Modularizando Modelos i*: uma Abordagem baseada em Transformação de Modelos

Márcia Lucena [1,2], Carla Silva[2], Emanuel Santos[2], Fernanda Alencar[3], Jaelson Castro[2]

[1]*Departamento de Informática, Universidade Federal do Rio Grande do Norte, Brasil*
[2]*Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil*
[3]*Departamento de Eletrônica e Sistemas, Universidade Federal de Pernambuco, Brasil*
{mjnrl, ctlls, ebs, jbc}@cin.ufpe.br, fmra@ufpe.br

Abstract

A Engenharia de Requisitos (ER) é uma atividade chave em quase todo processo de engenharia de software. i é uma abordagem orientada a metas bastante adotada na comunidade de ER, pois descreve o sistema de software e seu ambiente em termos de atores e suas dependências. Apesar do i* oferecer uma rica notação que permite definir o raciocínio de como os requisitos são alcançados, não há uma sistematização na forma como seus elementos são modularizados, comprometendo sua evolução e a transição para as fases seguintes. Neste contexto, este artigo apresenta uma estratégia para melhorar a modulardade dos modelos i* baseada em separação de interesses. São propostos um processo e um conjunto de regras de transformação para gerar modelos mais modulares. Além disso, restrições OCL são incorporadas a um metamodelo definido para i* garantindo assim que os modelos gerados pela abordagem estejam de acordo com a sintaxe do i*. Para avaliar nossa abordagem, usamos métricas de modularização para modelos i* antes e depois de aplicar nossa abordagem em um estudo de caso em comércio eletrônico.*

1. Introdução

O i* é uma abordagem orientada a *goals*[1] bastante usada na academia e na indústria [18]. Através desta abordagem podemos descrever tanto dependências sociais e intencionais no ambiente organizacional como também atributos de qualidade e funcionalidades do software. Além disso, o i* tem uma semântica rica de construtores que permite a criação de modelos para descrever requisitos funcionais e não funcionais. No entanto, quanto maior for o problema a ser descrito, maior será a dificuldade em gerenciar informações relacionadas com os requisitos, já que estas não estão adequadamente modularizadas, comprometendo o entendimento e a evolução dos modelos e a transição para as fases seguintes.

Para melhorar a modularidade dos modelos i* e, conseqüentemente, melhorar o entendimento destes, mecanismos de decomposição que dividem o software em partes gerenciáveis e significativas podem ser usados através do princípio de dividir para conquistar. Apesar do i* incorporar mecanismos de decomposição baseado em atores estratégicos, estes não são suficientes para deixar o ator que representa o software bem modularizado. Isto porque dentro deste ator pode conter informações de diferentes domínios dificultando o real entendimento do problema a ser atacado. Desta forma, um mecanismo de decomposição poderia ser usado para decompor o ator que representa o software em novos atores facilitando assim o entendimento e o gerenciamento destes atores.

As regras de transformação apresentadas neste artigo foram originalmente apresentadas em [11] com a finalidade de diminuir o tamanho dos modelos i* e usando métricas de complexidade para avaliá-los. Neste novo artigo definimos uma estratégia sistemática para decompor modelos i* baseado em separação de interesses e complementamos as regras de transformação de modelo com restrições escritas em OCL. Estas regras foram definidas para garantir que os modelos gerados estejam de acordo com a sintaxe do i*. Além disso, as métricas de modularidade proposta por [14] foram adaptadas e usadas para avaliar os modelos gerados por esta abordagem.

Regra de transformação de modelo é o conceito principal de muitas abordagens de transformação de modelos [5], pois descreve a lógica da transformação. As regras de transformação produzirão modelos sintaticamente corretos e semanticamente equivalentes tendo um metamodelo da linguagem como base. No nosso

[1] *Goal* podem ser traduzido para meta, mas neste trabalho vamos deixar em inglês para manter coerência com o acrônimo GORE (*Goal oriented requirements engineering*).

caso o metamodelo no qual nos baseamos para desenvolver este trabalho foi proposto em [10] e está de acordo com i* original de Eric Yu [19]. Desta forma, o objetivo desta abordagem é gerar modelos i* mais fáceis de entender e de evoluir. A atual versão de nossa abordagem está sendo desenvolvida na ferramenta ISTarTool [3] onde podemos desenvolver a modelagem e executar as transformações em uma mesma ferramenta.

Este artigo está organizado da seguinte forma. Seção 2 apresenta uma visão geral do i* usando um exemplo de comércio eletrônico. Seção 3 apresenta nossa abordagem pra melhorar a modularidade dos modelos i* e mostrando algumas considerações. Seção 4 apresenta o uso da nossa abordagem e mostra uma avaliação do uso. Seção 5 apresenta trabalhos relacionados e finalmente a Seção 6 apresenta uma visão geral do trabalho e discute alguns trabalhos futuros.

2. Visão Geral do i*

i* tem dois níveis de abstração: o modelo SD (Strategic Dependency – dependência estratégica) e o modelo SR (Strategic Rationale – raciocínio estratégico). Por simplicidade usaremos modelo SD e modelo SR daqui por diante.

Para mostrar os conceitos principais de i* vamos considerar o estudo de caso Media@ apresentado em [4]. Medi@ é um sistema de comercio eletrônico que vende e envia produtos de media através de uma aplicação *front-end*. A Figura 1 apresenta uma parte do modelo i* para este sistema apresentando os dois níveis de abstração. O modelo SD apresenta os atores e suas dependências, enquanto que o modelo SR apresenta os detalhes internos dos atores, neste caso aparece o detalhamento do ator Medi@ que representa o software, mostrando seus elementos internos e tendo alguns tratando as dependências externas. Os subgrafos destacados A, B e C na Figura 1 serão discutidos nas seções seguintes.

No modelo SD um ator pode depender de outro para satisfazer um *goal*, para executar uma tarefa, fornecer um recurso ou satisfazer um *softgoal*. *Softgoals* estão relacionados a requisitos não-funcionais, enquanto goals, tarefas e recursos estão relacionados com funcionalidades do software [18]. O ator que depende do outro é chamado de *Depender* enquanto que o outro ator que recebe a requisição é chamado de *Dependee*[2]. Enquanto que o modelo SR é usado para:

- Descrever os interesses e motivações dos participantes no processo;
- Permitir a avaliação de possíveis alternativas para alcançar os *softgoals*;
- Detalhar as razões existentes em relação as dependências entre atores.

Além disso, no modelo SR, onde os atores são detalhados aparecem elementos intencionais para descrever a análise de oportunidades e vulnerabilidades e o tratamento das dependências entre atores. Os elementos intencionais podem ser *goals*, tarefas, recursos e *softgoals*. Também existem novos tipos de relacionamentos tais como *means-end*, decomposição-tarefa e contribuição.

Um relacionamento de decomposição-tarefa existe entre uma tarefa e suas partes, mostrando como uma tarefa é executada. Na Figura 1, o ator Medi@ apresenta a tarefa raiz *Gerenciar Loja pela Internet* e o relacionamento com suas partes. Esta tarefa é refinada, através de *goals* (*Pesquisa por Item a ser Manipulada e Pedidos pela Internet a serem Manipulados*), de tarefas (*Adaptação* e *Produzir Estatísticas*) e *softgoals* (*Adaptabilidade, Segurança, Disponibilidade* e *Atração de Novos Clientes*). Estes elementos por sua vez são detalhados através de relacionamentos do tipo decomposição-tarefa, means-end e contribuição para descrever os requisitos do sistema Medi@.

Um relacionamento *means-end* descreve uma ligação de um elemento meio (*means*) para alcançar um *goal* (*end*), ou seja, um *means* é uma alternativa para realizar um end (normalmente representado por um elemento do tipo goal) mostrando como este goal pode ser atingido . Por exemplo, no ator Medi@ (Figura 1), existe um relacionamento means-end da tarefa *Escolher Item Disponível* (*means*) para o *goal Item a ser Selecionado* (end).

O relacionamento do tipo contribuição descreve uma contribuição de um means (tarefa ou goal) para a realização de um end (softgoal). Este tipo de relacionamento fornece um raciocínio qualitativo usando um esquema de valores para representar a contribuição (exemplo, Help, Hurt, Make[3]) [8]. Por exemplo, na Figura 1, a tarefa *Atualizar GUI* contribui positivamente (Help) para satisfação do softgoal *Disponibilidade*.

Estes construtores são utilizados para analisar, descobrir e especificar os requisitos do software, contribuindo para produzir modelos i* contendo informações sobre características do ambiente organizacional como também a do software a ser desenvolvido. Desta forma, quanto maior for o detalhamento dos modelos i* menos modularizados eles se tornam, principalmente por causa do refinamento do ator que representa o

[2] Depender e Dependee não tem uma tradução aceitável.

[3] Não foi traduzido para ficar coerente com a sintaxe do i*. É traduzido respectivamente para Contribui positivamente, Contribui negativamente, Realiza.

software. Na metodologia Tropos [4] este ator é denominado por ator software. Neste artigo nós iremos adotar também esta denominação para designar os atores que detalham informações sobre software. Uma forma de melhorar a modularidade do ator que representa o software (daqui por diante designado como ator software) seria adotar uma estratégia de decomposição baseada em separação de interesses para dividir o ator que representa o software em atores mais coesos e fáceis de gerenciar.

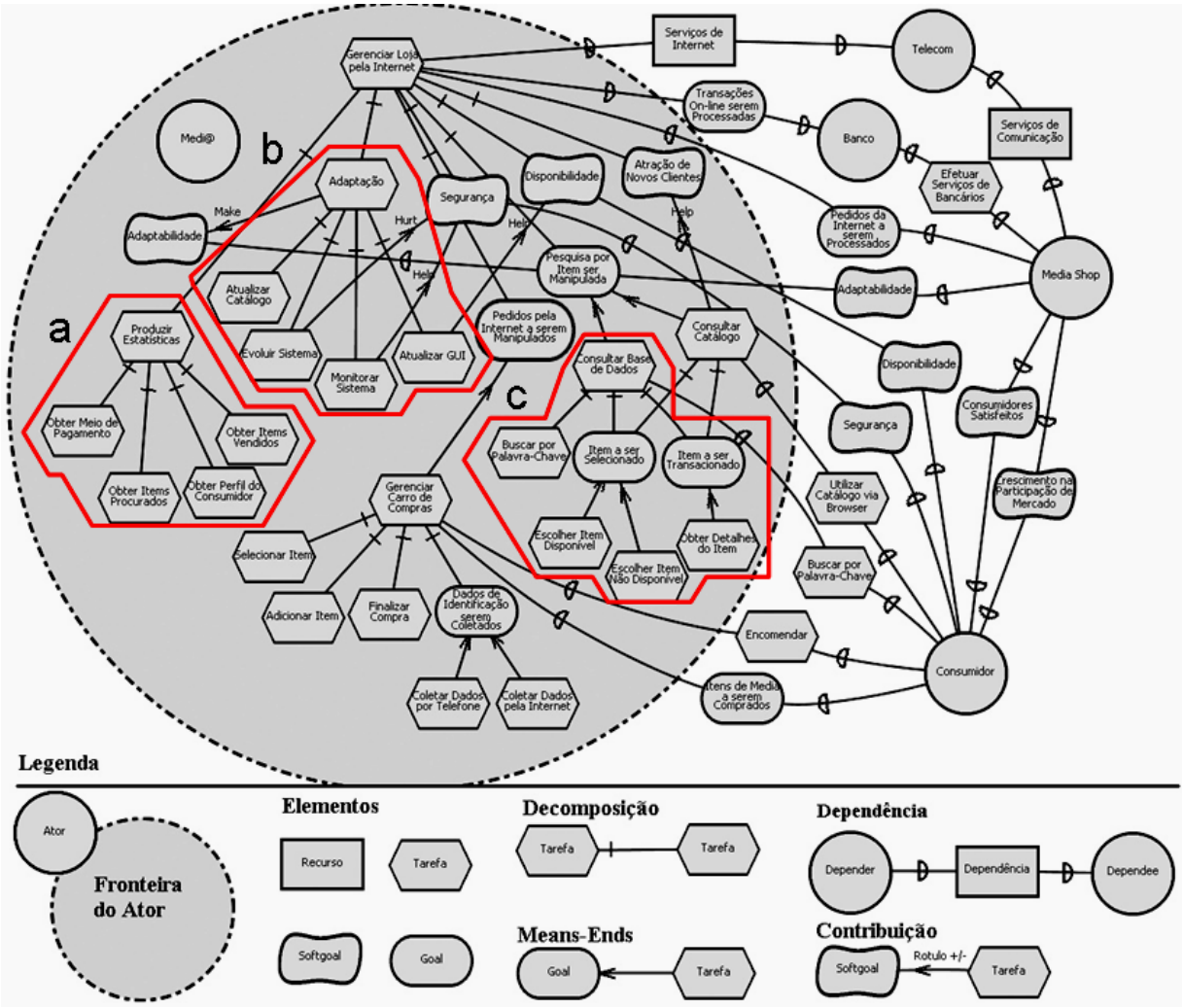


Figura 1 Modelo i* do Sistema Medi@

3. Uma abordagem para modularizar modelos i*

Com o objetivo de melhorar a modularidade dos modelos i* nós propomos um processo composto por três atividades:

(i) Avaliação dos modelos i*;
(ii) Análise dos Elementos Internos;
(iii) Aplicação de Regras de Transformação de Modelos.

A Figura 2 apresenta uma visão geral deste processo. Para que estas atividades sejam realizadas é necessário usar respectivamente:

- métricas para avaliar o grau de modularidade dos modelos i* iniciais e modelos finais;
- heurísticas para guiar a decomposição do ator software;
- um conjunto de regras a serem usadas para a transformar modelos i*.

Ao final do processo as métricas de avaliação são usadas para avaliar o nível de modularidade dos modelos i* gerados após o processo e comparar com as métricas geradas inicialmente. Este é um processo semi-automático, pois algumas atividades podem ser automatizadas como a avaliação dos modelos e a aplicação das regras. Enquanto que a atividade de análise de elementos internos dependerá do engenheiro de requisitos e de sua experiência no domínio em questão, neste caso são apresentadas heurísticas que ajudarão nesta atividade.

3.1 Avaliação dos Modelos i*

O processo é iniciado a partir da avaliação dos modelos i* iniciais, usando métricas de modularidade. Uma vez que resultados experimentais têm indicado que freqüentemente modelos i* sofrem de falta de modularidade [6], nós adaptamos as métricas de modularidade proposta por [14] para modelos i*. No trabalho de [14] as métricas relacionadas à modularidade dos artefatos de software são de separação de interesses, acoplamento e coesão.

A métrica para separação de interesses avalia se os interesses estão espalhados em diversos componentes e

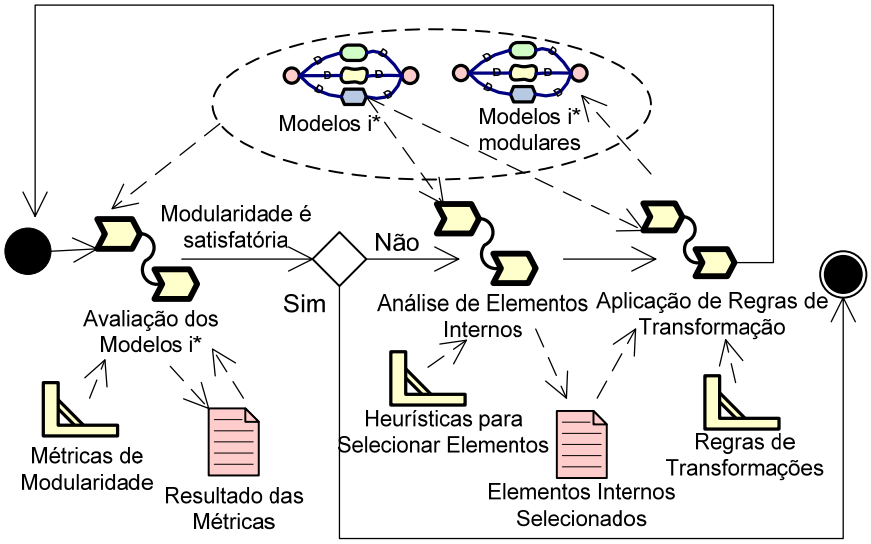


Figura 2 Processo para Modularizar Modelos i*

se há uma interconexão entre interesses diferentes. A métrica para acoplamento avalia quantos componentes dependem de um componente em particular. A métrica para coesão determina quanto de informação interna de um componente não contribui para um interesse específico. A adaptação destas métricas é apresentada na Seção 4.

Esta avaliação pode ajudar o engenheiro de requisitos a escolher que elementos dentro de um ator devem ser movidos para outro ator a fim de melhorar a modularidade de modelos i*.

O resultado desta avaliação pode ajudar a decidir se é necessário aplicar as regras da transformação ou não. Se o resultado da avaliação mostra que o modelo i* precisa ser modularizado, nós podemos começar a segunda atividade do processo que será descrita na próxima Seção.

3.2 Análise de Elementos Internos

Nesta atividade os elementos que fazem parte do ator software são analisados de acordo com o critério de separação de interesses e modularização de elementos que não estão fortemente relacionados com o domínio da aplicação. Este critério foi definido para que estes elementos possam ser facilmente reusados em diferentes domínios. Por exemplo, no modelo i* apresentado na Figura 1, que descreve o detalhamento do ator Medi@ e seus relacionamentos, o engenheiro de requisitos pode identificar aqueles elementos que estão mais relacionados com o domínio da aplicação (neste caso Comércio Eletrônico) e aqueles que não são. Por exemplo, os sub-grafos destacados em A, B e C da Figura 1 são independentes do domínio de comércio eletrônico e por isso podem ser movidos do ator software (Medi@) para outro novo ator. Após fazer isto, o modelo final será semanticamente equivalente com o modelo original.

No final desta atividade os atores que representam o software ficam mais fáceis de entender e manter já que

agregam poucos elementos internos. Além disso, esta abordagem promove reuso dos atores que representam o software. Desde que novos atores do software são independentes do domínio da aplicação, eles podem estar presente na especificação dos requisitos do sistema de outro domínio. Portanto, ao separar os elementos independentes em outros atores pode nos levar a promover reusabilidade e manutenabilidade da especificação do software em requisitos. O exemplo apresentado na Figura 1 mostra que os elementos relativos a funcionalidade de Produção de Estatística, sub-grafo A, poderia ser usado como parte de um software de um domínio diferente. O mesmo raciocínio poderia ser aplicado aos dois sub-grafos destacados na Figura 1 (sub-grafo B e C).

Com o objetivo de auxiliar o engenheiro de requisitos na escolha dos elementos que podem ser retirados do ator software, nós propomos as seguintes heurísticas:

(H1) Procurar elementos internos no ator software que são independentes do domínio da aplicação;

(H2) Checar se estes elementos podem ser movidos do ator software para outro ator sem comprometer o comportamento e o entendimento dos detalhes internos do ator;

(H3) Checar se estes elementos podem ser reusados em diferentes domínios.

É importante observar que a decisão sobre quais elementos serão escolhidos de acordo com estas heurísticas dependerá da experiência do engenheiro de requisitos. De acordo com o elemento escolhido uma das regras de transformação será aplicada. Estas regras são apresentadas na seção a seguir.

3.3 Aplicação de Regras de Transformação

Nesta atividade uma das regras de transformação deverá ser escolhida a depender do tipo de elemento interno a ser movido e seu relacionamento dentro do sub-grafo. Estas regras de transformações são aplicadas a elementos do tipo *goals*, *softgoals* e tarefas, pois estes tipos elementos podem ser refinados justificando a transferência para outro ator.

As regras de transformação têm como objetivo delegar elementos internos do ator software para outros (novos) atores. Esta delegação deve garantir que novos atores e o ator original tenham um relacionamento de dependência. Desta forma o modelo original e o modelo final sejam semanticamente equivalentes que poderá ser garantida através da definição semântica destes modelos. Não faz parte do escopo deste artigo mostrar a equivalência entre estes modelos.

Outro ponto que queremos destacar é que para garantir que os modelos i* gerados pela regra sejam bem formados nós especificamos em [9] um metamodelo para a linguagem i* incluindo um conjunto de restrições existentes em [19] para representar os relacionamentos válidos entre os elementos internos. No entanto tivemos a necessidade de acrescentar a este conjunto de restrições duas novas expressões em OCL para garantir as válidas dependências entre elementos internos de diferentes atores. Desta forma, os modelos gerados pelas regras de transformação devem estar de acordo com as restrições do metamodelo. A necessidade destas novas restrições se deve ao fato de que uma dependência saindo de um elemento interno tem o mesmo significado de um relacionamento de decomposição-tarefa, ou seja, a dependência é um sub-elemento do elemento interno [19]. Por exemplo, a tarefa *Gerenciar Loja pela Internet* que aparece dentro do ator Medi@ (Figura 1) depende da satisfação do *goal Processar Transação Monetárias on-line* (dependência). Da mesma forma, uma dependência entrando no ator e sendo tratada por um elemento interno, tem uma equivalência com relacionamento *means-end* , ou seja, o elemento interno é um *means* para realizar a dependência (*end*) [19]. Por exemplo, na Figura 1 aparece que para realizar o *goal* Processar Pedidos pela Internet (dependência) depende da tarefa Gerenciar Loja na Internet (elemento interno do ator Medi@).

Considerando as restrições de dependências entrando e saindo do ator, a Tabela 1 apresenta as situações de dependências válidas entre elementos internos de atores diferentes.

Tabela 1 Válidas saídas e entradas de dependências

Depender	Dependência	Dependee
Tarefa	Tarefa	Tarefa
Tarefa	Goal	Tarefa
Tarefa	Goal	Goal
Tarefa	Recurso	Tarefa
Tarefa	Softgoal	Softgoal
Tarefa	Softgoal	Tarefa

Através do uso destas restrições no metamodelo, onde as regras de transformação são baseadas, os modelos gerados pela abordagem deverão estar de acordo com a sintaxe do i*. Assim, baseada na Tabela 1, duas novas expressões OCL são incorporadas ao metamodelo proposto em [9]: Expressão OCL 1, que está associada ao depender, e Expressão OCL 2, que está associada ao dependee e a dependência. As expressões estão descritas a seguir.

OCL Expression 1

```
context DependerLink inv:
```

```
  self.depender.oclIsTypeOf(Actor) OR
self.depender.oclIsTypeOf(Task)
```

OCL Expression 2

```
context DependeeLink inv:
  (self.dependee.oclIsTypeOf(Actor) OR
not (self.dependee.oclIsTypeOf(Resource))
AND self.dependee.oclIsTypeOf(Goal)
implies self.dependency.type=Goal AND
    self.dependee.oclIsTypeOf(SoftGoal)
implies self.dependency.type=SoftGoal
```

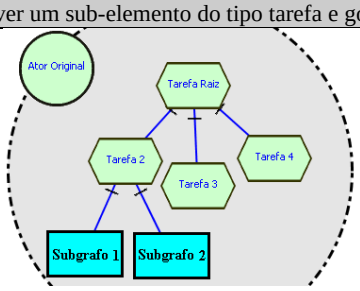
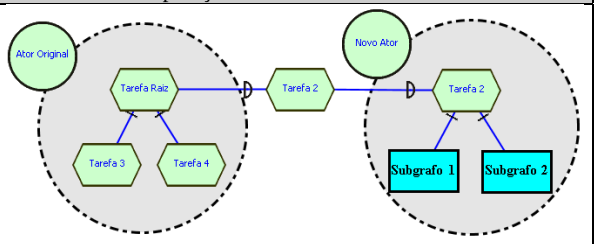
3.3.1 Apresentação das Regras

As regras de transformação são estruturadas da seguinte forma:

- Nome da regra, por exemplo, Regra de Transformação 1 ou RT1;
- Uma figura para ilustrar o contexto no qual o modelo original é encontrado;
- Uma figura para ilustrar o modelo final depois de aplicada a regra;
- Uma descrição da pré-condição para a regra ser aplicada e
- Uma descrição dos efeitos produzidos pela regra.

São quatro regras de transformação: RT1, RT2, RT3 e RT4. RT1 e RT2 são para tratar, respectivamente, os elementos que participam de relacionamentos do tipo decomposição-tarefa e means-end. Enquanto que RT3 e RT4 são regras mais restritivas para tratar situações, relacionadas respectivamente, com softgoal e elementos que estão participando de diferentes sub-grafos.

Tabela 2. RT para mover sub-elementos

RT1- Mover um sub-elemento do tipo tarefa e goal em um relacionamento decomposição-tarefa	
	 **Modelo Final**
Pré-condição: Uma tarefa raiz de um grafo é decomposta em sub-elementos (tarefas ou *goals*) que são as raízes de sub-grafos e não compartilham com outros sub-grafos.	
Efeitos: O sub-grafo que é independente do domínio da aplicação (o sub-grafo cujo elemento raiz é Tarefa 2) é movido do ator original para um novo ator. Este novo ator tem o mesmo nome da raiz da qual o sub-grafo foi transferido. Esta raiz irá ser replicada como um relacionamento de dependência do mesmo tipo relacionando o ator original, como um depender, e o novo ator, como um dependee. Além disso, todas as dependências externas existentes com este sub-grafo serão transferidas também para o novo ator.	

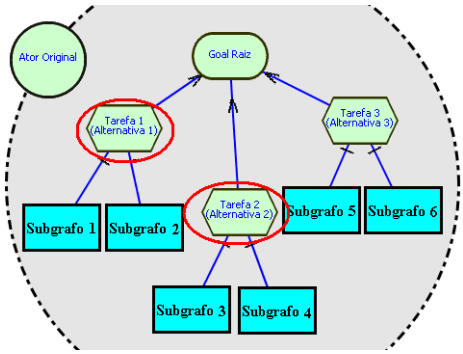
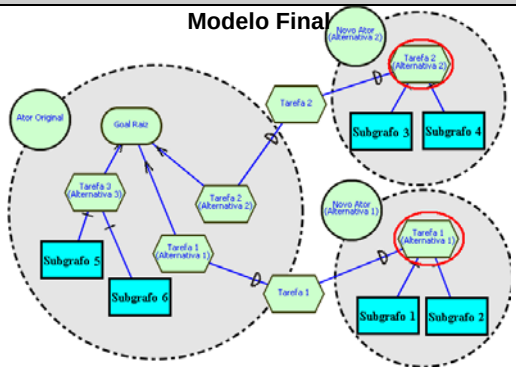
RT1 apresenta uma regra de transformação que move um sub-elemento presente em uma decomposição-tarefa para outro ator (Tabela 2). Esta regra de transformação foi definida baseada na equivalência da dependência saindo do ator [19], apresentada na seção anterior, onde uma dependência saindo de um elemento interno tem o mesmo significado de uma decomposição-tarefa. Também, nesta transformação, os elementos intencionais presentes nas dependências entrando no novo ator são replicados neste novo ator, como ocorre na versão do i* usada em Tropos [2].

RT2 apresenta um modelo original, onde o sub-grafo a ser movido tem uma raiz como *means* em um relacionamento *means-end*. Neste caso, o sub-grafo é movido para um novo ator, o elemento raiz continua no ator original, uma dependência do mesmo tipo é criada na dependência e um novo ator aparece.

Após aplicar as regras RT1 e RT2, pode ocorrer que o modelo final não esteja em conformidade com a notação do i* sugerida por [19] ou mesmo pelo Guia do iStar [8]. Por exemplo, um relacionamento de contribuição no modelo original pode resultar em um relacionamento cruzando de um ator para outro ator (veja Tabela 4, modelo original). Neste caso, nós precisamos usar uma regra corretiva, tal como RT3. RT3 sugere a replicação de um softgoal envolvido em um relacionamento de contribuição tanto dentro do ator original quanto em uma dependência de softgoal saindo do novo ator para o ator original. Esta regra foi definida para preservar a informação sobre relacionamentos de contribuição, mantendo a informação sobre estes rela-

cionamentos de contribuição e a coerência dos modelos i* como é proposto em [9]. No entanto, observando a Tabela 1, podemos notar que apenas o elemento do tipo tarefa como depender pode ser válido. Sendo assim, nós modificamos a forma como o softgoal é tratado no trabalho de [9] e ao invés da dependência sair do softgoal como é sugerido em [9] a solicitação da dependência vem do elemento raiz que nos casos das regras sempre será uma tarefa. Isto ocorre já que para os casos das regras acima apenas as tarefas poderão levar os sofgoals já que não existe softgoals como means para goals mas sim softgoals como sub-elemento de tarefas [19].

Tabela 3. RT mover alternativas

RT2 - Mover sub-grafo "means" em relacionamentos means-end	
 **Modelo Original**	**Modelo Final** 
Pré-condição: Existe um elemento do tipo goal que é um "end" em um ou mais relacionamentos means-ends e pelo menos um dos seus "means" é um sub-grafo (alternativa) que não compartilha qualquer elemento com outros sub-grafos.	
Efeito: Cada sub-grafo independente (alternativa) será transferido para um novo ator com o mesmo nome do sub-grafo raiz. As raízes dos sub-grafos transferidos permanecem no ator original para manter os relacionamentos "means-end" original. Uma nova dependência do mesmo tipo do elemento "means" é criada partindo do ator original para a raiz do sub-grafo transferido para o novo ator.	

RT4 (Tabela 5) é aplicado quando os sub-grafos a serem movidos tem um sub-elemento compartilhado (Tarefa 3, no modelo original apresentado na Tabela 5) com outros sub-grafos. Uma das regras anteriores RT1 ou RT2 precisa ser aplicada para mover um sub-grafo (neste caso o sub-grafo com Goal 1 como raiz) para outro ator. Mas existe um elemento compartilhado então para casos como este a regra RT4 é usada. Esta regra sugere que uma política de prioridade seja estabelecida para escolher com que sub-grafo o elemento compartilhado (Tarefa 3) permanecerá. Desta forma os seguintes casos são analisados:

- Checar os tipos de elementos raízes no sub-grafo que compartilham o elemento. O sub-grafo cujo tipo da raiz tiver a maior prioridade ficará com o elemento compartilhado. A prioridade para o tipo do elemento raiz é *goal*, *softgoal* e tarefa (nesta ordem);
- Se todas as raízes dos sub-grafos são do mesmo tipo, checar a posição da raízes dos sub-grafos que compartilham o elemento. A raiz do sub-grafo que é mais próxima da raiz mais geral ficará com o elemento compartilhado;

Se todas as raízes dos sub-grafos são do mesmo tipo e estiverem no mesmo nível em relação a raiz do grafo geral, então o elemento compartilhado ficará com o sub-grafo que permanecer no ator original.

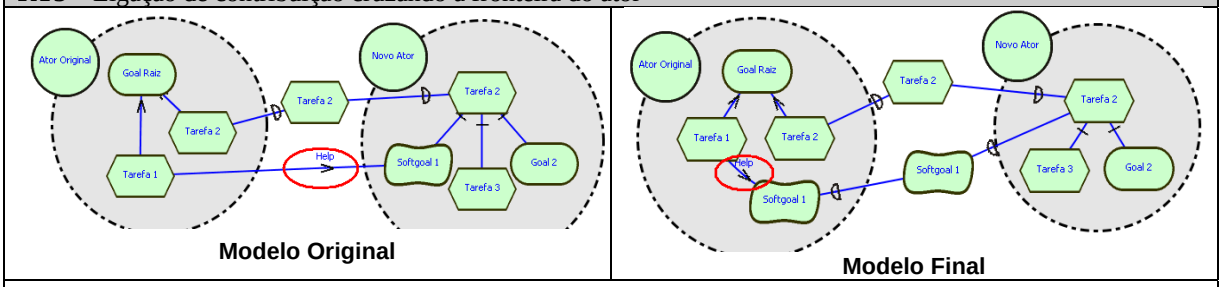
3.4 Considerações

Para melhorar a modularidade dos modelos i* foi usado um mecanismo de decomposição baseada em separação de interesses e regras de transformação. Uma forma sistemática para usar esta abordagem foi apresentada e o primeiro passo deste processo é a avaliação da modularidade do modelo i*.

Na atividade relacionada com Análise de elementos internos foram propostas heurísticas (H1, H2, H3) para auxiliar o engenheiro de requisitos a decidir quais elementos poderão ser movidos. Estas heurísticas foram

inspiradas na abordagem de especificação de requisitos de [15] que usou mecanismos para extrair elementos de especificações de requisitos, os modularizando em uma parte separada, este trabalho foi usado no contexto de Early Aspects [13].

Tabela 4. RT para mover uma ligação de contribuição

RT3 – Ligação de contribuição cruzando a fronteira do ator

Pré-condição: Há elementos como uma tarefa, *goal* or softgoal contribuindo para um softgoal que está fora da fronteira do ator.
Efeitos: Um softgoal, com o mesmo nome, deve ser criado dentro do ator para onde a contribuição está se dirigindo mantendo assim a ligação de contribuição dentro da fronteira do ator. Uma dependência de softgoal, com o mesmo nome, deve ser criada. Esta dependência parte do elemento acima do softgoal (Tarefa 2) do novo ator para o softgoal do ator original.

Apesar das H1 e H2 estarem relacionadas elas foram separadas para deixar mais claro o entendimento. H3 foi usada para promover o reuso de requisitos como defende a comunidade de linha de produto de software. [12].

As regras de transformação foram definidas para mover apenas elementos do tipo *goal* e tarefa, pois é possível garantir que o modelo final seja equivalente ao modelo original, baseado em propriedades definidas em i* [19], comentada ao apresentar a aplicação das regras de transformação.

RT2 contribui para tornar mais fácil a visualização das alternativas relacionadas aos *goals*, pois o engenheiro de requisitos pode avaliar cada alternativa separadamente.

No entanto, não foram definidas regras de transformações para mover elementos do tipo recurso e nem *softgoal*. Elementos do tipo softgoal podem ser refinados usando relacionamentos de contribuição e para manter a integridade do modelo produzido, decidimos mantê-los no ator original. Além disso, os softgoals recebem contribuição de diferentes sub-grafos sua transferência para outro ator implicaria no surgimento de muitas dependências. A regra relacionada com softgoal é a RT3 que evita que relacionamentos de contribuição cruzem a fronteira do ator para atingir a fronteira de outro ator.

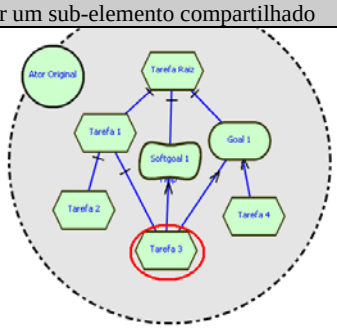
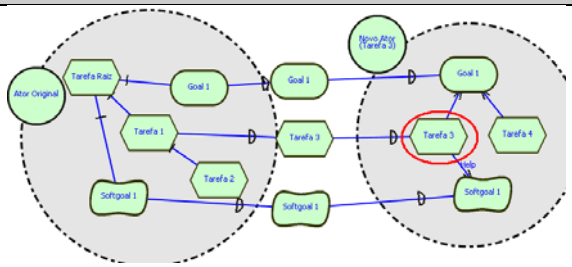
Enquanto que, os elementos do tipo recurso não precisam de uma regra especifica para movê-los já que eles não são decompostos. No entanto, se eles fazem parte de um sub-grafo que será movido para outro ator então o recurso acompanha o sub-grafo.

Quanto mais elementos a serem movidos mais atores relacionados com software e suas dependências são criados. Em algumas situações estas características podem aumentar de forma desordenada. Para lidar com esta tendência uma heurística extra poderia ser adicionada nesta atividade para primeiro analisar se os elementos a serem movidos podem ser inseridos em um ator já existente. Para identificar um possível ator, o engenheiro de requisitos poderia analisar se o raciocínio do ator e os novos elementos estão lidando com a mesma parte do problema. Então este ator deveria ser capaz de lidar com estas novas responsabilidades sem comprometer com seu objetivo fundamental.

4. Aplicando a Abordagem

Após o engenheiro de requisitos ter decidido aplicar a abordagem com objetivo de aumentar a modularidade do modelo SR do ator Medi@ (Figure 1), ele pode identificar elementos que possam ser removidos para outros atores. Por exemplo, considerando as heurísticas H1 e H2 apresentadas na seção 3.2, as tarefas *Produzir Estatísticas*, *Adaptação* e *Consultar Base de Dados* foram identificadas. As tarefas *Produzir Estatísticas* e *Adaptação* apresentam situações que não estão estritamente relacionados ao domínio de *e-commerce*. Portanto, elas podem ser movidas do ator original sem interferir no seu comportamento e propósito original.

Tabela 5. RT para mover elementos compartilhados

RT4 - Mover um sub-elemento compartilhado	
 **Modelo Original**	 **Modelo Final**
Pré-condição: Há um elemento (Tarefa 3) que está compartilhado por diferentes sub-grafos (através de decomposição-tarefa, contribuição ou ligação means-end) e ao menos um destes sub-grafos é movido para um novo ator (regras RT1 e RT2).	
Efeitos: O elemento compartilhado permanecerá no sub-grafo que o elemento raiz tem a maior prioridade. Os relacionamentos com os elementos restantes devem ser substituídos por dependências, como definido nas regras RT1, RT2 e RT3.	

Similarmente, a tarefa *Consultar Base de Dados* relacionada através de um relacionamento *means-end* com o *goal Pesquisa por Item a ser Manipulada*, pode ser movido do Medi@ porque esta tarefa pode ser reusada em outros domínios.

Para selecionar as regras de transformação adequadas a esses elementos, necessitamos observar o tipo de relacionamento que esses elementos têm com os demais elementos. Por exemplo, RT1 pode ser aplicada à tarefa *Produzir Estatísticas* que é uma sub-elemento da tarefa *Gerenciar Loja pela Internet*, em um relacionamento decomposição-tarefa. Mais tarde, RT3 é aplicada para substituir os relacionamentos de contribuição através das fronteiras dos atores pelas dependências de softgoals, para manter a coerência com a notação i*.

Para o sub-grafo no qual a tarefa *Consultar Base de Dados* é a raiz, RT2 é aplicada para mover os sub-elementos que são alternativas (tarefa *Consultar Base de Dados*) para alcançar o *goal* (*Pesquisa por item ser Manipulada*). Posteriormente, RT4 é aplicada para mover elementos compartilhados (os *goals Itens a serem Selecionados* e *Item a ser Transacionado* são, também, sub-elementos da tarefa *Consultar Catálogo*). A Figura 3 mostra o modelo final após a aplicação do processo proposto.

Após ter aplicado o processo ao modelo do SR de Medi@, o modelo final foi avaliado e comparado com o modelo original levando em consideração os atributos da modularidade. As métricas usadas para executar esta avaliação são as métricas da modularidade propostas em [14] apresentadas na seção 3.1. A adaptação das métricas de modularidade introduzidas por [14] depende de nossa definição dos conceitos a serem usados. Neste artigo, os conceitos de componente, operação e LOC (linhas de código) são mapeados aos elementos presentes na linguagem do i*, como:

- Os atores são mapeados para componentes, os atores são elementos de modelagem que o i* usa para conter outros elementos, dessa forma o mapeamento para componentes é natural;
- Cada elemento intencional (por exemplo, tarefas, goals, recursos e softgoals) é mapeado para uma operação;
- todo elemento gráfico, isto é, elementos intencionais, atores e relacionamentos, é traçado a um LOC. Esse mapeamento é mais abrangente que as anteriores pois abstrai as diferenças entre os conceitos base do i*, assim qualquer elemento inclusive atores e ligações são considerados como LOC.

Baseado nestes mapeamentos, as seguintes métricas foram usadas para avaliar nossa proposta: (i) Difusão de Interesses sobre Operações (CDO, Concern Diffusion over Operation), (ii) Difusão de Interesses sobre LOC (CDLOC, Concern Diffusion over LOC); (iii) Acoplamento entre Componentes (CBC, Coupling between Components); (iv) Falta da Coesão entre Operações (LCOO, Lack of Cohesion in Operations). A métrica CDO (i) conta o número de elementos intencionais cuja finalidade principal é contribuir à realiza-

ção de um interesse e o número de outros elementos intencionais que os exige. Enquanto, CDLOC (ii) conta o número de pontos de transição para cada interesse no refinamento do ator (e.g., SR do ator). Os pontos de transição são pontos no refinamento onde há uma mudança (*switch*) de interesses. CBC (iii) conta o número de outros atores que um ator específico é acoplado. Finalmente, LCOO (iv) mede a falta da coesão de um ator nos termos da quantidade de elementos intencionais que não contribuem para o interesse principal.

Tabela 6. Resultados da Avaliação

Métrica	Modelo i*	Modelo i* Final
CDO (Medi@)	8	8
CDLOC (Medi@)	4	1
CBC (Medi@)	0	3
LCOO (Medi@)	25	5

Para medir o CDO do ator Medi@, consideramos como o maior interesse do ator Medi@ o sub-grafo do *goal* Pedidos da Internet a serem Manipulados. A contagem dessa métrica considera os elementos que estão ligados direta ou indiretamente a esse sub-grafo, assim a tarefa *Gerenciar Carro de Compra* e todos os sub-elementos dela são contados. Uma vez que esse sub-grafo está presente no modelo original e no final, o número de elementos intencionais que contribuem para o principal interesse e o número de elementos intencionais que o exige são invariáveis. Assim, o número de elementos intencionais que contribuem para o principal interesse é o mesmo para ambos.

Devido o fato de que a responsabilidade de diversos interesses foram delegadas a atores novos, o número de elementos gráficos que representam pontos de transição entre interesses diferentes no ator Medi@ reduziu de 4 a 1 (CDLOC). Neste caso, a tarefa *Gerenciar Loja pela Internet* é o único ponto de intersecção entre interesses que se mantêm nos dois modelos. Assim, a métrica CDLOC demonstra que nossa proposta promoveu uma redução da difusão dos interesses sobre o ator Medi@, porque muitos interesses foram movidos do ator Medi@ para novos atores.

A métrica para acoplamento (CBC) tem uma diferença significativa entre os dois modelos. No modelo i* original, o ator Medi@ não está acoplado a nenhum outro ator que representa o sistema do comércio eletrônico (CBC igual a 0). Por outro lado, no modelo i* final, o ator Medi@ está relacionado com os relacionamentos da dependência de outros três atores do sistema (CBC igual a 3). LCOO igualmente apresentou uma diferença significativa entre ambos os modelos. Contudo considerando o maior interesse do ator Medi@ como o sub-grafo da goal *Pedidos pela Internet a serem Manipulados*, no modelo i* original, o ator Medi@ têm muitos elementos intencionais que não contribuem ao seu maior interesse (LCOO igual a 25). Na versão final do modelo i*, muitos elementos intencionais relativos a outros interesses foram movidos para os novos atores do software, permanecendo somente alguns elementos intencionais que não contribuem para o maior interesse do ator Medi@ (LCOO igual a 5). Este resultado mostra que no último caso os elementos intencionais do ator Medi@ contribuem para menos interesses do que no ator Medi@ do original. Assim, a separação de interesses melhorou no modelo i* modularizado quando comparada com o modelo original. Este medidor mostrou de uma maneira mais expressiva os benefícios de modularização do ator Medi@ usando nossa abordagem. A Tabela 6 apresenta os valores das métricas de modularidade aplicadas neste trabalho.

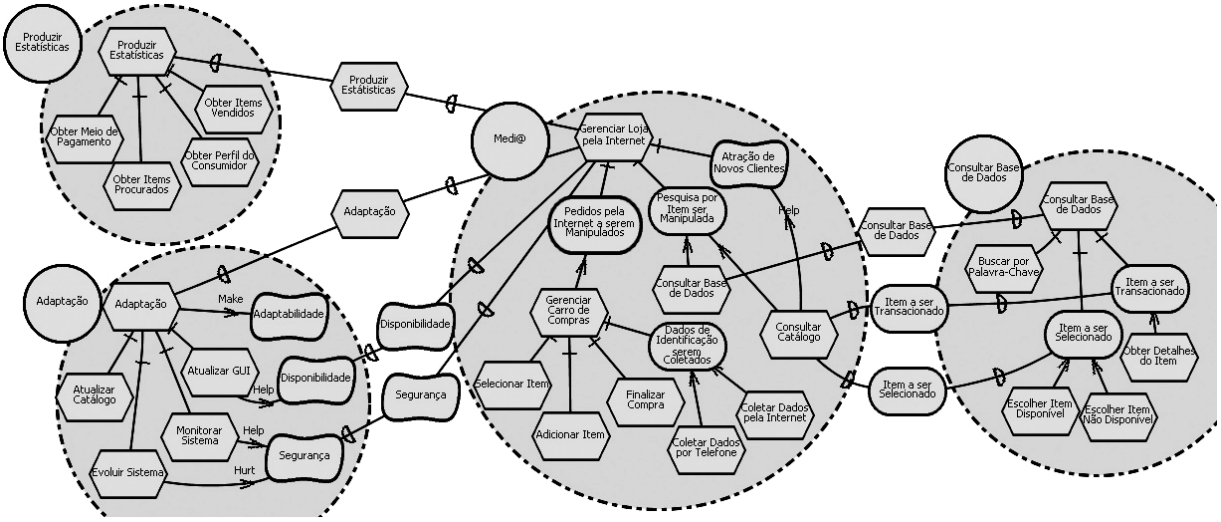


Figura 3. Modelo SR Final

5. Trabalhos Relacionados

Em [17] é apresentado um método sistemático para lidar com questões de escalabilidade em modelos i*. O método reformula o framework i* para proporcionar o conceito de *view* - uma projeção sobre um modelo de acordo com alguns critérios. *Views* foram usados como uma forma de dividir um modelo base em segmentos auto-contidos a fim de aumentar a compreensibilidade dos modelos i*. Cada *view* é associado com uma regra de seleção formalmente definida para permitir automatizar a projeção de um determinado ponto de vista.

Em [1], os autores utilizam os princípios do Desenvolvimento de Software Orientado a Aspectos - AOSD (do inglês, Aspect-Oriented Software Development) [13], para simplificar modelos i*. Essa abordagem identifica, modulariza e compõe interesses transversais em modelos i*. Eles estenderam a linguagem de modelagem i* pela adição de abstrações da AOSD. O objetivo era reduzir a complexidade do gráfico de modelos i*.

Em [16] foi realizado um estudo exploratório para identificar o impacto da aplicação de um catálogo de padrões para modificar modelos i*. Novos conceitos, que são parte de um catálogo de padrões, foram adicionados para alterar o modelo. Embora eles apresentem ganhos em outros atributos dos modelos, nenhuma melhoria na modularidade foi observada. Além disso, nossa abordagem propõe alterações iterativas nos modelos i*, mas não exige a adição de novos conceitos porque usa mecanismos de delegação e regras transformação para dividir o sistema ator em novos atores. Para escolher os elementos que serão delegados, usamos um processo semi-automático.

Tropos [4] se baseia em modelos i* em várias fases do ciclo de vida do software e utiliza a relação *is-part-of* para decompor atores de sistema. Mas, diferente da nossa abordagem, nenhuma forma sistemática foi proposta para a sua utilização.

6. Conclusões

Um processo para melhorar a modularidade dos modelos i* foi apresentado neste artigo. Esse processo propõe balancear as responsabilidades de um ator software, delegando essas responsabilidades a outros (ou novos) atores que constituem o software. Um processo semi-automático pode guiar a avaliação da modularidade dos modelos i*; o uso de heurísticas para escolher qual parte do ator software pode ser delegada a outro ator (software); e, a seleção entre um conjunto de regras de transformações para modificar os modelos i*. Essas regras criam novos atores, move partes de um ator software para esses novos atores, e assegura que o modelo final é equivalente ao modelo original. Neste trabalho utilizamos métricas para avaliar modularidade a modelos i* baseada nas métricas propostas por [14] e adicionamos restrições em OCL para garantir que os modelos i* finais estejam de acordo com a sintaxe i*.

Atualmente, estamos implementando as regras de transformação em uma linguagem apropriada e adicionando na ferramenta IStarTool [3]. Esta ferramenta foi implementada em java, usando ambiente Eclipse e usando a filosofia de desenvolvimento baseado em

modelos. Além disso, as restrições OCL apresentadas em [9] foram implementadas nesta ferramenta, o que facilita a adição das novas expressões OCL apresentadas neste artigo. Também como trabalho futuro pretendemos demonstrar a equivalência entre os modelos original e final apresentados nas regras e utilizar métricas formais proposta em [7] para avaliar os modelos gerados por nossa abordagem. Outro trabalho futuro inclui projeto da arquitetura a partir dos modelos i* gerados por esta abordagem.

7. Referências

[1] F. Alencar, et al., "Integration of Aspects with i* Models", Agent-Oriented Information Systems IV, LNCS, Vol. 4898, Springer-Verlag, pp. 183-201, 2008.

[2] P. Bresciani et al., "Tropos: An Agent-Oriented Software Development Methodology", Journal of Autonomous Agents and Multi-Agent Systems, 8(3), pp. 203-236, 2004.

[3] Campos, B. IStar Tool - Uma proposta de ferramenta para modelagem de i*. Master's thesis, Centro de Informatica - Universidade Federal de Pernambuco, 2008.

[4] J. Castro, et al., "Towards Requirements-Driven Information Systems Engineering: The Tropos Project", Information Systems Journal, 27(6), pp. 365-389, 2002.

[5] K. Czarnecki, S. Helsen, "Classification of Model Transformation Approaches", 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, Anaheim, CA, USA, 2003.

[6] Estrada, H., Rebollar, A. M., Pastor, O., Mylopoulos, J. An Empirical Evaluation of the i* Framework in a Model-Based Software Generation Environment. In CAiSE'06. LNCS 4001, Springer-Verlag , 513-527. 2006.

[7] Franch, X., "On Quantitative Analysis of Agent-Oriented Models". Procs. CAiSE´06, LNCS 4001, 2006.

[8] Grau, G. et al., 2008. i* Wiki Home, In http://istar.rwth-aachen.de/, 02/04/09.

[9] Horkoff, J., Using i* modeling for evaluation, Master thesis, Department of Computer Science, University of Toronto, Canada, 2007.

[10] Lucena, M., Santos, E., Silva, C., Alencar, F. Silva, M. J., Castro, J. Towards a Unified Metamodel for i*. In: Second IEEE International Conference on Research Challenges in Information Science 2008, Marrakech, 2008.

[11] Lucena, M., Silva, C., Santos, E., Alencar, F., Castro, J. Applying Transformation Rules to Improve i* Models. In: The 21th International Conference on Software Engineering and Knowledge Engineering (to appear) San Francisco Bay USA, 2009.

[12] Pohl, K., et al.. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer. 2005.

[13] Rashid, A. Moreira, J. Araujo, "Modularisation and Composition of Aspectual Requirements", 2nd Intl. Conf. on Aspect-Oriented Software Development, ACM Press, 2003, pp. 11–20.

[14] Sant'Anna C. N. et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. Proc. of SBES'03, Brazil, SBC, 19–34. 2003.

[15] Sousa, G. et al. Adapting the NFR framework to aspect-oriented requirements engineering. In SBES, Brazil, 2003.

[16] M. Strohmaier et al., "Can Patterns Improve i* Modeling? Two Exploratory Studies", REFSQ'08, LNCS, Vol. 5025, Springer-Verlag, France, 2008, pp. 153-167.

[17] You, Z., "Using Meta-Model-Driven Views to Address Scalability in i* Models", Master thesis, Department of Computer Science, University of Toronto, Canada, 2004.

[18] E. Yu, J. Castro, A. Perini, "Strategic Actors Modeling with i*", Tutorial Notes, 16th Intl. Conf. on Requirements Engineering, IEEE Computer Society, Spain, pp.01-60, 2008.

[19] Yu, E., "Modeling Strategic Relationships for Process Reengineering", Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.