014 10:10:00 :
Trip [time=23/01/2014 10:10:00, from=4, to=[13], numberOfPassengers=1]
managerChecking if there is an elevator schedule: 23/01/2014 10:10:00
istar.impl.Resource:ElevatorPosition
Elevator2 Simulation time:23/01/2014 10:10:00 :
Trip [time=23/01/2014 10:10:00, from=1, to=[1], numberOfPassengers=0]
Elevator2 is sending his schedule: Trip [time=23/01/2014 10:10:00, from=1, to=[1],
numberOfPassengers=0]
manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip
[time=23/01/2014 10:10:00, from=3, to=[7], numberOfPassengers=2],
Elevator2=Trip [time=23/01/2014 10:10:00, from=1, to=[1],
numberOfPassengers=0]}
manager >>> Starting UsersAndElevatorsDataInterpretation
=====
```

Mean:SendToElevator1 - eval: 31

Mean:SendToElevator2 - eval: 29

=====

SimulationStep=5 - SimulationTime=23/01/2014 10:10:00 - manager is sending to elevator 1

Passo 6 da simulação: 23/01/2014 10:12:00

Elevator1 Simulation time:23/01/2014 10:12:00 :

Trip [time=23/01/2014 10:12:00, from=4, to=[4], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:12:00, from=4, to=[4],
numberOfPassengers=0]

manager Simulation time:23/01/2014 10:12:00 :

managerChecking if there is an elevator schedule: 23/01/2014 10:12:00-

istar.impl.Resource:ElevatorPosition

Elevator2 Simulation time:23/01/2014 10:12:00 :

Trip [time=23/01/2014 10:12:00, from=1, to=[1], numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:12:00, from=1, to=[1],
numberOfPassengers=0]

manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip
[time=23/01/2014 10:12:00, from=4, to=[4], numberOfPassengers=0],

Elevator2=Trip [time=23/01/2014 10:12:00, from=1, to=[1],
numberOfPassengers=0]}

manager >>> Starting UsersAndElevatorsDataInterpretation

UsersTrip:Trip [time=23/01/2014 10:12:00, from=11, to=[3],
numberOfPassengers=1]

=====

Mean:SendToElevator1 - eval: 26

Mean:SendToElevator2 - eval: 23

=====

SimulationStep=5 - SimulationTime=23/01/2014 10:12:00 - manager is sending to elevator 1

Passo 7 da simulação: 23/01/2014 10:15:00

Elevator1 Simulation time:23/01/2014 10:15:00 :

```
Trip [time=23/01/2014 10:15:00, from=1, to=[12], numberOfPassengers=4]
manager Simulation time:23/01/2014 10:15:00 :
Elevator1 is sending his schedule: Trip [time=23/01/2014 10:15:00, from=1,
to=[12], numberOfPassengers=4]
managerChecking if there is an elevator schedule: 23/01/2014 10:15:00-
istar.impl.Resource:ElevatorPosition
Elevator2 Simulation time:23/01/2014 10:15:00 :
Trip [time=23/01/2014 10:15:00, from=1, to=[1], numberOfPassengers=0]
Elevator2 is sending his schedule: Trip [time=23/01/2014 10:15:00, from=1, to=[1],
numberOfPassengers=0]
manager@10.26.144.42:1099/JADE received 2 message(s): {Elevator1=Trip
[time=23/01/2014 10:15:00, from=1, to=[12], numberOfPassengers=4],
Elevator2=Trip [time=23/01/2014 10:15:00, from=1, to=[1],
numberOfPassengers=0]}
manager >>> Starting UsersAndElevatorsDataInterpretation
UsersTrip:Trip [time=23/01/2014 10:15:00, from=6, to=[1],
numberOfPassengers=1]
=====
Mean:SendToElevator1 - eval: 19
Mean:SendToElevator2 - eval: 31
=====
SimulationStep=6 - SimulationTime=23/01/2014 10:15:00 - manager is sending to
elevator 2
```

5.2.4. Cenário com três elevadores

No cenário com três elevadores, a descrição do problema é idêntica a do cenário com um e dois elevadores.

O modelo i* para o cenário com três elevadores é apresentado na Figura 39, seguido da especificação do requisito de consciência, com as diferenças em relação ao cenário com apenas um elevador destacadas em fonte cor vermelha, a seguir.

Os dados para simulação com três elevadores são apresentados na Tabela 20 a seguir.

Step	Time	UserRequest (from-to)	Elevator1 (from-to)	Elevator2 (from-to)	Elevator3 (from-to)
1	23/01/2014 10:00:00	1-10	1-1	1-1	15-1
2	23/01/2014 10:02:00	12-5	2-5	1-1	15-1
3	23/01/2014 10:05:00	15-1	8-2	1-1	15-1
4	23/01/2014 10:08:00	7-9	10-10	1-1	15-1
5	23/01/2014 10:10:00	4-13	3-7	1-1	15-1
6	23/01/2014 10:12:00	11-3	4-4	1-1	15-1
7	23/01/2014 10:15:00	6-1	1-12	1-1	15-1

Tabela 20 - Dados para simulação com três elevadores

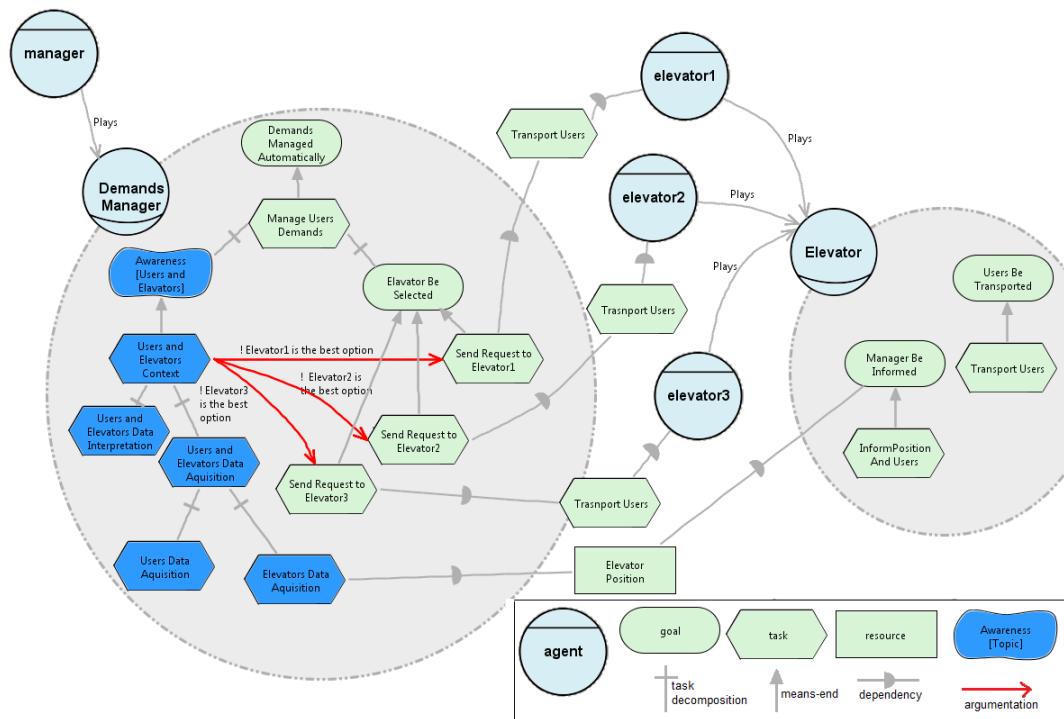


Figura 39 - Modelo i* para o cenário com três elevadores

Awareness[User request and Elevators]				
Topic description:	The agent should be aware of user request (for transport between different floors) and elevators current position and schedule in order to decide which elevator will serve the user request.			
Goal:	Users demands be managed automatically			
Awareness subtype:	User awareness; Position (elevators)			
Suggested operationalizations:	Use system user interface that allow users to inform their requests, including destination; Communicate with automation elevator system			
Alternative actions:	Send the request to elevator1; Send the request to elevator2; Send the request to elevator3;			
Entity:	Users and elevators			
Source of entity data:	User request devices and elevators			
ContextDescription	{UserRequest, ElevatorSchedule}			
Domains of variable				
Variable name	Domain			
UserRequest	{Trip}			
ElevatorSchedule	{Trip}			
Trip: is a data entity with attributes: number of users (integer); source floor (integer between 1 and 15); destination floor (integer between 1 and 15).				
Total distance (in number of floors to be covered): the distance that the elevator will go to serve the route for which has been set at a given time and the client's request.				
Context situations especification				
Situation name	Specification			
Elevator1 is the best option	Elevator1 will travel the minor distance to meet the user request			
Elevator2 is the best option	Elevator2 will travel the minor distance to meet the user request			
Elevator3 is the best option	Elevator3 will travel the minor distance to meet the user request			
Alternative action choice				
Situation	Alternative action	Impact		
Elevator1 is the best option	SendToElevator1	MAKE (strongly positive)		
Elevator2 is the best option	SendToElevator2	MAKE (strongly positive)		
Elevator3 is the best option	SendToElevator3	MAKE (strongly positive)		

Nessa simulação, consideramos em todos os passos o elevador2 está parado no andar 1 e o elevedar3 está indo do andar 15 para o andar 1.

Segue o resultado da simulação.

Passo 1 da simulação: 23/01/2014 10:00:00

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:00:00, from=1, to=[1],
numberOfPassengers=0]

Elevator3 is sending his schedule: Trip [time=23/01/2014 10:00:00, from=15,
to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:00:00, from=1, to=[1],
numberOfPassengers=0]

managerChecking if there is an elevator schedule: 23/01/2014 10:00:00

manager >>> Starting UsersAndElevatorsDataInterpretation

manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014
10:00:00, from=1, to=[1], numberOfPassengers=0], Elevator2=Trip
[time=23/01/2014 10:00:00, from=1, to=[1], numberOfPassengers=0],
Elevator3=Trip [time=23/01/2014 10:00:00, from=15, to=[1],
numberOfPassengers=0]}

manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014
10:00:00, from=1, to=[10], numberOfPassengers=1]

=====

manager evaluates the mean:SendToElevator1 as contribuition_value=32

manager evaluates the mean:SendToElevator2 as contribuition_value=32

manager evaluates the mean:ToElevator3 as contribuition_value=18

The mean:SendToElevator2 has been selected in order to satisfy/satisficy the
end:ElavatorBeSelected

=====

SimulationStep=1 - SimulationTime=23/01/2014 10:00:00 - manager is sending to
elevator 2

Passo 2 da simulação: 23/01/2014 10:02:00

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:02:00, from=1, to=[1],
numberOfPassengers=0]

Elevator3 is sending his schedule: Trip [time=23/01/2014 10:02:00, from=15,
to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:02:00, from=2, to=[5],
numberOfPassengers=1]

managerChecking if there is an elevator schedule: 23/01/2014 10:02:00

manager >>> Starting UsersAndElevatorsDataInterpretation
 manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014 10:02:00, from=2, to=[5], numberOfPassengers=1], Elevator2=Trip [time=23/01/2014 10:02:00, from=1, to=[1], numberOfPassengers=0], Elevator3=Trip [time=23/01/2014 10:02:00, from=15, to=[1], numberOfPassengers=0]}
 manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014 10:02:00, from=12, to=[5], numberOfPassengers=1]
 =====
 manager evaluates the mean:SendToElevator1 as contribuition_value=24
 manager evaluates the mean:SendToElevator2 as contribuition_value=23
 manager evaluates the mean:ToElevator3 as contribuition_value=27
 The mean:ToElevator3 has been selected in order to satisfy/satisfy the end:ElavatorBeSelected
 =====
 SimulationStep=2 - SimulationTime=23/01/2014 10:02:00 - manager is sending to elevator 3

Passo 3 da simulação: 23/01/2014 10:05:00

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:05:00, from=1, to=[1], numberOfPassengers=0]
 Elevator3 is sending his schedule: Trip [time=23/01/2014 10:05:00, from=15, to=[1], numberOfPassengers=0]
 Elevator1 is sending his schedule: Trip [time=23/01/2014 10:05:00, from=8, to=[2], numberOfPassengers=3]
 managerChecking if there is an elevator schedule: 23/01/2014 10:05:00
 manager >>> Starting UsersAndElevatorsDataInterpretation
 manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014 10:05:00, from=8, to=[2], numberOfPassengers=3], Elevator2=Trip [time=23/01/2014 10:05:00, from=1, to=[1], numberOfPassengers=0], Elevator3=Trip [time=23/01/2014 10:05:00, from=15, to=[1], numberOfPassengers=0]}\n
 manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014 10:05:00, from=15, to=[1], numberOfPassengers=1]
 =====
 manager evaluates the mean:SendToElevator1 as contribuition_value=8
 manager evaluates the mean:SendToElevator2 as contribuition_value=13

manager evaluates the mean:ToElevator3 as contribuition_value=27
The mean:ToElevator3 has been selected in order to satisfy/satisfy the end:ElavatorBeSelected

=====

SimulationStep=3 - SimulationTime=23/01/2014 10:05:00 - manager is sending to elevator 3

Passo 4 da simulação: 23/01/2014 10:08:00

Elevator3 is sending his schedule: Trip [time=23/01/2014 10:08:00, from=15, to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:08:00, from=10, to=[10], numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:08:00, from=1, to=[1], numberOfPassengers=0]

managerChecking if there is an elevator schedule: 23/01/2014 10:08:00

manager >>> Starting UsersAndElevatorsDataInterpretation

manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014 10:08:00, from=10, to=[10], numberOfPassengers=0], Elevator2=Trip [time=23/01/2014 10:08:00, from=1, to=[1], numberOfPassengers=0], Elevator3=Trip [time=23/01/2014 10:08:00, from=15, to=[1], numberOfPassengers=0]}

manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014 10:08:00, from=7, to=[9], numberOfPassengers=1]

=====

manager evaluates the mean:SendToElevator1 as contribuition_value=36

manager evaluates the mean:SendToElevator2 as contribuition_value=33

manager evaluates the mean:ToElevator3 as contribuition_value=19

The mean:SendToElevator1 has been selected in order to satisfy/satisfy the end:ElavatorBeSelected

=====

SimulationStep=4 - SimulationTime=23/01/2014 10:08:00 - manager is sending to elevator 1

Passo 5 da simulação: 23/01/2014 10:10:00

Elevator3 is sending his schedule: Trip [time=23/01/2014 10:10:00, from=15, to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:10:00, from=3, to=[7],
numberOfPassengers=2]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:10:00, from=1, to=[1],
numberOfPassengers=0]

managerChecking if there is an elevator schedule: 23/01/2014 10:10:00

manager@10.26.144.42:1099/JADE received 3 message(s): {Elevator1=Trip
[time=23/01/2014 10:10:00, from=3, to=[7], numberOfPassengers=2],
Elevator2=Trip [time=23/01/2014 10:10:00, from=1, to=[1],
numberOfPassengers=0], Elevator3=Trip [time=23/01/2014 10:10:00, from=15,
to=[1], numberOfPassengers=0]}

manager >>> Starting UsersAndElevatorsDataInterpretation

manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014
10:10:00, from=3, to=[7], numberOfPassengers=2], Elevator2=Trip
[time=23/01/2014 10:10:00, from=1, to=[1], numberOfPassengers=0],
Elevator3=Trip [time=23/01/2014 10:10:00, from=15, to=[1],
numberOfPassengers=0]}

manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014
10:10:00, from=4, to=[13], numberOfPassengers=1]

=====

manager evaluates the mean:SendToElevator1 as contribuition_value=31

manager evaluates the mean:SendToElevator2 as contribuition_value=29

manager evaluates the mean:ToElevator3 as contribuition_value=15

The mean:SendToElevator1 has been selected in order to satisfy/satisfy the
end: ElevatorBeSelected

=====

SimulationStep=5 - SimulationTime=23/01/2014 10:10:00 - manager is sending to
elevator 1

Passo 6 da simulação: 23/01/2014 10:12:00

Elevator3 is sending his schedule: Trip [time=23/01/2014 10:12:00, from=15,
to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:12:00, from=4, to=[4],
numberOfPassengers=0]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:12:00, from=1, to=[1],
numberOfPassengers=0]

managerChecking if there is an elevator schedule: 23/01/2014 10:12:00

manager >>> Starting UsersAndElevatorsDataInterpretation

manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014 10:12:00, from=4, to=[4], numberOfPassengers=0], Elevator2=Trip [time=23/01/2014 10:12:00, from=1, to=[1], numberOfPassengers=0], Elevator3=Trip [time=23/01/2014 10:12:00, from=15, to=[1], numberOfPassengers=0]}

manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014 10:12:00, from=11, to=[3], numberOfPassengers=1]

manager evaluates the mean:SendToElevator1 as contribuition_value=26

manager evaluates the mean:SendToElevator2 as contribuition_value=23

manager evaluates the mean:ToElevator3 as contribuition_value=27

The mean:ToElevator3 has been selected in order to satisfy/satisfy the end: ElevatorBeSelected

SimulationStep=6 - SimulationTime=23/01/2014 10:12:00 - manager is sending to elevator 3

Passo 7 da simulação: 23/01/2014 10:15:00

Elevator3 is sending his schedule: Trip [time=23/01/2014 10:15:00, from=15, to=[1], numberOfPassengers=0]

Elevator1 is sending his schedule: Trip [time=23/01/2014 10:15:00, from=1, to=[12], numberOfPassengers=4]

Elevator2 is sending his schedule: Trip [time=23/01/2014 10:15:00, from=1, to=[1], numberOfPassengers=0]

managerChecking if there is an elevator schedule: 23/01/2014 10:15:00

manager >>> Starting UsersAndElevatorsDataInterpretation

manager perceives the underlying situation: {Elevator1=Trip [time=23/01/2014 10:15:00, from=1, to=[12], numberOfPassengers=4], Elevator2=Trip [time=23/01/2014 10:15:00, from=1, to=[1], numberOfPassengers=0], Elevator3=Trip [time=23/01/2014 10:15:00, from=15, to=[1], numberOfPassengers=0]}

manager perceives the underlying situation: UserRequest=Trip [time=23/01/2014 10:15:00, from=6, to=[1], numberOfPassengers=1]

manager evaluates the mean:SendToElevator1 as contribuition_value=19

manager evaluates the mean:SendToElevator2 as contribuition_value=31

manager evaluates the mean:ToElevator3 as contribuition_value=27

The mean:SendToElevator2 has been selected in order to satisfy/satisficy the end:ElavatorBeSelected

Elevator2

=====

SimulationStep=7 - SimulationTime=23/01/2014 10:15:00 - manager is sending to elevator 2

Os cenários deste caso foram apresentados em ordem crescente de complexidade da escolha de alternativa. No cenário com apenas um elevador não havia escolha a ser feita de fato. No cenário com 2, e depois com 3 elevadores, a escolha é influenciada por dados de contexto de usuários (solicitação de viagem) e dados de contexto sobre os recursos disponíveis (posição e viagem agendada de cada elevador).

Embora tenhamos simulado o uso de até três elevadores, a modelagem e implementação de cenários com mais elevadores pode ser feita de forma similar.

Caso se deseje implementar cenários em que o sistema precise gerenciar solicitações de viagem de mais de um usuário simultaneamente, basta criar uma fila de solicitações, atualizando as agendas de viagem de cada elevador que for escolhido para atender a uma solicitação antes de escolher o elevador que deve atender a próxima solicitação.

O resultado alcançado com a simulação do sistema multiagente implementando o requisito de consciência especificado foi satisfatório.

5.3.

Caso 3 – Sistema multiagente para operação na bolsa de valores

5.3.1.

Descrição do problema

A meta principal do sistema é obter lucro, a partir de um montante inicial de dinheiro, através da realização de operações de compra e venda em um mesmo dia (*intraday*) de um determinado papel no mercado de ações da bolsa de valores. Para isto, o sistema deve perceber a movimentação do mercado e decidir, de forma autônoma, quando e qual operação (compra ou venda) deve ser executada. No caso que apresentaremos a seguir, o sistema deverá operar através da compra e venda de opções (de compra) de ações da Vale. Descreveremos melhor os dados usados na simulação mais adiante.

5.3.2. Considerações sobre o problema

Diferente dos casos do sistema para controle automático de portas e do sistema para controle de elevadores, apresentados anteriormente, o sistema para operação autônoma na bolsa de valores possui uma complexidade de classe superior aos primeiros. Do ponto de vista do requisito de consciência, os primeiros casos são de detecção de situações subjacentes a partir das quais o sistema deve tomar alguma (re)ação para alcançar suas metas. Nesses casos, a aquisição e análise dos dados podem ser feitas isoladamente, evento a evento, sem que um evento interfira de maneira significativa nas análises dos eventos posteriores. Além disso, a interpretação dos dados adquiridos pode ser representada por uma função direta definida para cada conjunto de valores de variáveis previstas. Ou seja, dado um conjunto de variáveis usadas para descrever uma situação (*Context Description*) e os valores observados de cada variável é possível identificar qual a situação subjacente, possivelmente com certeza absoluta (100%).

A operação autônoma na bolsa de valores é um caso de prognóstico, onde a partir das situações subjacentes, procura-se inferir que situações irão ocorrer no futuro. O que se busca é perceber as correlações entre as situações subsequentes, para que se possa inferir que situações ocorrerão no futuro a partir de situações correntes (recentes). A complexidade desses casos (prognóstico) é maior que a dos primeiros (detecção), como mostra a Figura 40, obtida de (Laney e Kart, 2012).

Em alguns casos, como o do mercado de ações, podem ser aplicadas técnicas de aprendizado de máquinas (*machine learning*), baseadas em métodos estatísticos, para inferir essas correlações. Nestes casos, haverá uma probabilidade (grau de acerto) associada à ocorrência das situações futuras.

Uma das possibilidades é uso de técnicas de aprendizado supervisionado. Neste caso, dado, um conjunto de exemplos, também denominado *corpus*, rotulados na forma (x_i, y_i) , em que x_i representa um exemplo e y_i denota o seu rótulo, deve-se produzir um classificador, também denominado modelo, capaz de predizer precisamente o rótulo de novos dados. É fundamental que o *corpus* seja estatisticamente representativo para que se possa gerar modelos com grau de acerto relevante. Esse processo de indução de um classificador a partir de uma amostra de dados é denominado treinamento. O classificador obtido também pode ser visto como uma função f , a qual recebe um dado x e fornece uma predição y (Lorena e Carvalho, 2007).

Cada exemplo, também referenciado por dado ou caso, é tipicamente representado por um vetor de características. Cada característica, também denominada atributo, expressa um determinado aspecto do exemplo. Os rótulos ou classes representam o fenômeno de interesse sobre o qual se deseja fazer previsões.

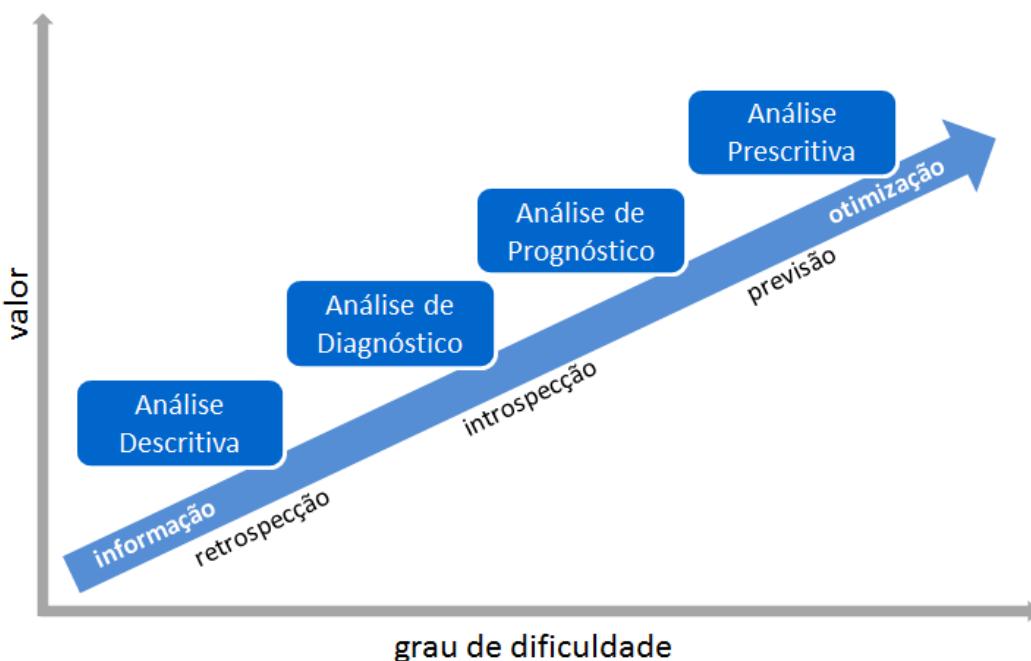


Figura 40 - Modelo de ascendência analítica do Gartner (Laney e Kart, 2012)

5.3.3.

Construção do modelo de aprendizado supervisionado

Inicialmente, há dois pontos fundamentais que precisam ser definidos para o treinamento de modelo: o *corpus* que será, ou seja, o conjunto de dados (exemplos com os respectivos valores de atributos) e a técnica de aprendizado.

O *corpus* usado é constituído de 52299 exemplos representando dados consolidados minuto a minuto sobre as operações da ação VALE5 na Bovespa entre 13/04/2009 e 14/10/2009. Cada exemplo possui como principais atributos: Instante (dd/MM/YYYY hh:mm), valor negociado, volume, valor mínimo e valor máximo negociado nos últimos minutos.

A Figura 42 apresenta a curva com os preços negociados da ação no período usado para treinamento.

A técnica de aprendizado supervisionado usada para a construção do modelo foi support vector machine - SVM regressiva (Vapnik, 1995) – como os rótulos são valores contínuos e não classes, trata-se de um problema de regressão e não de classificação. Uma das vantagens de usar SVMs é que elas

são robustas diante de dados de grande dimensão, sobre os quais outras técnicas de aprendizado comumente obtêm classificadores super ou sub ajustados. Outra característica atrativa é a convexidade do problema de otimização formulado em seu treinamento, que implica na existência de um único mínimo global. Essa é uma vantagem das SVMs sobre, por exemplo, as Redes Neurais Artificiais (RNAs) Perceptron Multicamadas (Multilayer Perceptron), em que há mínimos locais na função objetivo minimizada (Lorena e Carvalho, 2007).

Foi construído um modelo que gera previsões de tendência a cada minuto. Essa tendência pode ser de três tipos: tendência de alta, tendência de baixa ou neutra (sem tendência). Com base nesta tendência (que representa como o agente percebe e interpreta o mercado, ou seja, se torna consciente), o agente decide a cada minuto como agir em função desta consciência. As decisões tomadas pelo agente seguem da máquina de estados finita apresentada na Figura 41.

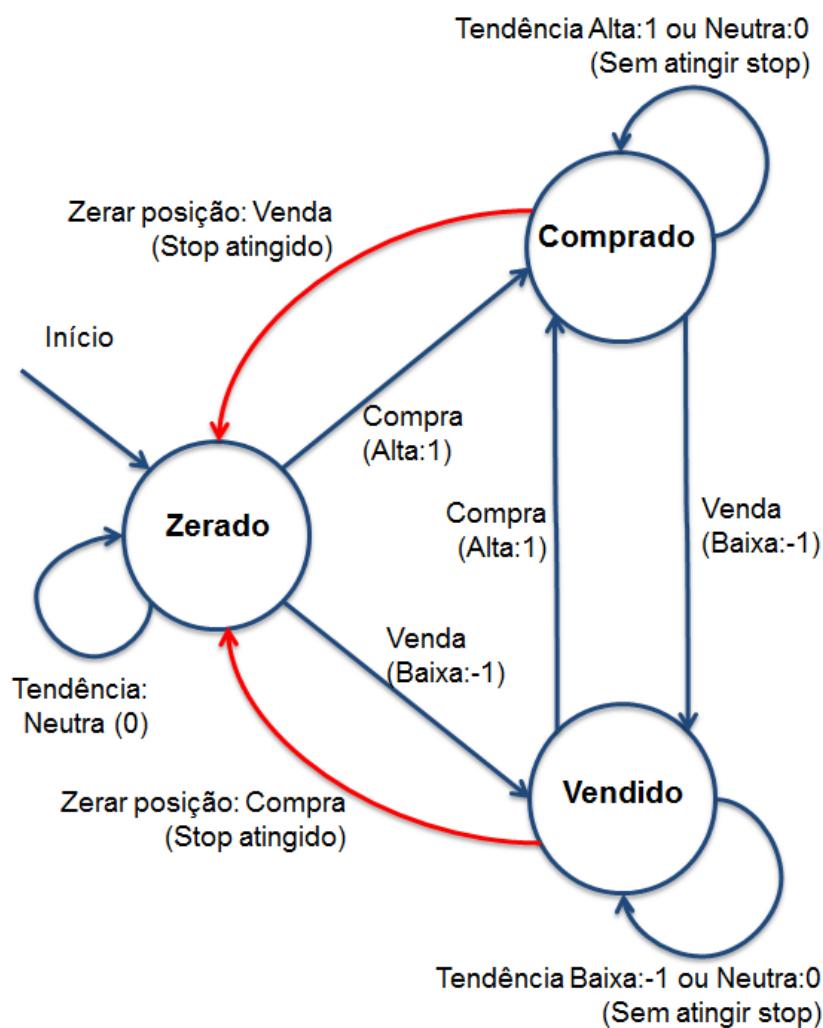


Figura 41 – Máquina de estados do modelo de decisão do Operador

Sempre que percebe uma tendência de alta no papel VALE5, detectada através da aplicação do modelo de aprendizado previamente treinado aos dados de correntes do mercado (contexto), o sistema efetua uma operação de compra de opções VALEK50 (por apresentarem valores menores com variação maior, caso a expectativa de alta apontada pela tendência se comprove, o lucro da operação tende a ser maior). Ao final da operação o sistema se encontra numa posição denominada “comprado”.

No caso de perceber uma tendência de baixa, o sistema faria a venda coberta de opções VALEK48 (valor superior ao das opções VALEK50). Para realizar uma operação de venda coberta, o sistema precisa comprar opções VALEK50 na mesma quantidade que vender opções VALEK48 e permanecer um valor bloqueado equivalente ao número de opções vendidas vezes a diferença de *strike* (valor de exercício das opções) entre os dois papéis – no caso dos papéis VALEK50 e VALEK48 a diferença de *strike* é igual a R\$ 2. Ao final da operação o sistema se encontra numa posição denominada “vendido”.

Caso o sistema se encontre na posição comprado e perceba uma tendência de baixa, o sistema realiza uma operação que de venda que o leva a ficar na posição vendido. De forma análoga, caso o sistema se encontre na posição vendido e perceba uma tendência de alta, o sistema deve realizar uma operação de compra que o leve a ficar na posição comprado.

Se o sistema estiver na posição comprado e perceber uma tendência de alta, o sistema não deve fazer nada (apenas aguardar o próximo evento). Da mesma forma se o sistema estiver na posição vendido e perceber uma tendência de baixa, o sistema também não deve fazer nada (apenas aguardar o próximo evento).

Assim, além de estar consciente da tendência do mercado o sistema precisa estar consciente de sua própria posição para decidir como agir. O modelo i^* do sistema multiagente é apresentado na Figura 43, a seguir. A descrição em iStarML do modelo i^* de cada cenário, juntamente com o rastro de criação dos agentes, encontra-se no Anexo I.

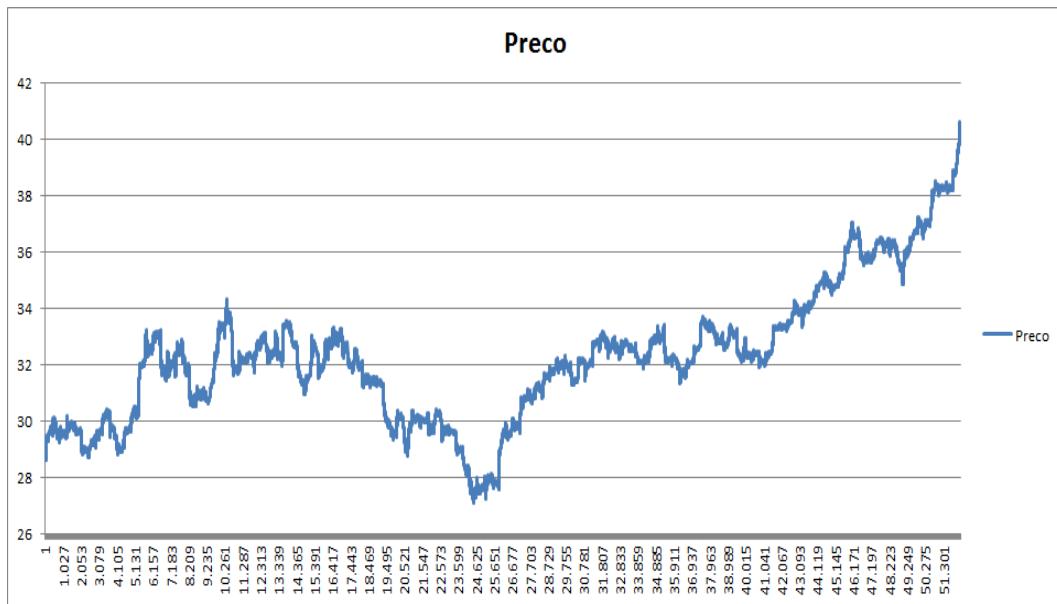


Figura 42 - Valor de VALE5 no intervalo de treinamento (13/04 a 14/10/2009)

5.3.4. Simulação e dados teste

As simulações de operação do sistema multiagente autônomo, foram realizadas com o sistema podendo efetuar as operações de compra e venda de opções⁹ de compra de ações da vale. Os valores das opções apresentam uma variação maior (percentualmente) dentro de um mesmo dia que o valor da ação daquela opção, mas em geram evoluem no mesmo sentido: quando uma ação sobe os valores de suas opções tendem a subir, assim como quando o valor da ação cai os valores das opções tendem a cair.

O modelo i* para o sistema multiagente para operação autônoma na bolsa de valores possui quatro papéis: operador (*Operator*), profeta (*Prophet*), sensor (*Sensor*) e corretor (*Broker*).

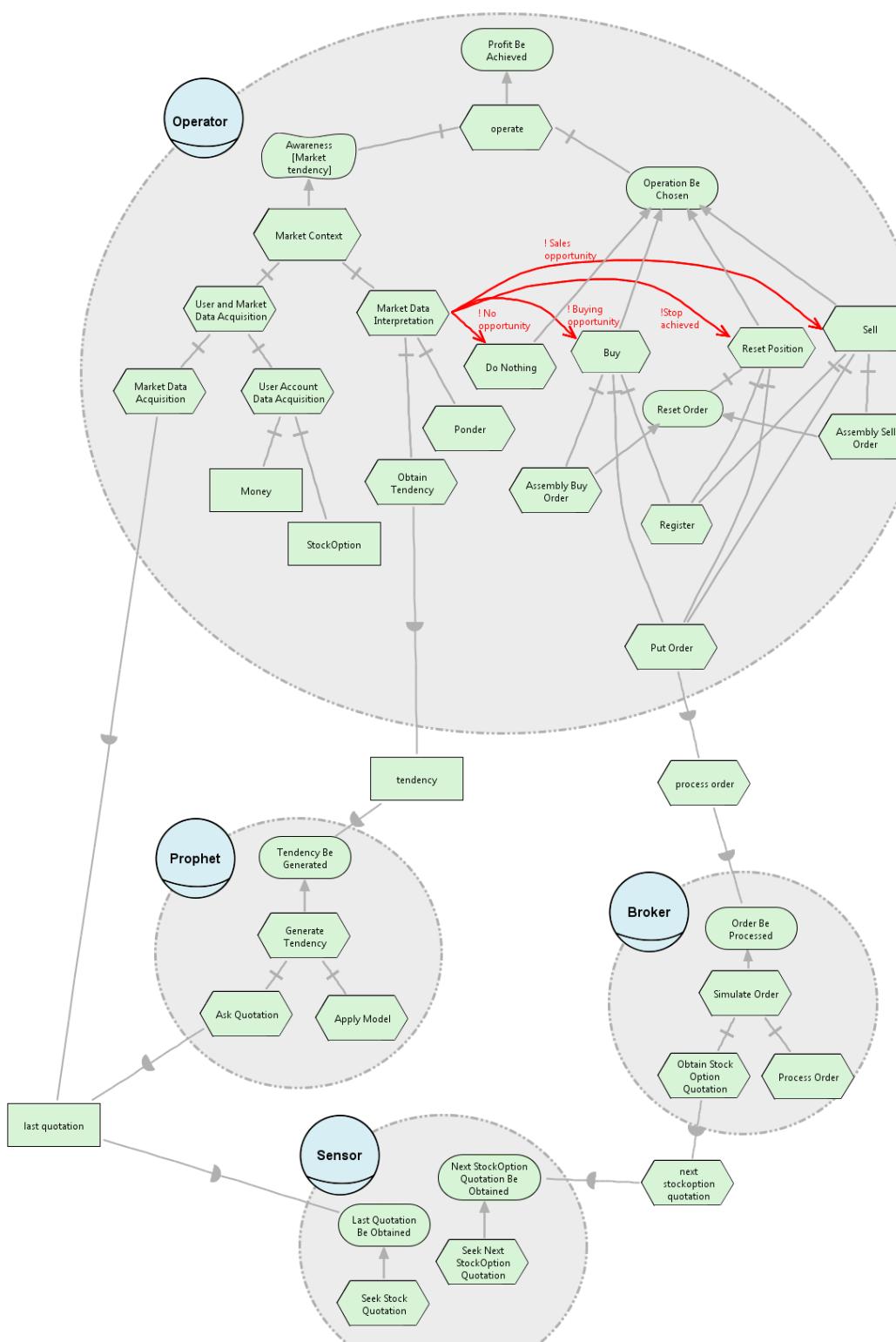
O operador é o papel principal e tem como meta obter lucro (*ProfitBeAchieved*). Para alcançar esta meta, o operador precisa decidir que ação tomar, na expectativa de obter lucro futuro. Para tomar esta decisão o operador precisa ter consciência do contexto atual do mercado, o que pode ser inferido com

⁹ Uma opção de compra de ação representa o direito de exercer a compra da ação por um valor pré-fixado na data de exercício da opção. Nesta data, se o valor da ação estiver acima do valor pré-fixado na opção, o exercício da opção permitirá efetuar uma compra por valor menor. Caso o valor da ação esteja abaixo do valor pré-fixado na opção, a opção terá valor nulo (zero) pois não será exercida.

base nas tendências de mercado que ele recebe do profeta, e na sua posição atual, ou seja, na quantidade de recursos (dinheiro e opções) que ele possui no momento. O profeta, por sua vez tem como meta gerar tendências futuras de mercado que informará ao operador. Para que possa gerar essas tendências, o profeta precisa de informações sobre a cotação atual da ação VALE5 para que possa aplicar seus modelos de aprendizado de máquina.

O sensor é responsável por informar ao profeta (e ao operador) a cotação atual da ação VALE5. Esta cotação deve ser fornecida no mesmo formato dos exemplos usados no treinamento, ou seja, com os seguintes atributos: Instante (dd/MM/YYYY hh:mm), Valor negociado, Volume, Valor mínimo e valor máximo negociado nos últimos minutos.

Uma vez decidido a efetuar uma operação de compra ou venda, o operador solicita ao corretor que efetue tal operação informando sua posição atual (quantidade de dinheiro disponível e bloqueado, e quantidade de opções VALEK48 e VALEK50). O corretor então calcula o resultado da operação solicitada pelo operador com base na cotação do valor das opções no minuto seguinte – consideramos os valores do minuto seguinte na simulação porque os dados refletem a consolidação das operações por minuto. Calculado o resultado da operação, o corretor informa ao operador sua posição após a execução da ordem recebida.

Figura 43 - Modelo i^* do SMA para operação na bolsa

A seguir, é apresentada a especificação do requisito de consciência – sem stop.

Awareness[MarketTendency]		
Topic description:	The agent should be aware of market context, represented by market tendencies to decide which operation to perform in order to obtain future profit. The agent should also be aware of its current position (amount of resources).	
Goal:	Profit be achieved	
Awareness subtype:	Informational environment;	
Suggested operationalizations :	Use machine learning techniques to perceive tendencies	
Alternative actions:	Buy options; Sell options; Do not operate	
Entity:	Market, Account	
Source of entity data:	Stock and change market system	
ContextDescription	{VALE5-DATA, VALEK48, VALEK50, Money}	
Domains of variable		
Variable name	Domain	
Market	{MarketData}	
VALEK48	Integer – quantity of VALEK48 in the account	
VALEK50	Integer – quantity of VALEK50 in the account	
VALE5-DATA attributes: Timestamp; Value (last minute); Volume (last minute); LowValue (last 15 minutes); HighValue (last 15 minutes);		
Context situations especification		
Situation name	Specification	
Buying opportunity	Operator perceives (is informed of) a high tendency	
Sales opportunity	Operator perceives (is informed of) a low tendency	
No opportunity	Operator perceives (is informed of) no tendency	
Alternative action choice		
Situation	Alternative action	Impact
Buying opportunity	BUY	MAKE (strongly positive)
Sales opportunity	SELL	MAKE (strongly positive)
No opportunity	Do nothing	MAKE (strongly positive)

A seguir apresentamos a especificação do requisito de consciência – com stop

Awareness[MarketTendency]		
Topic description:	The agent should be aware of market context, represented by market tendencies to decide which operation to perform in order to obtain future profit. The agent should also be aware of its current position (amount of resources) and the stop value .	
Goal:	Profit be achieved	
Awareness subtype:	Informational environment;	
Suggested operationalizations:	Use machine learning techniques to perceive tendencies	
Alternative actions:	Buy options; Sell options; Do not operate; Reset position	
Entity:	Market, Account	
Source of entity data:	Stock and change market system	
ContextDescription	{VALE5-DATA, VALEK48, VALEK50, Money}	
Domains of variable		
Variable name	Domain	
Market	{MarketData}	
VALEK48	Integer – quantity of VALEK48 in the account	
VALEK50	Integer – quantity of VALEK50 in the account	
VALE5-DATA attributes: Timestamp; Value (last minute); Volume (last minute); LowValue (last 15 minutes); HighValue (last 15 minutes);		
Context situations specification		
Situation name	Specification	
Buying opportunity	Operator perceives (is informed of) a high tendency	
Sales opportunity	Operator perceives (is informed of) a low tendency	
No opportunity	Operator perceives (is informed of) no tendency	
Stop achieved	Operator perceives that the stop value was achieved	
Alternative action choice		
Situation	Alternative action	Impact
Buying opportunity	BUY	MAKE (strongly positive)
Sales opportunity	SELL	MAKE (strongly positive)
No opportunity	Do nothing	MAKE (strongly positive)
Stop achieved	Reset position	MAKE (strongly positive)

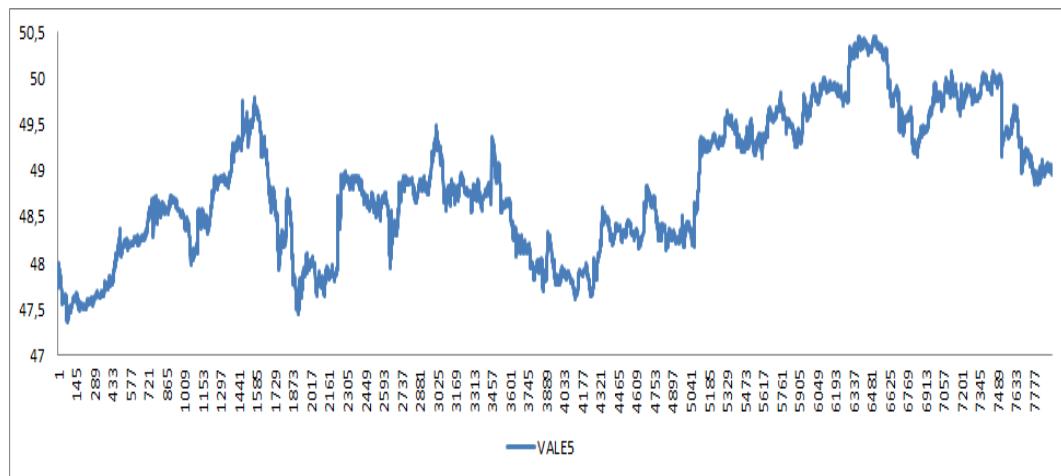


Figura 44 - Curva com a evolução de VALE5 durante simulação



Figura 45 - Curva com a evolução de VALEK48 e VALEK50 no período

Dados usados na Simulação

A simulação foi feita com dados das opções VALEK48 E VALEK50 de 15/10/2010 a 12/11/2010. A Figura 44 e Figura 45 apresentam as curvas com a evolução dos valores da ação VALE5 e das opções VALEK48 e VALEK50 no período, respectivamente. A Figura 46 apresenta uma pequena amostra dos dados.

Timestamp	VALE5 Value	Volume	LastMin	LastMax	VALEK48 Value	VALEK50 Value
15/10/10 10:06	47,8	193900	47,8	47,8	1,78	0,82
15/10/10 10:07	47,74	46600	47,74	47,8	1,74	0,79
15/10/10 10:08	47,74	23900	47,74	47,8	1,73	0,79
15/10/10 10:09	47,75	29400	47,74	47,8	1,75	0,82
15/10/10 10:10	47,76	32400	47,74	47,8	1,75	0,8
15/10/10 10:11	47,77	10500	47,74	47,8	1,74	0,8
15/10/10 10:12	47,85	33700	47,74	47,85	1,76	0,81
15/10/10 10:13	47,89	14500	47,74	47,89	1,79	0,8
15/10/10 10:14	47,91	34300	47,74	47,91	1,8	0,82
15/10/10 10:15	47,92	31600	47,74	47,92	1,81	0,82
15/10/10 10:16	48	82100	47,74	48	1,81	0,82
15/10/10 10:17	47,93	48300	47,74	48	1,8	0,81
15/10/10 10:18	47,91	29100	47,74	48	1,79	0,8
15/10/10 10:19	47,91	16100	47,74	48	1,78	0,79
15/10/10 10:20	47,92	13100	47,74	48	1,78	0,8
15/10/10 10:21	47,92	21600	47,74	48	1,78	0,79
15/10/10 10:22	47,86	53800	47,74	48	1,75	0,78
15/10/10 10:23	47,84	37500	47,75	48	1,76	0,76
15/10/10 10:24	47,83	39800	47,76	48	1,75	0,77
15/10/10 10:25	47,78	67600	47,77	48	1,71	0,76
15/10/10 10:26	47,8	69400	47,78	48	1,73	0,76
15/10/10 10:27	47,86	14700	47,78	48	1,74	0,78
15/10/10 10:28	47,82	9800	47,78	48	1,72	0,78
15/10/10 10:29	47,78	54500	47,78	48	1,7	0,77
15/10/10 10:30	47,75	69400	47,75	48	1,69	0,75

Figura 46 - Amostra dos dados usados na simulação

5.3.5.

Simulação de Operação

A seguir apresentamos o rastro das operações executadas na primeira hora no cenário com stop 2%. A tendência é representada pelo último valor de *Tendency*, podendo assumir os seguintes valores: Alta: 1; Sem tendência: 0; Baixa: -1.

5.3.6.

Resultados da Simulação

A Tabela 21 mostra um resumo das ações executadas automaticamente pelo sistema multiagente na primeira hora de operação durante a simulação.

Instante	Tendência prevista	Posição anterior	Motivo decisão	Ação executada	Nova posição
15/10/2010 10:00	-	-	-	-	Zerado
15/10/2010 10:12	Alta	Zerado	Previsão de alta	Compra	Comprado
15/10/2010 10:22	Neutra	Comprado	Stop Loss	Zerar Posição (Venda)	Zerado
15/10/2010 10:24	Baixa	Zerado	Previsão de baixa	Venda	Vendido
15/10/2010 10:41	Neutra	Vendido	Stop Gain	Zerar Posição (Compra)	Zerado
15/10/2010 10:42	Baixa	Zerado	Previsão de Baixa	Venda	Vendido
15/10/2010 10:52	Alta	Vendido	Previsão de Alta	Compra	Comprado
15/10/2010 10:55	Neutra	Comprado	Stop Loss	Zerar Posição (Venda)	Zerado
15/10/2010 10:57	Baixa	Zerado	Previsão de baixa	Venda	Vendido

Tabela 21 – Resumo das ações durante a primeira hora da simulação

Durante a operação, o software se adapta de acordo com a consciência que ele tem do mercado ao longo do tempo, decidindo e tomando as ações pertinentes. Inicialmente, o software se encontra na posição zerado (sem opções

em carteira). Ao perceber uma tendência de alta (no instante 15/10/2010 10:12) o software decide executar uma ação de compra mudando assim de posição para comprado. O software se mantém nessa posição até que no instante 15/10/2010 10:22 percebe que o “Stop Loss” (perda de 2%) foi atingido e decide desfazer a operação através de uma venda, retornando à posição zerado. No instante 15/10/2010 10:24 ao perceber uma tendência de baixa, o software toma a decisão de vender opções e passa a posição de vendido. No instante 15/10/2010 10:41 o software percebe que o “Stop Gain” (ganho de 2%) foi atingido e decide fechar a operação através de uma compra retornando à posição zerado. Em seguida, no instante 15/10/2010 10:42 o software percebe nova tendência de baixa e decide novamente realizar uma venda passando para a posição vendido. No instante 15/10/2010 10:52 o software percebe uma tendência de alta e executa uma ação de compra mudando sua posição de vendido para comprado. O software se mantém nessa posição até que no instante 15/10/2010 10:55 percebe que o “Stop Loss” (perda de 2%) foi atingido e desfaz a operação através de uma venda, retornando à posição zerado. No instante 15/10/2010 10:57 o software percebe nova tendência de baixa e decide realizar uma venda passando para a posição vendido. O rastro da execução dessa primeira hora da simulação encontra-se no Anexo II - Rastro da execução da primeira hora de simulação de investimento na bolsa.

A Figura 47 apresenta os resultados para ambos os cenários, com stop de 2% (*gain e loss*) e sem stop.

As curvas de evolução mostram os valores correntes a cada instante da simulação. O valor corrente calculado na simulação considera: a quantidade de dinheiro disponível (*Money*), a quantidade de dinheiro bloqueado (*Blocked*), o valor (quantidade vezes o preço no instante) das opções VALEK48 e VALEK50. É importante lembrar, que em uma posição “vendido” a quantidade de opções VALEK48 é negativa e o valor (quantidade vezes preço no instante) é subtraído.

Para assegurar que as operações simuladas estavam financeiramente corretas, foi aplicada após cada operação, tanto de compra quanto de venda, uma função de verificação financeira da operação onde o valor corrente imediatamente após a operação deveria ser igual ao valor corrente antes da operação menos as taxas de corretagem. Na simulação usamos a taxa de corretagem de R\$ 20 por operação.

Na simulação com stop de 2% (*gain e loss*), ao final do período foi atingido um valor de R\$ 56565,99 para um valor inicial de R\$ 30000,00, o que representa um aumento de 88,55%.

Na simulação sem stop, ao final do período foi atingido um valor de R\$ 26664,50 para um valor inicial de R\$ 30000,00, o que representa uma perda de 11,12%.

Embora o resultado com stop tenha se mostrado promissor, não representa o resultado real que o sistema multiagente teria obtido em uma operação real no período. As ordens de compra e venda simuladas pelo agente poderiam não ter sido executadas com o preço simulado por não haver volume de negócios suficiente para execução da ordem ou pelo tempo/instante em que ordem foi enviada (no mercado real, as ordens de compra e venda mais antigas têm preferência na execução quando há ordens com mesmo valor).

5.3.7. Considerações sobre os resultados da simulação

Do ponto de vista de validação da pesquisa, a modelagem e construção de um sistema multiagente para operação autônoma na bolsa de valores usando nossa abordagem para o requisito de consciência se mostrou adequada.

A complexidade maior dos casos de prognóstico recai na atividade/tarefa de interpretação dos dados. Nesses casos, o software não precisa apenas estar consciente do contexto no qual está atuando. O software precisa também estar consciente, ou ao menos inferir, como o contexto irá mudar. Para tanto, o uso de técnicas de aprendizado de máquina são recomendadas.

Como a certeza desta intepretação, decorrente dos modelos de aprendizado, não é absoluta pode levar a decisões equivocadas por parte do software. Essas decisões podem ser monitoradas, tornando o software autoconsciente de suas decisões e da satisfação ou não de suas metas em decorrência dessas decisões.

Caso perceba que uma decisão foi equivocada (não foi escolhida a melhor alternativa em função de interpretação equivocada da evolução do contexto) o software pode tomar ações para reduzir o impacto negativo dessas decisões.

A execução de uma ordem de stop /*loss* tem justamente esse objetivo: evitar um dano maior de uma decisão equivocada. Embora uma ordem do tipo *stop* possa ser colocada simultânea e paralelamente a uma operação de compra e venda em operações reais, na simulação optamos por deixar o sistema multiagente monitorar a evolução do contexto e enviasse uma ordem de compra ou venda simulando uma ordem *stop loss* caso percebesse a necessidade de reduzir danos de uma decisão passada equivocada.

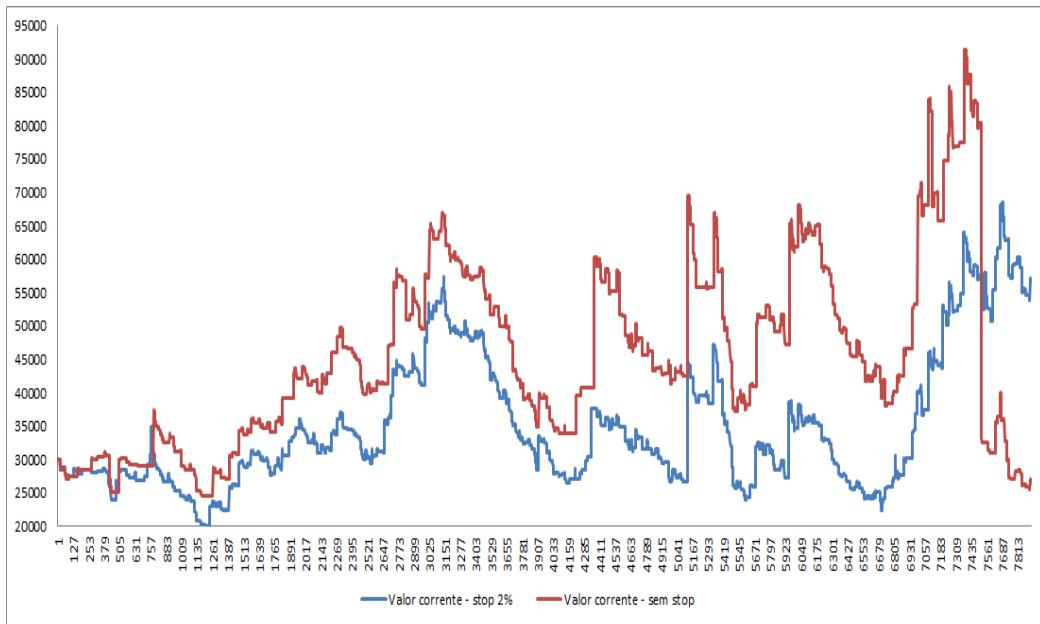


Figura 47 - Resultado da simulação sem e com uso de stop 2% (loss e gain)

5.3.8.

Considerações sobre a validação experimental

O objetivo da validação experimental foi demonstrar a viabilidade de nossa abordagem para o desenvolvimento de software consciente com base em requisitos.

O cerne de nossa proposta é a modelagem e especificação do requisito de consciência através de um padrão de desenho em i* (chamado de *SR Construct*) que ajuda o software a se tornar consciente do contexto em que está operando. Esta consciência é desenvolvida através da aquisição e interpretação de dados do contexto, sendo que a interpretação deve ser capaz de levar o software a tomar decisões e ações autônomas para satisfazer suas metas. Construímos nossa proposta usando uma modelagem intencional, especificamente em i*, e desenvolvemos um framework experimental para implementação em JADE.

Procuramos mostrar a validade da proposta em três casos distintos, com grau de dificuldade crescente do ponto de vista do requisito de consciência. Este grau de dificuldade advém da dificuldade em interpretar o que acontece e se adaptar, tomando decisões autônomas e agindo de acordo com esta interpretação. No caso da porta automática a interpretação é bastante simples. No caso dos elevadores, à medida que o número de elevadores aumenta a escolha da melhor decisão vai se tornando um pouco mais complexa. O caso de investimento na bolsa de valores, a consciência (percepção/interpretação) do

que acontece no mercado é mais complexa. Para tomar decisões, é preciso não apenas perceber o estado atual do mercado, mas prever os próximos estados futuros.

À parte as limitações impostas pelas escolhas que fizemos, que serão discutidas no Capítulo 6 – Conclusão, consideramos que nossa proposta apresenta uma abordagem útil para definição do requisito de consciência em um processo de engenharia de requisitos. Um ponto importante de nossa proposta é relacionar o contexto em que o software está inserido, com suas mudanças inerentes, ao mecanismo de decisão do software onde as alternativas são avaliadas em função da consciência do contexto que o software tem.

Resumo do capítulo

Nesse capítulo apresentamos três casos para validação experimental de nossa proposta de tratamento do requisito de consciência.

Os casos foram apresentados em uma ordem crescente de complexidade, iniciando com o sistema de controle automático de porta, passando pelo caso do controle de elevadores onde apresentamos três cenários (com um, dois e três elevadores) e finalizamos com o caso de operação autônoma na bolsa de valores, com a simulação de dois cenários de operação (com e sem *stop*).

Em cada caso, apresentamos o modelo i^* , definição do requisito de consciência, os dados usados na simulação e os resultados da simulação. A descrição em iStarML dos modelos i^* de cada cenário, juntamente com o rastro de criação dos agentes, encontra-se no Anexo I.

6 Conclusão

Neste capítulo listamos alguns trabalhos relacionados e discutimos as semelhanças e diferenças destes com o nosso trabalho. Discutimos as contribuições e limitações de nosso trabalho, bem como pontos a serem explorados em trabalhos futuros. Finalmente, apresentamos nossas considerações finais.

6.1. Trabalhos relacionados

“Ubiquitous Computing - Computing in Context” (Schmidt, 2002), tese de doutorado de Albrecht Schmidt na Lancaster University em 2002, é um trabalho diretamente relacionado com o nosso, no qual nos baseamos para definição e modelagem de contexto. Nele, Schmidt define a noção de contexto para software e a forma pela qual software pode ser consciente (*aware*) do contexto. O trabalho de Schmidt tem foco na computação ubíqua e a noção de contexto se aplica essencialmente aos aspectos que tratamos em consciência de contexto externo no catálogo de consciência de software. Neste âmbito, Schmidt propõe uma abordagem bastante sólida para apreensão destes tipos de contexto.

Em nosso trabalho, além de expandirmos a abrangência do requisito de consciência para outros tipos além de contexto externo, mais precisamente para consciência do autocomportamento e consciência do contexto social, apresentamos uma abordagem centrada na engenharia de requisitos. Nosso foco é a modelagem e análise do requisito de consciência, voltada para implementação em sistemas multiagente.

“Requirements-based Software System Adaptation” (Souza, 2012), tese de doutorado de Vitor Sousa na University of Trento em 2012, é um trabalho, assim como o nosso, centrado na engenharia de requisitos. Em sua tese, Victor apresenta um método que possibilita ao software, a partir do monitoramento dos requisitos do software descritos objetivamente em OCL (*Object Constraint Language* - linguagem declarativa para especificação de restrições em objetos de modelos UML), perceber a necessidade e disparar os mecanismos necessários à adaptação ou evolução do software.

A abordagem é baseada na aplicação de requisitos de evolução, requisitos que prescrevem a evolução desejada para outros requisitos, para modelar sistemas autoadaptativos cujos modelos evoluem de forma automática ou semiautomática em resposta a situações indesejadas. Nesta abordagem um conjunto de requisitos é monitorado e, quando necessário, a evolução do modelo é disparada e implementada de acordo as especificações dos requisitos de evolução - conjunto de primitivas operacionais a serem executadas nos elementos do modelo. Este processo de evolução é controlado por *feedback loops*.

Embora haja semelhança na abordagem (monitoração, análise e eventual atuação), inspirada no ciclo MAPE, o objeto de aplicação de nosso trabalho tem natureza diferente. Enquanto o trabalho de Vitor é focado na consciência dos requisitos, que constituem o alvo do monitoramento e evolução, nosso trabalho tem como objetivo melhorar o processo de decisão do software. Nesse sentido, a adaptação em nosso trabalho consiste em escolher a melhor alternativa para cada situação real prevista. Se por um lado, nosso trabalho tem um foco mais abrangente, visto que a consciência se aplica também ao contexto externo, e ao contexto social, por outro lado é mais restrito no sentido de não prevê mecanismos de adaptação ou evolução dos requisitos em si.

Os trabalhos “Development of Agent-Driven Systems: from i* Architectural Models to Intentional Agents’ Code” (Serrano e Leite, 2011) e “Dealing with softgoals at runtime: A fuzzy logic approach” (Serrano e Leite, 2011) de Mauricio Serrano e Julio Leite também apresentam algum grau de similaridade com nosso trabalho.

O primeiro trabalho apresenta uma abordagem para implementação de sistemas multiagente em JADEX – um add-on com arquitetura BDI (*Belief-Desire-Intention*) implementado na plataforma JADE, e propõe heurísticas transformacionais para mapear elementos de i* para abstrações BDI implementadas em JADEX. Neste trabalho optamos por usar iStarML como linguagem de descrição de modelos e implementamos diretamente no *framework* istarjade o mapeamento dos elementos de i* diretamente na plataforma JADE.

No segundo trabalho, “Dealing with softgoals at runtime: A fuzzy logic approach”, os autores propõem uma abordagem para lidar com as metas flexíveis em tempo de execução. A abordagem, baseada em uma análise de como os agentes humanos realmente analisam metas flexíveis na prática, combina regras de propagação (do impacto de escolhas de alternativas nas metas flexíveis) com ideias de lógica fuzzy em um mecanismo de raciocínio para

analisar as metas flexíveis e analisar os planos que satisfaçam essas metas em tempo de execução. Em nossa abordagem, as metas flexíveis que representam os requisitos não funcionais de consciência são resolvidas em tempo de execução para analisar a escolha de alternativas das metas principais dos agentes. Ou seja, enquanto no trabalho de Serrano e Leite, as alternativas são analisadas em função de seu impacto em metas flexíveis em tempo de desenho, nosso trabalho propõe a operacionalização da meta flexível de consciência para avaliar a contribuição das alternativas para as metas principais dos agentes em tempo de execução.

6.2. Contribuições do trabalho

Como principais contribuições podemos destacar:

- O catálogo para o requisito de consciência de software;
- A abordagem para modelagem e especificação do requisito de consciência de software;
- O mapeamento de modelos i* para plataforma JADE, implementada através do *framework* istarjade;
- Uma estratégia para operacionalização do requisito de consciência, aderente a abordagem de modelagem proposta, em sistemas multiagente intencionais.

A principal contribuição do catálogo de requisito é proporcionar a possibilidade de reuso de conhecimento do requisito de consciência de software, expresso através do refinamento desse requisito em subtipos de consciência de software e alternativas para operacionalização destes subtipos.

A abordagem de modelagem do requisito de consciência proposta tem com principal contribuição estabelecer uma ponte entre as funções de monitoramento e análise (da arquitetura MAPE) e os modelos intencionais de sistemas multiagente.

A contribuição principal do *framework* istarjade, além de possibilitar a implementação em JADE de agentes modelados em i*, é possibilitar a análise de requisitos em tempo de execução (*requirements at run-time*), fundamental para

análise de alternativas em cenários em que o processo de resolução dos agentes for mais complexo, como no caso da bolsa de valores.

A principal contribuição da estratégia de operacionalização apresentada é possibilitar a implementação de sistemas multiagente autônomos e autoadaptativos, com capacidade de operar em diferentes contextos.

6.3. Limitações do trabalho

O nosso trabalho apresenta algumas limitações práticas decorrentes das opções que fizemos ao longo do desenvolvimento do mesmo.

Do ponto de vista de implementação, nosso trabalho está limitado a sistemas multiagente na plataforma JADE. Para que esta limitação seja superada, é necessário mapear os elementos de i*, descritos em iStarML com as extensões propostas, para a plataforma alvo de forma semelhante a que foi feita para JADE. Isto pode ser feito para outras plataformas de sistemas multiagente, mas seria muito mais custoso (ou até mesmo inviável) fazê-lo para outro tipo de plataforma que não fosse de agentes.

O framework iStarJade impõe uma limitação na arquitetura do software. Esta limitação não existiria caso fosse usado um processo como Tropos (Bresciani et al., 2004), onde a arquitetura seria modelada com mais liberdade de escolhas.

Do ponto de vista de modelagem, nosso trabalho está limitado ao framework i*. A mesma abordagem pode ser desenvolvida para outras linguagens de modelagens. Podemos afirmar que é viável fazê-lo para linguagens que comportem as abstrações de agentes, metas e alternativas, ou seja, linguagens orientadas a meta e a agentes. Linguagens que não apresentem estes conceitos praticamente inviabilizam a adoção desta abordagem. Ainda assim, a estratégia geral proposta - decomposição do requisito de consciência nas etapas: aquisição e interpretação de dados e avaliação de alternativas – pode ser adotada. Nesse sentido, a estrutura canônica que representa essa estratégia de modelagem, que instanciamos em i* como um *SR Construct*, pode ser vista com um padrão de desenho (*design pattern*) que pode ser adaptado para outras linguagens de modelagem.

Do ponto de vista do conhecimento sobre o requisito de consciência o catálogo apresentado no capítulo 2 é, obviamente, limitado. O catálogo, embora acumule conhecimento que possa ser aprimorado, representa a visão atual do

que entendemos ser relevante para o tratamento do requisito de consciência de software. Nesse sentido, acreditamos que a evolução do catálogo é necessária.

Do ponto de vista da eficácia de nossa abordagem na solução de problemas reais, os limites residem na capacidade de aquisição de dados sobre um determinado contexto, e principalmente, na capacidade de interpretação desses dados. A eficácia é produto do acerto das escolhas de alternativas, e o acerto destas escolhas é, em última análise, produto da interpretação correta das situações de contexto. Ou seja: quanto melhor for a interpretação do contexto, melhor será a eficácia do software.

Em relação ao requisito de consciência de software em si, algumas limitações são inerentes aos processos de definição de requisitos e implementação de software. Um limite, aparentemente rígido, reside no fato que o software pode, no melhor dos casos, adquirir consciência para o escopo que foi desenhado/projetado. Não parece ser possível que software seja capaz de adquirir capacidades relacionadas aos requisitos de consciência, como perceber novas situações reais ou escolher alternativas, sem que isto tenha sido vislumbrado de alguma forma em tempo de desenho – de alguma forma teria de ser considerado no desenho do software a capacidade de reconhecer novas situações e decidir em face à essas novas situações. Assim, nos parece razoável crer que a consciência do software é limitada, em última análise, pela sua própria definição de requisitos.

6.4. Trabalhos futuros

A seguir, listamos alguns temas para trabalhos futuros que julgamos importantes para o enriquecimento desta pesquisa.

A evolução do catálogo do requisito de consciência de software é um candidato certo para um trabalho futuro pelos motivos e benefícios já discutidos anteriormente.

Outro trabalho futuro interessante é adoção de nossa estratégia de para modelagem, definição e implementação do requisito de consciência em outras linguagens de modelagem e implementação. Nesse sentido, KAOS seria um candidato natural para linguagem de modelagem intencional. Outra possibilidade seria investigar a viabilidade, e as possíveis restrições, de adotar nossa estratégia em UML.

Além da linguagem de modelagem, o framework pode ser desenvolvido em outras linguagens que permitam a implementação de sistemas multiagente. Para tanto, é importante que a linguagem permita programação distribuída e o gerenciamento de linhas de execução (threads) em nível do usuário, o que pode ser feito em Ruby (Flanagan e Matsumoto, 2008).

Outro trabalho futuro diz respeito ao aprimoramento dos mecanismos para aquisição, e principalmente, interpretação de dados de contexto. Como a função de interpretação é extremamente dependente do domínio do problema, uma possibilidade de trabalho futuro seria o desenvolvimento de um guia que auxiliasse na escolha das técnicas para interpretação, principalmente para os casos complexos que envolvam aprendizado de máquina.

Outro trabalho futuro interessante é o desenvolvimento de um interpretador baseado nos conceitos da semiótica, em que a percepção do contexto se daria com base na experiência prévia adquirida pelo interpretador.

Por fim, há o trabalho futuro de melhorar a implementação do sistema multiagente para operação autônoma da bolsa a fim de torná-lo operacional, e rentável.

6.5. Considerações finais

Ainda que tenhamos adotado uma definição de consciência para software bastante restrita, em comparação com o que entendemos por consciência para seres humanos, essa consciência básica, passível de ser entendida, modelada e implementada em software se mostrou útil na construção de sistemas multiagente autônomos que operem em contextos dinâmicos e complexos, como demonstraram os casos de validação experimental.

Por isso, consideramos que a pesquisa alcançou resultados satisfatórios para as questões que propusemos inicialmente: “Como tratar o requisito de consciência?”, “Como modelar este requisito?” e “Como guiar a sua implementação?”.

Mesmo assim, reconhecemos que há limites que nos parecem intransponíveis para a construção de software consciente. Um destes limites, já comentado na seção anterior, diz respeito à limitação da consciência do software a sua própria definição enquanto requisito desse software.

Outros limites ainda mais desafiadores têm a ver com as próprias noções de contexto e interpretação.

Em “Assinatura Acontecimento Contexto” (Derrida, 1971), o filósofo Jacques Derrida questiona:

“Mas serão os requisitos de um contexto impossíveis de determinar? É essa, no fundo, a questão mais geral que queria tentar elaborar. Existirá um conceito rigoroso e científico do contexto? Não abrigará a noção de contexto, por trás de certa confusão, pressuposições filosóficas muito determinadas?”.

Ainda no mesmo texto, Derrida afirma:

“Para o dizer desde já da maneira mais sumária, queria demonstrar porque é que um contexto nunca é absolutamente determinável, ou melhor, em que a sua determinação nunca é assegurada ou saturada.”

Segundo Derrida, esta não saturação estrutural teria como um dos efeitos “assinalar a insuficiência teórica do conceito corrente de contexto (linguístico ou não linguístico) tal como é recebido em numerosos domínios de investigação, com todos os conceitos aos quais é sistematicamente associado”.

Estes questionamentos me fazem refletir sobre os limites para o nosso entendimento, e o possível entendimento por software, do contexto. Nesse sentido, todo entendimento do contexto seria limitado, e por consequência, qualquer interpretação a partir desse entendimento pode ser equivocada.

Outro ponto sensível do processo de interpretação é atribuição de significado. Por mais que aprimoremos o processo de interpretação de um contexto percebido, o processo de interpretação precisa transpor esse contexto percebido para os modelos do software. Essa transposição, como todo processo de atribuição de significado, está sujeita a falhas.

Creio que esses limites, da delimitação de contexto e da atribuição de significado, representem desafios não apenas para consciência de software, mas para a própria filosofia.

Referências bibliográficas

Abowd, D., Dey, A. K., Orr, R., Brotherton, J. (1998). **Context-awareness in wearable and ubiquitous computing.** Virtual Reality, 3(3), 200-211.

Anton, A. I. **Goal-based requirements analysis.** (1996). In Requirements Engineering, Proceedings of the Second International Conference on (pp. 136-144). IEEE.

BASILI V.R. (1992) **Software Modeling and Measurement: The Goal Question Metric Paradigm.** Computer Science Technical Report Series, CS-TR-2956 (UMIACS-TR-92-96), University of Maryland, College Park, MD, September 1992.

Bellifemine, F., Poggi, A., Rimassa, G. **JADE – A FIPA-compliant agent framework.** (1999). Proceedings of PAAM'99, London, April 1999, pp.97-108.

Bellifemine, F., Caire, G., Poggi, A., Rimassa, G. **JADE : A White Paper.** (2003). In EXP magazine, Telecom Italia, Vol. 3, No. 3, September 2003.

Bellifemine, F. et al. **JADE PROGRAMMER'S GUIDE.** (2010). Disponível em: <http://jade.tilab.com/doc/programmersguide.pdf>. Acessado em 13/01/2014

Bennett, J., Lanning, S. **The netflix prize.** (2007). In Proceedings of KDD cup and workshop (Vol. 2007, p. 35).

Bergasa, L. M., Nuevo, J., Sotelo, M. A., Barea, R., Lopez, M. E. **Real-time system for monitoring driver vigilance.** (2006). Intelligent Transportation Systems, IEEE Transactions on, 7(1), 63-77.

Biometrics Market Intelligence. <http://www.biometricsmi.com>. Acessado em 30/01/2010.

Biometrics: Overview. (2007). Biometrics.cse.msu.edu. Acessado em 10/06/2012.

Bordini, R.H., Hübner, J.F., et al.. **Jason: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net.** (2005). Manual, version 0.6 edition (February 2005)

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J. **Tropos: An agent-oriented software development methodology.** (2004). Autonomous Agents and Multi-Agent Systems, 8(3), 203-236.

Cabri, G., Ferrari, L., Leonardi, L. **Agent role-based collaboration and coordination: a survey about existing approaches.** (2004). In Systems, Man and Cybernetics, 2004 IEEE International Conference on (Vol. 6, pp. 5473-5478). IEEE.

Cares, C., et al. **iStarML Reference's Guide.** (2007) Technical Report LSI-07-46-R, 2007.

Castelfranchi, C. **Formalizing the informal?: Dynamic social order, bottom-up social control, and spontaneous normative relations.** (2004) JAL, 1(1-2):47–92, 2004.

Chopra, A. K.; Dalpiaz, F.; Giorgini, P.; Mylopoulos, J. (2010). **Modeling and Reasoning about Service-Oriented Applications via Goals and Commitments.** In Advanced Information Systems Engineering. Lecture Notes in Computer Science Volume 6051, 2010, pp 113-128.

Chopra, A. K.; Singh, M.P. (2011). **Specifying and applying commitment-based business patterns.** In Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems, 2011.

Chung L., Nixon B. A., Yu E., and Mylopoulos J.. (2000). **Non-functional requirements in software engineering.** Kluwer Academic Publishers, 2000

Cunha, H. **Uso de estratégias orientadas a metas para modelagem de requisitos de segurança.** (2007). Dissertação (Mestrado em Informática). Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

Dalpiaz, F., Chopra, A. K., Giorgini, P., Mylopoulos, J. (2010). **Adaptation in open systems: Giving interaction its rightful place.** In Conceptual Modeling–ER 2010 (pp. 31-45). Springer Berlin Heidelberg.

Dalpiaz, F., Giorgini, P., Mylopoulos, J. (2009). **An architecture for requirements-driven self-reconfiguration.** In Advanced Information Systems Engineering (pp. 246-260). Springer Berlin Heidelberg.

Dalpiaz, F.; Giorgini, P.; Mylopoulos, J. (2009). **Software Self-Reconfiguration: a BDI based approach (Extended Abstract).** 8th International Conference on Autonomous Agents and Multiagent Systems AAMAS (2009).

Dastani, M., Grossi, D., Meyer, J.J.C., Tinnemeier, N. **Normative multi-agent programs and their logics.** (2009). In Meyer, J.J.C., Broersen, J., eds.: Knowledge Representation for Agents and Multi-Agent Systems, Berlin, Heidelberg, Springer-Verlag (2009) 16–31.

Derrida, J. **Assinatura Acontecimento Contexto.** (1971). Comunicação ao Congres International des Sociétés de philosophie de languefrançaise (Montreal, Agosto, 1971). O tema do colóquio era "A comunicação".

Dignum, V., Vázquez-Salceda, J., Dignum, F. **Omni: Introducing social structure, norms and ontologies into agent organizations.** (2005). In Programming Multi-Agent Systems (pp. 181-198). Springer Berlin Heidelberg.

Duboc, L; Letier, E.; Rosenblum, D. **Systematic Elaboration of Scalability Requirements through Goal-Obstacle Analysis.** (2012). IEEE Transactions on Software Engineering Journal – TSE. 2012.

Dybalova, D., Testerink, B., Dastani, M., Logan, B. (2013). **A Framework for Programming Norm-Aware Multi-Agent Systems.** In F. Dignum, & A. Chopra (Eds.), Proceedings of the 15th International Workshop on Coordination, Organisations, Institutions and Norms ({COIN} 2013).

Encyclopedia Britannica Academic Edition. (2013). Disponível em: www.britannica.com. Acessado em 01/07/2013 at 09:35 am.

Extensible Markup Language (XML). (2006). Version 1.1 (Second Edition). Disponível em: <http://www.w3.org/TR/xml11/#sec-xml11>

Fawcett, T., Provost, F. **Adaptive fraud detection.** (1997). Data mining and knowledge discovery, 1(3), 291-316.

FIPA website. Disponível em: <http://www.fipa.org>. Acessado em 13/01/2014

Flanagan, D., Matsumoto, Y. (2008). **The Ruby Programming Language.** O'Reilly Media. 2008. 442 pags.

Giovanni, C.. **JADE TUTORIAL - JADE PROGRAMMING FOR BEGINNERS.** (2009). Disponível em: <http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf>. Acessado em 13/01/2014.

Hinchey, M. G., Sterritt, R. (2006). **Self-managing software.** Computer, 39(2), 107-109.

howstuffworks. Disponível em:

<http://electronics.howstuffworks.com/gadgets/automotive/anti-sleep-alarm1.htm>

Acessado em: 05/08/2013.

Hübner, J., Sichman, J.. **Saci Programming Guide.** (2003). Version 0.9. Universidade de São Paulo.

IBM Autonomic Computing White Paper. **An architectural blueprint for autonomic computing.** (2005). Third Edition.

Information technology -- Automatic identification and data capture (AIDC) techniques -- Harmonized vocabulary -- Part 5: Locating systems. ISO/IEC 19762-5:2008.

Inverardi, P.; Mori, M. (2011). **Requirements models at run-time to support consistent system evolutions.** On Requirements@Run.Time (RE@RunTime), 2011 2nd International Workshop.

JADE (Java Agent DEvelopment Framework). Available at:
<http://jade.tilab.com/>

Jain, A. K., Kumar, A. **Biometrics of next generation: An overview.** (2010). Second Generation Biometrics.

Jason - a Java-based interpreter for an extended version of AgentSpeak.
Available at: <http://jason.sourceforge.net/wp/>

Jennings, N. R. **Agent-oriented software engineering.** (1999) In Multiple Approaches to Intelligent Systems (pp. 4-10). Springer Berlin Heidelberg.

Kendall, E. A. **Agent software engineering with role modelling.** (2001) In Agent-Oriented Software Engineering (pp. 163-169). Springer Berlin Heidelberg.

Korteum, G., Segall, Z., Bauer, M. (1998) **Context-Aware, Adaptive Wearable Computers as Remote Interfaces to “Intelligent” Environments.** 2nd International Symposium on Wearable Computers. 58-65.

Laney, D., Kart, L. **Emerging Role of the Data Scientist and the Art of Data Science.** (2012). G00227058 Gartner.

Lapouchnian, A. **Goal-oriented requirements engineering: An overview of the current research.** (2005). University of Toronto.

LEITE, J.C.S.P. **Engenharia de Requisitos - Notas de Aula.** (2006). Disponível em: <http://engenhariaderequisitos.blogspot.com/2006/08/processo.html>

Leonhardt, U. **Supporting Location-Awareness in Open Distributed Systems.** (1998). Phd Thesis, Department of Computing, Imperial College of Science, University of London. 1998.

Liu, L.; Yu, E. **Designing information systems in social context: a goal and scenario modelling approach.** (2004) Information Systems 29, 187–203.

Lorena, A., Carvalho, A. **Uma introdução às support vector machines.** (2007) Revista de Informática Teórica e Aplicada, 14(2), 43-67. 2007.

Marca, D. A., McGowan, C. L. (1987). **SADT: structured analysis and design technique.** McGraw-Hill, Inc..

McManis, C. **Take an in-depth look at the Java Reflection API.** (1997). JavaWorld.com,(Sep. 1997), 1-10.

Modgil, S., Faci, N., Meneguzzi, F., Oren, N., Miles, S., Luck, M. **A framework for monitoring agent-based normative systems.** (2009). In Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1 (pp. 153-160).

Oliveira, A.; Leite, J.C.S.P; Cysneiros, L.M. **Método ERi*c - Engenharia de Requisitos Intencional.** (2008). WER 2008.

Oliveira, A.; Leite, J.C.S.P; Cysneiros, L.M. **Using π Meta Modeling for Verifying π Models.** (2010). iStar 2010: 76-80.

Oliveira, A.; Leite, J.C.S.P; Cysneiros, L.M.; Lucena, C. **i* Diagnoses: A Quality Process for Building i* Models.** (2008). In PROCEEDINGS OF THE FORUM AT THE CAISE'08 CONFERENCE.

OWL 2 Web Ontology Language - Document Overview (Second Edition). (2012). Available at: <http://www.w3.org/TR/owl2-overview/>

Qureshi, N.A.; Liaskos, S.; Perini, A. (2011). **Reasoning about adaptive requirements for self-adaptive systems at runtime.** On Requirements@Run.Time (RE@RunTime), 2011 2nd International Workshop.

Resource Description Framework (RDF) - Model and Syntax Specification. (2004). Available at: <http://www.w3.org/TR/REC-rdf-syntax/>

SACI - Simple Agent Communication Infrastructure. Available at: <http://www.lti.pcs.usp.br/saci/index.shtml>

Schilit, B., Adams, N., Want, R. (1994). **Context-Aware Computing Applications.** 1st International Workshop on Mobile Computing Systems and Applications. 85-90

Schmidt, Albrecht. (2002). **Ubiquitous Computing - Computing in Context.** PhD dissertation, Lancaster University. 2002.

Schmidt, D. C. **Guest editor's introduction: Model-driven engineering.** (2006). Computer, 39(2), 0025-31.

Serrano, M.; Leite, J.C.S.P. (2011). **Capturing transparency-related requirements patterns through argumentation.** In: First International Workshop on Requirements Patterns (RePa), pp.32-41, 29 Aug. 2011.

Serrano, M.; Leite, J.C.S.P. **Dealing with softgoals at runtime: A fuzzy logic approach.** On Requirements@Run.Time (RE@RunTime), 2011 2nd International Workshop.

Serrano, M.; Leite, J.C.S.P. **Development of Agent-Driven Systems: from *i** Architectural Models to Intentional Agents Code.** (2011). iStar 2011: 55-60.

Simon H. A. (1996). **The Sciences of the Artificial.** 3rd Ed, MIT Press, Cambridge, MA, USA.

Souza, V. (2012). **Requirements-based Software System Adaptation.** PhD Dissertation. International Doctorate School in Information and Communication Technologies. DISI - University of Trento.

Souza, V.E.S., Mylopoulos, J. (2011). **From awareness requirements to adaptive systems: A control-theoretic approach.** On Requirements@Run.Time (RE@RunTime), 2011 2nd International Workshop.

Stanford Encyclopedia for Philosophy. (2013). Available at: <http://plato.stanford.edu/entries/consciousness/#2.2>. Acessado em 01/07/2013 at 09:51 am.

Supakkul, S., Hill, T., Chung, L., Tun, T. T., & Leite, J.C.S.P. **An NFR pattern approach to dealing with NFRs.** (2010). In Requirements Engineering Conference (RE), 2010 18th IEEE International(pp. 179-188). IEEE.

The Free Dictionary: Dictionary, Encyclopedia and Thesaurus. (2013). Available at: <http://www.thefreedictionary.com>. Acessado em 01/07/2013 at 10:40 am.

The global positioning system: a shared national asset: recommendations for technical improvements and enhancements. (1995). National Research

Council (U.S.). Committee on the Future of the Global Positioning System; National Academy of Public Administration. National Academies Press. p. 16. ISBN 0-309-05283-1., Chapter 1, p. 16.

Trail: The Reflection API - The Java Tutorials. Disponível em: <http://docs.oracle.com/javase/tutorial/reflect/>. Acessado em 14/01/2014

Truong, H. L., & Dustdar, S. (2009). **A survey on context-aware web service systems.** International Journal of Web Information Systems, 5(1), 5-31

Van Lamsweerde, A. **Requirements engineering in the year 00: a research perspective.** (2000). In: The Proceedings of the 22nd International Conference on Software Engineering, Limerick, June 2000, ACM press, New York.

Van Lamsweerde, A. (2001). **Goal-oriented requirements engineering: A guided tour.** In Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on (pp. 249-262). IEEE.

Vapnik, V. **The nature of Statistical learning theory.** (1995). Springer-Verlag, New York, 1995.

Von Bertalanffy, L. (1968). **General system theory: Foundations, development, applications.** (Revised Ed., p. 289). New York: George Braziller.

Ward, A., Jones, A., Hopper, A. (1997). **A New Location Technique for the Active Office.** IEEE Personal Communications 4(5) 42-47

Weiser, M. (1993). **Some computer science issues in ubiquitous computing.** Communications of the ACM, 36(7), 75-84. 1993.

Wikipedia. Available at: <http://www.wikipedia.org/>. (2013). Acessado em 01/07/2013 as 11:25 am.

WordNet - A lexical database for English. (2013). Available at: <http://wordnet.princeton.edu/>. Acessado em 01/07/2013 at 10:05 am.

Wooldridge, M.; Jennings, N. R. **Intelligent agents: theory and practice.** (1995) The Knowledge Engineering Review 10 (2) 115-152.

y López, F. L., Luck, M. (2004). **A model of normative multi-agent systems and dynamic relationships.** In Regulated agent-based social systems (pp. 259-280). Springer Berlin Heidelberg.

Yu, E. **Modelling Strategic Relationships for Process Reengineering.** (1995) Ph.D. thesis, also Tech. Report DKBS-TR-94-6, Dept. of Computer Science, University of Toronto, 1995.

Yu, E.; Mylopoulos, J. **Why goal-oriented requirements engineering.** (1998). In Proceedings of the Fourth International Workshop on Requirements Engineering: Foundations of Software Quality, Pisa, Italy, Presses Universitaires de Namur, Paris, June 1998, pp. 15–22.

Zambonelli, F., Jennings, N., Wooldridge, M. **Organizational Rules as an Abstraction for the Analysis and Design of Multi-agent Systems.** (2001) Journal of Knowledge and Software Engineering. Vol. 11, No. 3, 2001.

Anexo I- Informações complementares da validação experimental

Caso 2 – Sistema de controle de elevadores – cenário com um elevador

Descrição em iStarML do modelo i* para sistema multiagente de controle de elevador com um elevador

```
<istarml version="2.0">
<diagram>
    <actor type="role" id="1" name="DemandsManager">
        <boundary>
            <ielement id="1.1" type="task" name="SendToElevator1"
                basic="true"/>
            <ielement id="1.10" type="goal" name="ElavatorBeSelected" >
                <ielementLink type="means-end" run-mode="unique">
                    <ielement iref="1.1"/>
                </ielementLink>
            </ielement>
            <ielement id="1.11" type="task" name="UsersDataAquisition"
                basic="true"/>
            <ielement id="1.12" type="task" name="ElevatorsDataAquisition"
                basic="true"/>
            <ielement id="1.13" type="task"
                name="UsersAndElevatorsDataAquisition">
                <ielementLink type="decomposition" run-mode="sequential" >
                    <ielement iref="1.11"/>
                    <ielement iref="1.12"/>
                </ielementLink>
            </ielement>
            <ielement id="1.14" type="task"
                name="UsersAndElevatorsDataInterpretation" basic="true"/>
            <ielement id="1.15" type="context-awareness"
                name="UsersAndElevatorsContext">
                <ielementLink type="decomposition" run-mode="sequential">
                    <ielement iref="1.13"/>
                    <ielement iref="1.14"/>
                </ielementLink>
                <ielementLink type="argumentation">
                    <ielement iref="1.1" situation-
                        name="elevator1IsTheOnlyOption"/>
                </ielementLink>
            </ielement>
            <ielement id="1.16" type="softgoal"
                name="Awareness[UsersAndElevators]">
                <ielementLink type="means-end" run-mode="unique">
                    <ielement iref="1.15"/>
                </ielementLink>
```

```

</ielement>
<ielement id="1.20" type="task" name="ManageUsersDemands" >
    <ielementLink type="decomposition" run-mode="sequential" >
        <ielement iref="1.16"/>
        <ielement iref="1.10"/>
    </ielementLink>
</ielement>
<ielement id="1.21" type="goal"
name="DemandsManagedAutomatically"
main="true">
    <ielementLink type="means-end" run-mode="unique">
        <ielement iref="1.20"/>
    </ielementLink>
</ielement>
</boundary>
</actor>
<actor type="role" id="2" name="Elevator">
    <boundary>
        <ielement id="2.1" type="task" name="InformPositionAndUsers"
basic="true"/>
        <ielement id="2.2" type="goal" name="ManagerBeInformed"
main="true">
            <ielementLink type="means-end" run-mode="unique">
                <ielement iref="2.1"/>
            </ielementLink>
        </ielement>
        <ielement id="2.3" type="task" name="TransportUsers" basic="true"/>
        <ielement id="2.4" type="goal" name="UsersBeTransported"
main="true">
            <ielementLink type="means-end" run-mode="unique">
                <ielement iref="2.3"/>
            </ielementLink>
        </ielement>
    </boundary>
</actor>
<ielement id="3" type="resource" name="ElevatorPosition"
waiting_time="3500">
    <dependency>
        <depender aref="1" /> <!-- "DemandsManager" -->
        <dependee iref="2.2" /> <!-- "InformPositionAndUsers" -->
    </dependency>
</ielement>
<ielement id="4" type="task" name="TransportUsers"
waiting_time="3500">
    <dependency>
        <depender aref="1" /> <!-- "DemandsManager" -->
        <dependee iref="2.4"/> <!-- "UsersBeTransported" -->
    </dependency>

```

```

</ielement>
<actor id="5" type="agent" name="manager">
    <actorLink type="plays" aref="1"/>
</actor>
<actor id="6" type="agent" name="Elevator1">
    <actorLink type="plays" aref="2"/>
</actor>
</diagram>
</istarml>

```

Segue o rastro de criação dos agentes em *istarjade*:

```

Agent manager@10.26.144.42:1099/JADE has internal element
name=DemandsManagedAutomatically
Agent manager@10.26.144.42:1099/JADE internal element type=class
mainelement.DemandsManagedAutomatically
name=DemandsManagedAutomatically has the mean type=class istar.impl.Task
name=ManageUsersDemands
Agent manager@10.26.144.42:1099/JADE has internal element
name=ManageUsersDemands
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=ManageUsersDemands has the subelement type=class
istar.impl.Belief name=UsersAndElevatorsContext
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=ManageUsersDemands has the subelement type=class
istar.impl.Goal name=ElavatorBeSelected
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsContext
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsDataAquisition
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=UsersAndElevatorsDataAquisition has the subelement
type=class basicelement.UsersDataAquisition name=UsersDataAquisition
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=UsersAndElevatorsDataAquisition has the subelement
type=class basicelement.ElevatorsDataAquisition name=FireDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=FireDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsDataInterpretation
Agent manager@10.26.144.42:1099/JADE has internal element
name=ElavatorBeSelected
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Goal name=ElavatorBeSelected has the mean type=class
basicelement.MandaParaElevator1 name=MandaParaElevator1
Agent manager@10.26.144.42:1099/JADE has internal element
name=MandaParaElevator1
-----
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=ManagerBelInformed

```

Agent Elevator1@10.26.144.42:1099/JADE internal element type=class
mainelement.ManagerBeInformed name=ManagerBeInformed has the mean
type=class basicelement.InformPositionAndUsers
name=InformPositionAndUsers

Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=InformPositionAndUsers

Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=UsersBeTransported

Agent Elevator1@10.26.144.42:1099/JADE internal element type=class
mainelement.UsersBeTransported name=UsersBeTransported has the mean
type=class basicelement.TransportUsers name=TransportUsers

Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=TransportUsers

manager is registering as listener to the service:

istar.impl.Resource:ElevatorPosition

manager is registering as listener to the service: istar.impl.Task:TransportUsers

Elevator1 is registering as provider to the service: service

type=istar.impl.Resource:ElevatorPosition

Elevator1 is registering as provider to the service: istar.impl.Task:TransportUsers

Caso 2 – Sistema de controle de elevadores – cenário com dois elevadores

Descrição em iStarML do modelo i* com dois elevadores.

```
<istarml version="2.0">
<diagram>
<actor type="role" id="1" name="DemandsManager">
<boundary>
<ielement id="1.1" type="task" name="SendToElevator1"
basic="true"/>
<ielement id="1.2" type="task" name="SendToElevator2"
basic="true"/>
<ielement id="1.10" type="goal" name="ElavatorBeSelected" >
<ielementLink type="means-end" run-mode="unique">
<ielement iref="1.1"/>
<ielement iref="1.2"/>
</ielementLink>
</ielement>
<ielement id="1.11" type="task" name="UsersDataAquisition"
basic="true"/>
<ielement id="1.12" type="task" name="ElevatorsDataAquisition"
basic="true"/>
<ielement id="1.13" type="task"
name="UsersAndElevatorsDataAquisition">
<ielementLink type="decomposition" run-mode="sequential" >
<ielement iref="1.11"/>
<ielement iref="1.12"/>
</ielementLink>
</ielement>
<ielement id="1.14" type="task"
name="UsersAndElevatorsDataInterpretation" basic="true"/>
<ielement id="1.15" type="context-awareness"
name="UsersAndElevatorsContext">
<ielementLink type="decomposition" run-mode="sequential">
<ielement iref="1.13"/>
<ielement iref="1.14"/>
</ielementLink>
<ielementLink type="argumentation">
<ielement iref="1.1" situation-
name="elevator1IsTheBestOption"/>
<ielement iref="1.2" situation-
name="elevator2IsTheBestOption"/>
</ielementLink>
</ielement>
<ielement id="1.16" type="softgoal"
name="Awareness[UsersAndElevators]" >
<ielementLink type="means-end" run-mode="unique">
<ielement iref="1.15"/>
</ielementLink>
</ielement>
<ielement id="1.20" type="task" name="ManageUsersDemands" >
<ielementLink type="decomposition" run-mode="sequential" >
<ielement iref="1.16"/>
<ielement iref="1.10"/>
```

```

        </ielementLink>
    </ielement>
    <ielement id="1.21" type="goal"
        name="DemandsManagedAutomatically"
        main="true">
        <ielementLink type="means-end" run-mode="unique">
            <ielement iref="1.20"/>
        </ielementLink>
    </ielement>
</boundary>
</actor>
<actor type="role" id="2" name="Elevator">
    <boundary>
        <ielement id="2.1" type="task" name="InformPositionAndUsers"
            basic="true" />
        <ielement id="2.2" type="goal" name="ManagerBeInformed"
            main="true">
            <ielementLink type="means-end" run-mode="unique">
                <ielement iref="2.1"/>
            </ielementLink>
        </ielement>
        <ielement id="2.3" type="task" name="TransportUsers"
            basic="true" />
        <ielement id="2.4" type="goal" name="UsersBeTransported"
            main="true">
            <ielementLink type="means-end" run-mode="unique">
                <ielement iref="2.3"/>
            </ielementLink>
        </ielement>
    </boundary>
</actor>
<ielement id="3" type="resource" name="ElevatorPosition">
    <dependency>
        <depender aref="1" /> <!-- "DemandsManager" -->
        <dependee iref="2.2" /> <!-- "InformPositionAndUsers" -->
    </dependency>
</ielement>
<ielement id="4" type="task" name="TransportUsers">
    <dependency>
        <depender iref="1.1" /> <!-- "SendToElevator1" -->
        <dependee aref="11"/> <!-- agent "Elevator1" -->
    </dependency>
</ielement>
<ielement id="5" type="task" name="TransportUsers">
    <dependency>
        <depender iref="1.2" /> <!-- "SendToElevator2" -->
        <dependee aref="12"/> <!-- agent "Elevator2" -->
    </dependency>
</ielement>
<actor id="5" type="agent" name="manager">
    <actorLink type="plays" aref="1"/>
</actor>
<actor id="11" type="agent" name="Elevator1">
    <actorLink type="plays" aref="2"/>
</actor>

```

```

<actor id="12" type="agent" name="Elevator2">
    <actorLink type="plays" aref="2"/>
</actor>
</diagram>
</istarml>
```

Segue o rastro de criação dos agentes em *istarjade*:

```

Agent manager@10.26.144.42:1099/JADE has internal element
name=DemandManagedAutomatically
Agent manager@10.26.144.42:1099/JADE internal element type=class
mainelement.DemandManagedAutomatically
name=DemandManagedAutomatically has the mean type=class istar.impl.Task
name=ManageUsersDemands
Agent manager@10.26.144.42:1099/JADE has internal element
name=ManageUsersDemands
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=ManageUsersDemands has the subelement type=class
istar.impl.Belief name=UsersAndElevatorsContext
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=ManageUsersDemands has the subelement type=class
istar.impl.Goal name=ElavatorBeSelected
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsContext
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsDataAquisition
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=UsersAndElevatorsDataAquisition has the subelement
type=class basicelement.UsersDataAquisition name=UsersDataAquisition
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=UsersAndElevatorsDataAquisition has the subelement
type=class basicelement.ElevatorsDataAquisition name=FireDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=FireDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsDataInterpretation
Agent manager@10.26.144.42:1099/JADE has internal element
name=ElavatorBeSelected
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Goal name=ElavatorBeSelected has the mean type=class
basicelement.MandaParaElevator1 name=MandaParaElevator1
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Goal name=ElavatorBeSelected has the mean type=class
basicelement.SendToElevator2 name=SendToElevator2
Agent manager@10.26.144.42:1099/JADE has internal element
name=MandaParaElevator1
Agent manager@10.26.144.42:1099/JADE has internal element
name=SendToElevator2
-----
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=ManagerBelInformed
Agent Elevator1@10.26.144.42:1099/JADE internal element type=class
```

mainelement.ManagerBeInformed name=ManagerBeInformed has the mean
type=class
basicelement.InformPositionAndUsers name=InformPositionAndUsers
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=InformPositionAndUsers
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=UsersBeTransported
Agent Elevator1@10.26.144.42:1099/JADE internal element type=class
mainelement.UsersBeTransported name=UsersBeTransported has the mean
type=class
basicelement.TransportUsers name=TransportUsersAgent
Elevator1@ 10.26.144.42:1099/JADE has internal element name=TransportUsers

Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=ManagerBeInformed
Agent Elevator2@10.26.144.42:1099/JADE internal element type=class
mainelement.ManagerBeInformed name=ManagerBeInformed has the mean
type=class basicelement.InformPositionAndUsers
name=InformPositionAndUsers
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=InformPositionAndUsers
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=UsersBeTransported
Agent Elevator2@10.26.144.42:1099/JADE internal element type=class
mainelement.UsersBeTransported name=UsersBeTransported has the mean
type=class basicelement.TransportUsers name=TransportUsers
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=TransportUsers

manager is registering as listener to the service:
istar.impl.Resource:ElevatorPosition
manager is registering as listener to the service: istar.impl.Task:TransportUsers
Elevator1 is registering as provider to the service: service
type=istar.impl.Resource:ElevatorPosition
Elevator1 is registering as provider to the service: istar.impl.Task:TransportUsers
Elevator2 is registering as provider to the service: service
type=istar.impl.Resource:ElevatorPosition
Elevator2 is registering as provider to the service: istar.impl.Task:TransportUsers

Caso 2 – Sistema de controle de elevadores – cenário com três elevadores

Descrição em iStarML do modelo i* com três elevadores.

```
<istarml version="2.0">
  <diagram>
    <actor type="role" id="1" name="DemandsManager">
      <boundary>
        <ielement id="1.1" type="task" name="SendToElevator1"
        basic="true"/>
        <ielement id="1.2" type="task" name="SendToElevator2"
        basic="true"/>
        <ielement id="1.3" type="task" name="SendToElevator3"
        basic="true"/>
        <ielement id="1.10" type="goal" name="ElavatorBeSelected" >
          <ielementLink type="means-end" run-mode="unique">
            <ielement iref="1.1"/>
            <ielement iref="1.2"/>
            <ielement iref="1.3"/>
          </ielementLink>
        </ielement>
        <ielement id="1.11" type="task" name="UsersDataAquisition"
        basic="true"/>
        <ielement id="1.12" type="task" name="ElevatorsDataAquisition"
        basic="true"/>
        <ielement id="1.13" type="task"
        name="UsersAndElevatorsDataAquisition">
          <ielementLink type="decomposition" run-mode="sequential" >
            <ielement iref="1.11"/>
            <ielement iref="1.12"/>
          </ielementLink>
        </ielement>
        <ielement id="1.14" type="task"
        name="UsersAndElevatorsDataInterpretation" basic="true"/>
        <ielement id="1.15" type="context"
        name="UsersAndElevatorsContext">
          <ielementLink type="decomposition" run-mode="sequential">
            <ielement iref="1.13"/>
            <ielement iref="1.14"/>
          </ielementLink>
          <ielementLink type="argumentation">
            <ielement iref="1.1" situation-
            name="elevator1IsTheBestOption"/>
            <ielement iref="1.2" situation-
            name="elevator2IsTheBestOption"/>
            <ielement iref="1.3" situation-
            name="elevator3IsTheBestOption"/>
          </ielementLink>
        </ielement>
        <ielement id="1.16" type="softgoal"
        name="Awareness[UsersAndElevators]" >
          <ielementLink type="means-end" run-mode="unique">
            <ielement iref="1.15"/>

```

```

</ielementLink>
</ielement>
<ielement id="1.20" type="task" name="ManageUsersDemands" >
    <ielementLink type="decomposition" run-mode="sequential" >
        <ielement iref="1.16"/>
        <ielement iref="1.10"/>
    </ielementLink>
</ielement>
<ielement id="1.21" type="goal"
    name="DemandsManagedAutomatically"
    main="true">
    <ielementLink type="means-end" run-mode="unique">
        <ielement iref="1.20"/>
    </ielementLink>
</ielement>
</boundary>
</actor>
<actor type="role" id="2" name="Elevator">
    <boundary>
        <ielement id="2.1" type="task" name="InformPositionAndUsers"
            basic="true" />
        <ielement id="2.2" type="goal" name="ManagerBeInformed"
            main="true">
            <ielementLink type="means-end" run-mode="unique">
                <ielement iref="2.1"/>
            </ielementLink>
        </ielement>
        </elements>
        <ielement id="2.3" type="task" name="TransportUsers"
            basic="true" />
        <ielement id="2.4" type="goal" name="UsersBeTransported"
            main="true">
            <ielementLink type="means-end" run-mode="unique">
                <ielement iref="2.3"/>
            </ielementLink>
        </ielement>
    </boundary>
</actor>
<ielement id="3" type="resource" name="ElevatorPosition">
    <dependency>
        <depender aref="1" /> <!-- "DemandsManager" -->
        <dependee iref="2.2" /> <!-- "InformPositionAndUsers" -->
    </dependency>
</ielement>
<ielement id="4" type="task" name="TransportUsers">
    <dependency>
        <depender iref="1.1" /> <!-- "SendToElevator1" -->
        <dependee aref="11"/> <!-- agent "Elevator1" -->
    </dependency>
</ielement>
<ielement id="5" type="task" name="TransportUsers">
    <dependency>
        <depender iref="1.2" /> <!-- "SendToElevator2" -->
        <dependee aref="12"/> <!-- agent "Elevator2" -->
    </dependency>

```

```

</ielement>
<ielement id="6" type="task" name="TransportUsers">
    <dependency>
        <dependee iref="1.3" /> <!-- "SendToElevator2" -->
        <dependee aref="13"/> <!-- agent "Elevator2" -->
    </dependency>
</ielement>
<actor id="5" type="agent" name="manager">
    <actorLink type="plays" aref="1"/>
</actor>
<actor id="11" type="agent" name="Elevator1">
    <actorLink type="plays" aref="2"/>
</actor>
<actor id="12" type="agent" name="Elevator2">
    <actorLink type="plays" aref="2"/>
</actor>
<actor id="13" type="agent" name="Elevator3">
    <actorLink type="plays" aref="2"/>
</actor>
</diagram>
</istarml>

```

Segue o rastro de criação dos agentes em istarjade:

```

Agent manager@10.26.144.42:1099/JADE has internal element
name=DemandManagedAutomatically
Agent manager@10.26.144.42:1099/JADE internal element type=class
mainElement.DemandManagedAutomatically
name=DemandManagedAutomatically has the mean type=class istar.impl.Task
name=ManageUsersDemands
Agent manager@10.26.144.42:1099/JADE has internal element
name=ManageUsersDemands
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=ManageUsersDemands has the subelement type=class
istar.impl.Belief name=UsersAndElevatorsContext
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=ManageUsersDemands has the subelement type=class
istar.impl.Goal name=ElavatorBeSelected
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsContext
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsDataAquisition
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=UsersAndElevatorsDataAquisition has the subelement
type=class basicElement.UsersDataAquisition name=UsersDataAquisition
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Task name=UsersAndElevatorsDataAquisition has the subelement
type=class basicElement.ElevatorsDataAquisition name=FireDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=FireDataAquisition
Agent manager@10.26.144.42:1099/JADE has internal element
name=UsersAndElevatorsDataInterpretation

```

```

Agent manager@10.26.144.42:1099/JADE has internal element
name=ElavatorBeSelected
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Goal name=ElavatorBeSelected has the mean type=class
basicelement.MandaParaElevator1 name=SendToElevator1
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Goal name=ElavatorBeSelected has the mean type=class
basicelement.SendToElevator2 name=SendToElevator2
Agent manager@10.26.144.42:1099/JADE internal element type=class
istar.impl.Goal name=ElavatorBeSelected has the mean type=class
basicelement.ToElevator3 name=ToElevator3
Agent manager@10.26.144.42:1099/JADE has internal element
name=SendToElevator1
Agent manager@10.26.144.42:1099/JADE has internal element
name=SendToElevator2
Agent manager@10.26.144.42:1099/JADE has internal element
name=ToElevator3
-----
[carregador] startUpBehaviour is executing
[carregador] startUpBehaviour is finishing
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=ManagerBeInformed
Agent Elevator1@10.26.144.42:1099/JADE internal element type=class
mainelement.ManagerBeInformed name=ManagerBeInformed has the mean
type=class basicelement.InformPositionAndUsers
name=InformPositionAndUsers
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=InformPositionAndUsers
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=UsersBeTransported
Agent Elevator1@10.26.144.42:1099/JADE internal element type=class
mainelement.UsersBeTransported name=UsersBeTransported has the mean
type=class basicelement.TransportUsers name=TransportUsers
Agent Elevator1@10.26.144.42:1099/JADE has internal element
name=TransportUsers
-----
Agent Elevator3@10.26.144.42:1099/JADE has internal element
name=ManagerBeInformed
Agent Elevator3@10.26.144.42:1099/JADE internal element type=class
mainelement.ManagerBeInformed name=ManagerBeInformed has the mean
type=class basicelement.InformPositionAndUsers
name=InformPositionAndUsers
Agent Elevator3@10.26.144.42:1099/JADE has internal element
name=InformPositionAndUsers
Agent Elevator3@10.26.144.42:1099/JADE has internal element
name=UsersBeTransported
Agent Elevator3@10.26.144.42:1099/JADE internal element type=class
mainelement.UsersBeTransported name=UsersBeTransported has the mean
type=class basicelement.TransportUsers name=TransportUsersAgent
Elevator3@10.26.144.42:1099/JADE has internal element name=TransportUsers
-----
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=ManagerBeInformed
Agent Elevator2@10.26.144.42:1099/JADE internal element type=class
mainelement.ManagerBeInformed name=ManagerBeInformed has the mean

```

```
type=class                                basicelement.InformPositionAndUsers
name=InformPositionAndUsers
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=InformPositionAndUsers
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=UsersBeTransported
Agent Elevator2@10.26.144.42:1099/JADE internal element type=class
mainelement.UsersBeTransported name=UsersBeTransported has the mean
type=class basicelement.TransportUsers name=TransportUsers
Agent Elevator2@10.26.144.42:1099/JADE has internal element
name=TransportUsers
-----
-----
manager is registering as listener to the service:
istar.impl.Resource:ElevatorPosition
manager is registering as listener to the service: istar.impl.Task:TransportUsers
Elevator1 is registering as provider to the service: service
type=istar.impl.Resource:ElevatorPosition
Elevator1 is registering as provider to the service: istar.impl.Task:TransportUsers
Elevator2 is registering as provider to the service: service
type=istar.impl.Resource:ElevatorPosition Elevator2 is registering as provider to
the service: istar.impl.Task:TransportUsers
Elevator3 is registering as provider to the service: service
type=istar.impl.Resource:ElevatorPosition Elevator3 is registering as provider to
the service: istar.impl.Task:TransportUsers
```

Caso 3 – Sistema multiagente para operação intraday no mercado de ações da bolsa de valores

Descrição em iStarML do modelo i* para o sistema multiagente para operação autônoma na bolsa de valores.

```
<?xml version="1.0"?>
<iistarml version="1.0">
<diagram name="market">
    <actor type="role" id="1" name="operator">
        <boundary>
            <ielement id="0" type="task" name="DoNothing" basic="true" />
            <ielement id="1.0" type="task" name="PutOrder">
                <ielementLink type="decomposition" run-mode="sequential">
                    <ielement iref="6"/>
                </ielementLink>
            </ielement>
            <ielement id="1.0.1" type="task" name="Register" basic="true" />
            <ielement id="1.1.1" type="resource" name="money" basic="true"/>
            <ielement id="1.1.2" type="task" name="AssemblyBuyOrder" basic="true"/>
            <ielement id="1.1" type="task" name="Buy">
                <ielementLink type="decomposition" run-mode="sequential">
                    <ielement iref="1.1.2"/>
                    <ielement iref="1.0"/>
                    <ielement iref="1.0.1"/>
                </ielementLink>
            </ielement>
            <ielement id="1.2.1" type="resource" name="StockOption" basic="true"/>
            <ielement id="1.2.2" type="task" name="AssemblySellOrder" basic="true"/>
            <ielement id="1.2" type="task" name="Sell">
                <ielementLink type="decomposition" run-mode="sequential">
                    <ielement iref="1.2.2"/>
                    <ielement iref="1.0"/>
                    <ielement iref="1.0.1"/>
                </ielementLink>
            </ielement>
            <ielement id="1.3.1" type="task" name="ResetPosition" basic="true"/>
            <ielement id="1.3" type="task" name="ResetPosition">
                <ielementLink type="decomposition" run-mode="sequential">
                    <ielement iref="1.3.1"/>
                    <ielement iref="1.0"/>
                    <ielement iref="1.0.1"/>
                </ielementLink>
            </ielement>
        </boundary>
    </actor>
</diagram>
```

```
<ielement id="1.4" type="task" name="ObtainTendency">
    <ielementLink type="decomposition" run-mode="sequential">
        <ielement iref="5"/>
    </ielementLink>
</ielement>
<ielement id="1.5" type="task" name="Ponder" basic="true"/>
<ielement id="1.6" type="goal" name="OperationBeChosen" >
    <ielementLink id="1.6.l" type="means-end" run-mode="unique">
        <ielement iref="1.1"/>
        <ielement iref="1.2"/>
        <ielement iref="1.3"/>
        <ielement iref="0"/>
    </ielementLink>
</ielement>
<ielement id="1.7" type="task" name="DataInterpretation">
    <ielementLink type="decomposition" run-mode="sequential">
        <ielement iref="1.4"/>
        <ielement iref="1.5"/>
    </ielementLink>
</ielement>
<ielement id="1.8" type="context-awareness"
    name="UserAndMarketContext">
    <ielementLink type="decomposition" run-mode="sequential">
        <ielement iref="1.1.1"/>
        <ielement iref="1.2.1"/>
        <ielement iref="1.7"/>
    </ielementLink>
    <ielementLink type="argumentation" >
        <ielement iref="1.6"/>
    </ielementLink>
</ielement>
<ielement id="1.9" type="softgoal"
name="Awareness[MarketTendency]">
    <ielementLink type="means-end" run-mode="unique">
        <ielement iref="1.8"/>
    </ielementLink>
</ielement>
<ielement id="1.10" type="task" name="operate" >
    <ielementLink type="decomposition" run-mode="sequential">
        <ielement iref="1.9"/>
        <ielement iref="1.6"/>
    </ielementLink>
</ielement>
<ielement id="1.11" type="goal" name="ProfitBeAchieved"
main="true">
    <ielementLink type="means-end" run-mode="unique">
        <ielement iref="1.10"/>
    </ielementLink>
```

```
</ielement>
</boundary>
</actor>
<actor id="2" type="role" name="broker">
<boundary>
<ielement id="2.0" type="task" name="ObtainStockOptionQuotation">
<ielementLink type="decomposition" run-mode="sequential">
<ielement iref="8"/>
</ielementLink>
</ielement>
<ielement id="2.1" type="task" name="ProcessOrder" basic="true"/>
<ielement id="2.2" type="task" name="SimulateOrder">
<ielementLink type="decomposition" run-mode="sequential">
<ielement iref="2.0"/>
<ielement iref="2.1"/>
</ielementLink>
</ielement>
<ielement id="2.3" type="goal" name="OrderBeProcessed"
main="true">
<ielementLink type="means-end" run-mode="unique">
<ielement iref="2.2"/>
</ielementLink>
</ielement>
</boundary>
</actor>
<actor id="3" type="role" name="prophet">
<boundary>
<ielement id="3.1" type="task" name="AskQuotation">
<ielementLink type="decomposition" run-mode="sequential">
<ielement iref="7"/>
</ielementLink>
</ielement>
<ielement id="3.2" type="task" name="ApplyModel" basic="true"/>
<ielement id="3.4" type="task" name="GenerateTendency">
<ielementLink type="decomposition" run-mode="sequential">
<ielement iref="3.1"/>
<ielement iref="3.2"/>
</ielementLink>
</ielement>
<ielement id="3.5" type="goal" name="TendencyBeGenerated"
main="true">
<ielementLink type="means-end" run-mode="unique">
<ielement iref="3.4"/>
</ielementLink>
</ielement>
</boundary>
</actor>
<actor id="4" type="role" name="sensor">
```

```

<boundary>
  <ielement id="4.1" type="task" name="SeekStockQuotation"
  basic="true"/>
  <ielement id="4.2" type="goal" name="LastQuotationBeObtained"
  main="true">
    <ielmentLink type="means-end" run-mode="unique">
      <ielment iref="4.1"/>
    </ielmentLink>
  </ielment>
  <ielment id="4.3" type="task"
  name="SeekNextStockOptionQuotation" basic="true"/>
  <ielment id="4.4" type="goal"
  name="NextStockOptionQuotationBeObtained"
  main="true">
    <ielmentLink type="means-end" run-mode="unique">
      <ielment iref="4.3"/>
    </ielmentLink>
  </ielment>
</boundary>
</actor>
<ielment id="5" type="resource" name="tendency">
  <dependency>
    <depender iref="1.4"/>
    <dependee iref="3.5"/>
  </dependency>
</ielment>
<ielment id="6" type="task" name="process_order">
  <dependency>
    <depender iref="1.0"/>
    <dependee iref="2.3"/>
  </dependency>
</ielment>
<ielment id="7" type="resource" name="last_quotation">
  <dependency>
    <depender iref="3.1"/>
    <dependee iref="4.2"/>
  </dependency>
</ielment>
<ielment id="8" type="resource" name="next_stock-option_quotation">
  <dependency>
    <depender iref="2.0"/>
    <dependee iref="4.4"/>
  </dependency>
</ielment>
<actor id="10" type="agent" name="operator1">
  <actorLink type="plays" aref="1"/>
</actor>
<actor id="11" type="agent" name="broker1">

```

```

<actorLink type="plays" aref="2"/>
</actor>
<actor id="12" type="agent" name="prophet1">
    <actorLink type="plays" aref="3"/>
</actor>
<actor id="13" type="agent" name="sensor1">
    <actorLink type="plays" aref="4"/>
</actor>
</diagram>
</istarml>

```

O rastro de criação dos agentes (para ambos os casos é apresentado a seguir).

```

Actor prophet type: class istar.impl.Role
AskQuotation TaskDecomposition link: 1
sub-element: type:class istar.impl.Resource id:7 name:last_quotation
GenerateTendency TaskDecomposition link: 1
sub-element: type:class istar.impl.Task id:3.1 name:AskQuotation
sub-element: type:class basicelement.ApplyModel id:3.2 name:ApplyModel
TendencyBeGenerated Means-End link: 3
mean: type:class istar.impl.Task id:3.4 name:GenerateTendency
prophet Dependee element: id:5 - dependum: type:class istar.impl.Resource
name:tendency
Actor broker type: class istar.impl.Role
ObtainStockOptionQuotation TaskDecomposition link: 1
sub-element: type:class istar.impl.Resource id:8 name:next_stock-
option_quotation
SimulateOrder TaskDecomposition link: 1
sub-element: type:class istar.impl.Task id:2.0 name:ObtainStockOptionQuotation
sub-element: type:class basicelement.ProcessOrder id:2.1 name:ProcessOrder
OrderBeProcessed Means-End link: 3
mean: type:class istar.impl.Task id:2.2 name:SimulateOrder
broker Dependee element: id:6 - dependum: type:class istar.impl.Task
name:process_order
Actor operator type: class istar.impl.Role
PutOrder TaskDecomposition link: 1
sub-element: type:class istar.impl.Task id:6 name:process_order

```

Buy TaskDecomposition link: 1

sub-element: type:class basicelement.AssemblyBuyOrder id:1.1.2
name:AssemblyBuyOrder

sub-element: type:class istar.impl.Task id:1.0 name:PutOrder

sub-element: type:class basicelement.Register id:1.0.1 name:Register

Sell TaskDecomposition link: 1

sub-element: type:class basicelement.AssemblySellOrder id:1.2.2
name:AssemblySellOrder

sub-element: type:class istar.impl.Task id:1.0 name:PutOrder

sub-element: type:class basicelement.Register id:1.0.1 name:Register

ResetPosition TaskDecomposition link: 1

sub-element: type:class basicelement.ResetPosition id:1.3.1
name:ResetPosition

sub-element: type:class istar.impl.Task id:1.0 name:PutOrder

sub-element: type:class basicelement.Register id:1.0.1 name:Register

UserAccountDataAcquisition TaskDecomposition link: 1

sub-element: type:class basicelement.Money id:1.1.1 name:Money

sub-element: type:class basicelement.StockOption id:1.2.1 name:StockOption

MarketDataAcquisition TaskDecomposition link: 1

sub-element: type:class istar.impl.Resource id:9 name:last_optionquotation

UserAndMarketDataAcquisition TaskDecomposition link: 1

sub-element: type:class istar.impl.Task id:1.4.1
name:UserAccountDataAcquisition

sub-element: type:class istar.impl.Task id:1.4.2 name:MarketDataAcquisition

ObtainTendency TaskDecomposition link: 1

sub-element: type:class istar.impl.Resource id:5 name:tendency

OperationBeChosen Means-End link: 3

mean: type:class istar.impl.Task id:1.1 name:Buy

mean: type:class istar.impl.Task id:1.2 name:Sell

mean: type:class istar.impl.Task id:1.3 name:ResetPosition

mean: type:class basicelement.DoNothing id:0 name:DoNothing

MarketDataInterpretation TaskDecomposition link: 1

sub-element: type:class istar.impl.Task id:1.5 name:ObtainTendency

sub-element: type:class basicelement.Ponder id:1.6 name:Ponder

Awareness[MarketTendency] Means-End link: 3

mean: type:class istar.impl.ContextAwareness id:1.9 name:market_context

operate TaskDecomposition link: 1

sub-element: type:class istar.impl.Softgoal id:1.10
name:Awareness[MarketTendency]
sub-element: type:class istar.impl.Goal id:1.7 name:OperationBeChosen
ProfitBeAchieved Means-End link: 3
mean: type:class istar.impl.Task id:1.11 name:operate
Actor sensor type: class istar.impl.Role
LastQuotationBeObtained Means-End link: 3
mean: type:class basicelement.SeekStockQuotation id:4.1
name:SeekStockQuotation
NextStockOptionQuotationBeObtained Means-End link: 3
mean: type:class basicelement.SeekNextStockOptionQuotation id:4.3
name:SeekNextStockOptionQuotation
sensor Dependee element: id:7 - dependum: type:class istar.impl.Resource
name:last_quotation
sensor Dependee element: id:8 - dependum: type:class istar.impl.Resource
name:next_stock-option_quotation
Actor operator1 type: class istar.impl.IstarAgent
Actor operator1 plays Role: operator
Actor sensor1 type: class istar.impl.IstarAgent
Actor sensor1 plays Role: sensor
Actor broker1 type: class istar.impl.IstarAgent
Actor broker1 plays Role: broker
Actor prophet1 type: class istar.impl.IstarAgent
Actor prophet1 plays Role: prophet.

Anexo II – Rastro da 1^a hora de simulação sistema de investimento**Time=15/10/2010 10:12**

The mean:operate has been selected in order to satisfy/satisfy the
end:ProfitBeAchieved

The mean:market_context has been selected in order to satisfy/satisfy the
end:Awareness[MarketTendency]

Money=30000.0;Blocked=0.0

LowStockQtt=0

HighStockQtt=0

LowStockPrice=0.0

HighStockPrice=0.0

Tendency: time=15/10/2010

10:12;47.85;47.74;47.85;47.78824413;47.91569829;1

=====

Ponder evaluation of Buy: 1

Ponder evaluation of Sell: 0

Ponder evaluation of ResetPosition: 0

Ponder evaluation of DoNothing: 0

The mean:Buy has been selected in order to satisfy/satisfy the
end:OperationBeChosen

AssemblyBuyOrder executing

Order: Order [time=15/10/2010 10:12, currentQttLowStock=0,
currentQttHighStock=0, currentMoney=30000.0, currentBlocked=0.0,
orderType=1]

...

...

...

Time=15/10/2010 10:22

Money=60.0;Blocked=0.0

LowStockQtt=0

HighStockQtt=37400

LowStockPrice=1.79

HighStockPrice=0.8

operator1 >>> Starting Ponder

Tendency: time=15/10/2010 10:22;47.86;47.74;48.0;47.7805772;47.93432119;0

StopLoss achieved - HighOption current price:0.78 HighOption ordered price:0.8
stopLoss %:0.02

=====

Ponder evaluation of Buy: 0

Ponder evaluation of Sell: 0

Ponder evaluation of ResetPosition: 1

Ponder evaluation of DoNothing: 0

The mean:ResetPosition has been selected in order to satisfy/satisficy the
end:OperationBeChosen

AssemblyBuyOrder executing

Order: Order [time=15/10/2010 10:22, currentQttLowStock=0,
currentQttHighStock=37400, currentMoney=60.0, currentBlocked=0.0,
orderType=0]

...

...

...

Time=15/10/2010 10:24

Money=28464.0;Blocked=0.0

LowStockQtt=0

HighStockQtt=0

LowStockPrice=1.76

HighStockPrice=0.76

operator1 >>> Starting Ponder

Tendency: time=15/10/2010 10:24;47.83;47.76;48.0;47.75334994;47.90355951;-

1

=====

Ponder evaluation of Buy: 0

Ponder evaluation of Sell: 1

Ponder evaluation of ResetPosition: 0

Ponder evaluation of DoNothing: 0

The mean:Sell has been selected in order to satisfy/satisficy the
end:OperationBeChosen

AssemblySellOrder executing

Order: Order [time=15/10/2010 10:24, currentQttLowStock=0,
currentQttHighStock=0, currentMoney=28464.0, currentBlocked=0.0,
orderType=-1]

...

...

Time=15/10/2010 10:41

Money=17609.0;Blocked=20600.0

LowStockQtt=-10300

HighStockQtt=10300

LowStockPrice=1.71

HighStockPrice=0.76

operator1 >>> Starting Ponder

Tendency: time=15/10/2010

10:41;47.65;47.55;47.86;47.56506166;47.72722582;**0**

StopGain achieved - LowOption current price:1.6 LowOption ordered price:1.71

stopGain %:0.02

=====

Ponder evaluation of Buy: 0

Ponder evaluation of Sell: 0

Ponder evaluation of ResetPosition: 1

Ponder evaluation of DoNothing: 0

The mean:ResetPosition has been selected in order to satisfy/satisficy the

end:OperationBeChosen

AssemblyBuyOrder executing

Order: Order [time=15/10/2010 10:41, currentQttLowStock=-10300,
currentQttHighStock=10300, currentMoney=17609.0, currentBlocked=20600.0,
orderType=0]

...

...

...

Time=15/10/2010 10:42

Money=28899.0;Blocked=0.0

LowStockQtt=0

HighStockQtt=0

LowStockPrice=1.59

HighStockPrice=0.69

operator1 >>> Starting Ponder

Tendency: time=15/10/2010 10:42;47.6;47.55;47.82;47.5202269;47.67529692;**-1**

=====

Ponder evaluation of Buy: 0

Ponder evaluation of Sell: 1

Ponder evaluation of ResetPosition: 0
Ponder evaluation of DoNothing: 0
The mean:Sell has been selected in order to satisfy/satisficy the
end:OperationBeChosen
AssemblySellOrder executing
Order: Order [time=15/10/2010 10:42, currentQttLowStock=0,
currentQttHighStock=0, currentMoney=28899.0, currentBlocked=0.0,
orderType=-1]
...
...
...

Time=15/10/2010 10:52

Money=17089.0;Blocked=21400.0
LowStockQtt=-10700
HighStockQtt=10700
LowStockPrice=1.59
HighStockPrice=0.69
operator1 >>> Starting Ponder
Tendency: time=15/10/2010
10:52;47.62;47.55;47.67;47.55828974;47.68608869;**1**
=====

Ponder evaluation of Buy: 1
Ponder evaluation of Sell: 0
Ponder evaluation of ResetPosition: 0
Ponder evaluation of DoNothing: 0

The mean:Buy has been selected in order to satisfy/satisficy the
end:OperationBeChosen
AssemblyBuyOrder executing
Order: Order [time=15/10/2010 10:52, currentQttLowStock=-10700,
currentQttHighStock=10700, currentMoney=17089.0, currentBlocked=21400.0,
orderType=1]
...
...
...

Time=15/10/2010 10:55

Money=-7.0;Blocked=0.0
LowStockQtt=0

HighStockQtt=40600
LowStockPrice=1.61
HighStockPrice=0.71
operator1 >>> Starting Ponder
Tendency: time=15/10/2010 10:55;47.62;47.6;47.67;47.56470825;47.68375474;**0**
StopLoss achieved - HighOption current price:0.69 HighOption ordered price:0.71
stopLoss %:0.02
=====

Ponder evaluation of Buy: 0
Ponder evaluation of Sell: 0
Ponder evaluation of ResetPosition: 1
Ponder evaluation of DoNothing: 0
The mean:ResetPosition has been selected in order to satisfy/satisficy the
end:OperationBeChosen
AssemblyBuyOrder executing
Order: Order [time=15/10/2010 10:55, currentQttLowStock=0,
currentQttHighStock=40600, currentMoney=-7.0, currentBlocked=0.0,
orderType=0]
...
...
...
Time=15/10/2010 10:57
Money=28799.0;Blocked=0.0
LowStockQtt=0
HighStockQtt=0
LowStockPrice=1.6
HighStockPrice=0.71
operator1 >>> Starting Ponder
Tendency: time=15/10/2010
10:57;47.58;47.58;47.67;47.52243387;47.64497706;**-1**
=====

Ponder evaluation of Buy: 0
Ponder evaluation of Sell: 1
Ponder evaluation of ResetPosition: 0
Ponder evaluation of DoNothing: 0
The mean:Sell has been selected in order to satisfy/satisficy the
end:OperationBeChosen

AssemblySellOrder executing

Order: Order [time=15/10/2010 10:57, currentQttLowStock=0,
currentQttHighStock=0, currentMoney=28799.0, currentBlocked=0.0,
orderType=-1]

...

...

...

==//==