César Henrique Bernabé

Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução e desenvolvimento da ferramenta CASE Unagi

Vitória, ES

2017

César Henrique Bernabé

Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução e desenvolvimento da ferramenta CASE Unagi

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Departamento de Informática

Orientador: Prof. Dr. Vítor E. Silva Souza

Vitória, ES

2017

César Henrique Bernabé
Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução e desenvolvimento da ferramenta CASE Unagi/ César Henrique Bernabé. – Vitória, ES, 2017-
60 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Vítor E. Silva Souza

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Departamento de Informática, 2017.

1. Engenharia de Requisitos Orientada a Objetivos. 2. Zanshin. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução e desenvolvimento da ferramenta CASE Unagi

CDU 02:141:005.7

César Henrique Bernabé

Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução e desenvolvimento da ferramenta CASE Unagi

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Vitória, ES
2017

Agradecimentos

Primeiramente agradeço a toda minha família pelo apoio que me deram desde quando iniciei essa jornada a 5 anos atrás, agradeço principalmente meus pais, Clovis e Fátima, e a minha irmã Maísa, que fizeram tudo que podiam para me apoiar a trilhar esse longo caminho.

Agradeço aos meus amigos de curso e agora amigos para o resto da vida, que sempre ajudaram a segurar a barra nos momentos difíceis e sempre remaram junto a mim nesse barco - Lucas, Luiz Mai, Celso, Luana, João, Luiz Couto e Silas, muito obrigado. Assim como também não posso esquecer meus amigos de fora da faculdade, mas que sempre estiveram comigo nos momentos bons e ruins e sei que vou poder contar com vocês para todos os outros momentos que vierem - Euge, Shu, Fellipe, Gabriela, Ramon, Gustavo e Nikolas, amo vocês.

Um obrigado mais que especial ao meu orientador, Vítor. Você me trouxe lições e conhecimento que eu nunca imaginei adquirir, agradeço também pela paciência, disposição e boa vontade, foi um prazer ter um mestre como você. Assim também agradeço aos meus professores, que com muita competência e profissionalismo, me ensinaram o máximo que podiam da melhor forma possível, sou muito grato a vocês. Também não posso esquecer de agradecer aos integrantes do grupo de pesquisa em Engenharia de Requisitos e aos colegas do NEMO, em especial ao Pedro, obrigado por todas as experiências trocadas.

E por fim, quero agradecer a Deus por sempre ouvir minhas preces, por permitir que eu conseguisse ter essas oportunidades e por ter colocado todas essas pessoas maravilhosas na minha vida.

"Não importa quem você é,
não importa o que você fez,
não importa de onde você veio,
você sempre pode mudar,
transformar-se em uma versão melhor de si mesmo".
(Madonna)

Resumo

Com o avanço da tecnologia, sistemas de software são executados em ambientes cada vez mais dinâmicos, sendo assim obrigados a lidar com requisitos cada vez mais complexos. Nesse universo tecnológico, onde sistemas precisam reagir a diversos tipos de condições destaca-se o uso de sistemas que modificam seu comportamento e performance em tempo de execução para satisfazer requisitos mesmo em caso de falha, podendo se replanejar e reconfigurar à medida que novas exigências são encontradas e precisam ser atendidas. *Zanshin* (SOUZA, 2012) é uma abordagem baseada em Engenharia de Requisitos Orientada a Objetivos (*Goal-Oriented Requirements Engineering* ou simplesmente GORE) criada para apoiar o desenvolvimento de sistemas adaptativos. Baseado no fato de que todo sistema possui, ainda que implícito, um ciclo de retroalimentação, o *framework* utiliza dessa premissa para monitorar e avaliar o comportamento do sistema alvo, enviando assim instruções de adaptação para o mesmo. O processo de monitoramento do *Zanshin* é baseado em um metamodelo enriquecido de elementos focados na estratégias de adaptação, ambos elementos e modelos foram desenvolvidos especificamente para esse fim.

Este trabalho discute uma nova proposta para o metamodelo operacional do *Zanshin*, que foi reformulado buscando compreender as restrições de *GORE* de uma forma mais precisa do que na adotada anteriormente. Além disso, para apoiar o processo de modelagem dos requisitos dentro desta abordagem, é apresentada a ferramenta CASE Unagi, que implementa um editor gráfico para modelos de sistemas adaptativos e oferece um sistema de integração com o *Zanshin*, permitindo assim que modelos criados sejam importados diretamente para a plataforma de apoio a sistemas adaptativos.

Palavras-chaves: Engenharia de Requisitos, GORE, Sistemas Adaptativos, Metamodelo, Desenvolvimento Dirigido por Modelos, MDD, Zanshin, Unagi.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	eXtensible Markup Language
GORE	Goal-Oriented Requirements Engineering
GORO	Goal-Oriented Requirements Ontology
EMF	Eclipse Modeling Framework
MDD	Model-Driven Development

Sumário

1 Introdução

O avanço da tecnologia nas ultimas décadas permitiu que a complexidade das atividades realizadas por computadores se tornasse cada vez maior, demandando que projetos de sistemas de software passassem a abranger ainda mais os detalhes de domínio do ambiente em que os programas computacionais seriam executados (ANDERSSON et al., 2009; BRUN et al., 2009). Esse fator motivou estudos na área de modelagem e projeto de sistemas, fazendo com que novas pesquisas buscassem abranger os processos de projeto, construção e teste de software.

Entretanto, para garantir a estabilidade dos sistemas, fazia-se uso majoritariamente da intervenção humana, o que rapidamente tornou-se inviável à medida que os sistemas cresciam para atender o aumento da demanda de novos usuários (ANDERSSON et al., 2009). Assim, a utilização de sistemas adaptativos vem tornando-se a solução mais viável e prática para a atual conjuntura do desenvolvimento de softwares. Além disso, o aumento do número de diferentes dispositivos e situações em que esses softwares podem ser executados faz com que eles passem a enfrentar uma grande diversidade de contextos (muitas vezes imprevisíveis) de execução, fundamentando ainda mais a pesquisa na área de softwares adaptativos (KEPHART; CHESS, 2003).

Sistemas adaptativos são dotados da capacidade de tomar decisões para se ajustar e se reconfigurar mediante mudanças de contexto, permitindo assim que os requisitos elicitados continuem a ser atendidos de forma satisfatória (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Entretanto, poucas soluções desse tipo consideram a modelagem das características adaptativas no sistema desde a fase de modelagem do mesmo. *Zanshin* (SOUZA, 2012) aparece como uma abordagem que baseia-se em modelos para projetar características adaptativas em sistemas por meio de novos tipos de requisitos, que definem o chamado "ciclo de retroalimentação", que operacionaliza a adaptação. Seguindo uma abordagem de Desenvolvimento Dirigido por Modelos, *Zanshin* apresenta um metamodelo que permite que modelos de sistemas sejam especificados de acordo com os requisitos do *framework*.

Este trabalho encontra-se no contexto da pesquisa em torno da abordagem *Zanshin*, mais especificamente lidando com duas de suas limitações: (1) seu metamodelo atual não representa fielmente o conjunto de modelos desejados; e (2) não há uma ferramenta CASE que auxilie engenheiros de requisitos no uso da abordagem.

1.1 Objetivos

Este trabalho possui dois objetivos principais: a reconstrução do metamodelo operacional do *Zanshin* e a criação de uma ferramenta gráfica usada para modelar sistemas adaptativos usando Engenharia de Requisitos Orientada a Objetivos (*Goal-Oriented Requirements Engineering* ou *GORE*). O primeiro refere-se ao metamodelo de objetivos que é baseado em *CORE* (*Core Ontology for Requirements Engineering*) (JURETA et al., 2007), usado pelo framework para casar os requisitos do modelo de domínio específico com as instâncias dos elementos do metamodelo de *GORE*, e assim garantir que os objetivos do sistema estão sendo atendidos satisfatoriamente. Já a segunda atividade refere-se à ferramenta de modelagem de sistemas baseada nesse metamodelo operacional do *Zanshin*, usando sintaxe baseada em uma linguagem de especificação de modelos ontológicos conceituais conhecida como iStar (DALPIAZ; FRANCH; HORKOFF, 2016). Para ambos os casos, foram utilizados os conceitos aprendidos ao longo do curso de Ciência da Computação. Dessa forma são objetivos específicos deste projeto:

- Levantar as deficiências do metamodelo atual do *Zanshin*, identificando os pontos em que as relações entre os elementos deveriam ser modificadas para representarem mais fielmente as formalidades da Engenharia de Requisitos Orientada a Objetivos;
- Elaborar um metamodelo atualizado que, além de representar mais rigorosamente a hierarquia dos elementos *GORE*, também reflita as necessidades da arquitetura do *Zanshin*, como por exemplo as relações entre elementos;
- Modificar o código fonte do *framework* para que o mesmo possa utilizar o novo metamodelo desenvolvido e executar o mecanismo de adaptações considerando esse novo metamodelo;
- Desenvolver uma ferramenta que permita ao usuário criar uma representação gráfica do modelo do sistema alvo e implementar, dentro dessa ferramenta, um módulo para converter o modelo gráfico representado para arquivos XML que possam ser importados diretamente para o *Zanshin*;
- Apresentar os trabalhos desenvolvidos, juntamente com as perspectivas futuras de otimização de ambos os sistemas apresentados.

1.2 Metodologia

O trabalho realizado compreendeu as seguintes atividades:

1. *Revisão Bibliográfica*: Estudo sobre Engenharia de Requisitos Orientada a Objetivos,

Desenvolvimento Dirigido por Modelos e suas ferramentas, de publicações acadêmicas sobre *Zanshin* e sobre desenvolvimento de sistemas adaptativos;

2. *Estudo das Tecnologias:* Levantamento das tecnologias disponíveis para *Eclipse Modeling Framework* (EMF) que permitam o desenvolvimento de editores gráficos dentro da plataforma Eclipse™, de tecnologias que podem ser utilizadas para o desenvolvimento de editores gráficos e do código fonte do *Zanshin* (que também foi desenvolvido nessa plataforma);

3. *Elaboração do novo Metamodelo*: Nessa etapa o novo metamodelo a ser usado foi elaborado gradativamente a partir de informações obtidas dos documentos estudados e das discussões realizadas em reuniões com grupo de estudos de Engenharia de Requisitos na UFES;

4. *Adequação do Zanshin ao novo Metamodelo*: Após finalização do metamodelo, inicou-se processo de adequação do *framework* para que o mesmo pudesse operar de acordo com a nova proposta, consistindo da modificação do código-fonte do *Zanshin*, bem como realização de testes de validação para garantir a consistência do novo metamodelo;

5. *Implementação da Ferramenta de Modelagem*: Uma primeira versão da ferramenta foi desenvolvida no contexto de trabalho de Iniciação Científica, entretanto a mesma usava o metamodelo antigo do *Zanshin*. Essa etapa consiste, portanto, no refatoramento da ferramenta gráfica que permite a modelagem de sistemas adaptativos seguindo as formalidades do novo metamodelo proposto para o sistema *Zanshin*, bem como a criação de módulo usando linguagem de transformação de modelos para texto, que permite a exportação do modelo desenvolvido nessa ferramenta para arquivo XML adequado aos padrões do *framework*;

6. *Redação da Monografia:* Escrita da monografia, etapa obrigatória do processo de elaboração do Projeto de Graduação. Para a escrita desta, foi utilizada a linguagem *LaTeX*[1] utilizando o template *abnTeX*[2] que atende os requisitos das normas da ABNT (Associação Brasileira de Normas Técnicas) para elaboração de documentos técnicos e científicos brasileiros. Para apoiar este processo, foi utilizado o aplicativo *TeXstudio*.[3]

[1] LaTeX – http://www.latex-project.org/
[2] abnTeX – http://www.abntex.net.br
[3] www.texstudio.org

1.3 Organização do Texto

Este texto está dividido em quatro partes principais além desta introdução, que seguem:

- **Capítulo 2** – Referencial Teórico: apresenta discussão acerca de *GORE* e *Model-Driven Development*, focando na relação desses tópicos com sistemas adaptativos e com o processo de desenvolvimento da ferramenta *Unagi*. Ademais, é discutida a arquitetura do sistema *Zanshin* e suas características;
- **Capítulo 3** – Zanshin: nesse capitulo são apresentados os processos e decisões que levaram à elaboração do novo metamodelo do *Zanshin*, bem como as modificações decorrentes dessas modificações na arquitetura da plataforma;
- **Capítulo 4** – Unagi: Nesse capítulo é abordado o processo de desenvolvimento da ferramenta *Unagi* e os pormenores da implementação de todos os módulos da mesma;
- **Capítulo 5** – Revisão: é feita validação do metamodelo evoluído do *Zanshin* e da implementação da ferramenta *Unagi*;
- **Capítulo 6** – Considerações Finais: apresenta as conclusões obtidas ao final deste trabalho, bem como as dificuldades encontradas e as perspectivas de trabalhos futuros para esse contexto.

2 Referencial Teórico

Este capítulo apresenta os principais conceitos teóricos que alicerçaram a evolução do metamodelo de requisitos do *Zanshin* e de conceitos que fundamentaram o desenvolvimento da ferramenta *Unagi*. A Seção 2.1 apresenta um breve resumo sobre Desenvolvimento Dirigido por Modelos (*Model-Driven Development* ou MDD) assim como as principais ferramentas que foram utilizadas durante o desenvolvimento do *Unagi*, como as funcionalidades EMF de modelagem do Eclipse™, o plugin Sirius™, dentre outros. A Seção 2.2 fala sobre Engenharia de Requisitos Orientada a Objetivos, destacando os principais conceitos dessa área que foram utilizados ao longo deste trabalho. A Seção 2.3 apresenta o sistema *Zanshin* e os detalhes do metamodelo original do *framework*.

2.1 Desenvolvimento Dirigido por Modelos

Pesquisadores vêm tentando ao longo dos anos criar abstrações que ajudem programadores a focar no conteúdo do desenvolvimento ao invés das especifidades da tecnologia de criação adotada (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). O Desenvolvimento Dirigido por Modelos (*Model-Driven Development* ou MDD) pode ser visto como a forma de programação de mais alto nível de abstração existente atualmente (ATKINSON; KUHNE, 2003), promovendo o uso de artefatos do processo de desenvolvimento de software para lidar com complexidade por meio de abstração (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014).

Em outras palavras, MDD parte da premissa que um sistema é um modelo consistente com seu metamodelo (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014) e, assim, em vez de exigir que programadores escrevam cada simples detalhe da implementação de um sistema, permite que uma funcionalidade necessária para um software possa ser visualmente modelada (ATKINSON; KUHNE, 2003). Portanto, essa técnica viabiliza que muitas atividades complexas (porém rotineiras) da área de programação de software sejam automatizadas como por exemplo, o suporte a persistência, interoperabilidade e distribuição (ATKINSON; KUHNE, 2003).

A modelagem usa da percepção visual humana para melhorar o processo de compreensão sobre o domínio de um software, já que modelos nos auxiliam a entender problemas complexos e suas possíveis soluções por meio da abstração. Assim, MDD baseia-se na premissa de que o desenvolvimento de software deve focar principalmente na produção de modelos e não na criação de código (SELIC, 2003).

A primeira vantagem dessa abordagem é que podemos usar conceitos mais ligados

ao domínio do problema que o software vai resolver, ao invés de focar em conceitos técnicos ligados a linguagem de programação, por exemplo. Essa vantagem acarreta em alguns outros benefícios: modelos são mais compreensíveis do que códigos e portanto, tornam-se também mais fáceis de especificar e manter (SELIC, 2003). Além disso, modelos são menos sensíveis a alterações de tecnologias, ou seja, são independentes de plataforma (SELIC, 2003).

Esse trabalho usa MDD em vários processos: primeiramente podemos considerar que *Zanshin* usa das ferramentas desse método para instanciação, validação e outras atividades relacionadas ao modelo do sistema que é fornecido pelo usuário. Segundo, *Unagi* usa a produção automática de código por meio da interpretação de modelos visuais (SELIC, 2003; VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014), além do que o processo de modelagem de sistemas e a geração automática dos arquivos XML para *Zanshin* também pode ser vista como um processo de *Model-Driven Development*.

2.1.1 Eclipse Modeling Framework

O Eclipse™ é um projeto de código aberto com o objetivo de prover uma plataforma de desenvolvimento altamente integrada. O processo de criação de sistemas no Eclipse™ pode ser dividido em alguns projetos, entre eles o Projeto de Modelagem (*Modeling Project*), que concentra-se em tecnologias baseadas no desenvolvimento orientado a modelos (STEINBERG et al., 2008). Esse ambiente é chamado de *Eclipse Modeling Framework* ou *EMF*, e provê funcionalidades como transformação de modelos, integração de bases de dados e geração de editores gráficos (STEINBERG et al., 2008). Modelos especificados dentro do contexto EMF relacionam abstrações de modelagem diretamente a seus conceitos de implementação, sendo a união das tecnologias UML, XML e Java™, permitindo a conversão automática entre todas essas (STEINBERG et al., 2008).

A tecnologia EMF pode ser melhor explicada com um exemplo: um sistema de gerenciamento de ordem de compras de uma loja, que necessita incluir casos como "cobrar" e "entregar" em um endereço e uma coleção de itens (nesse caso, compras). A Figura 1 mostra o diagrama do sistema em UML. Esse modelo pode também ser descrito dentro do EMF usando modelos *Ecore*, como na Figura 2. Modelos *Ecore* são basedos no metamodelo para especificação exibido na Figura 3, onde classes são representadas por `EClass`, atributos por `EAttribute`, relações por `EReference` e tipos de dados por `EDataType`. Assim, a "conversão" do modelo UML para *Ecore* é dada ao se instanciar classes de *Ecore* de acordo com a especificidade do domínio do problema (STEINBERG et al., 2008), como pode ser visto na Figura 4.

Assim que o modelo é detalhado em *Ecore*, o EMF está pronto para gerar código automaticamente, por meio dos seguintes passos:

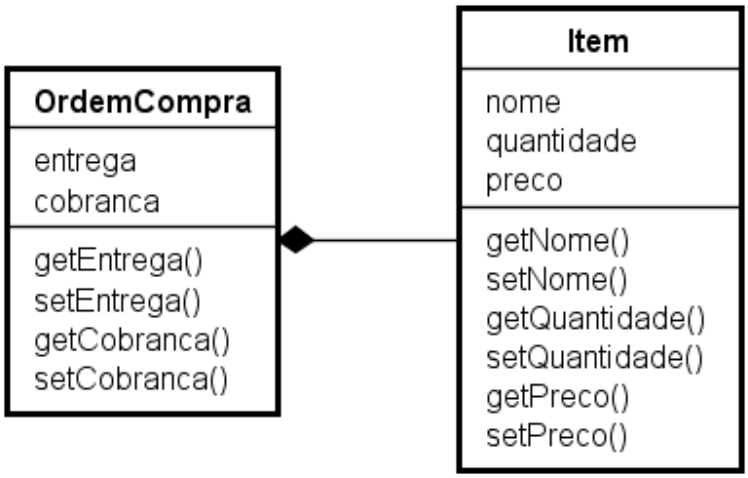

Figura 1 – Classes do diagrama do exemplo em UML.

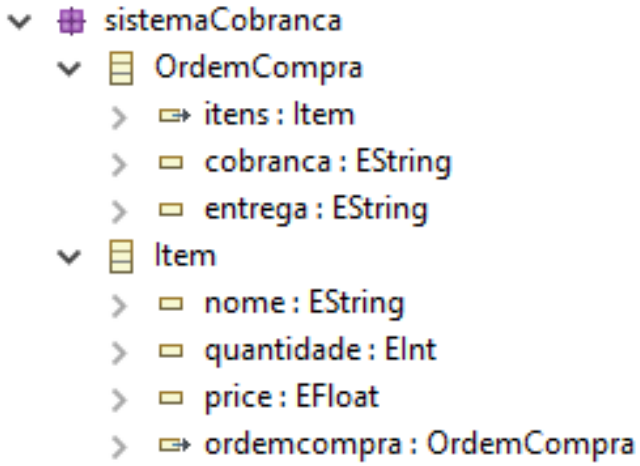

Figura 2 – Classes do diagrama do exemplo em *Ecore*.

- Para cada tipo `EClass` cria-se uma interface e sua classe de implementação correspondente. No caso do exemplo, para a classe `OrdemCompra` seriam criadas a interface `OrdemCompra` e a classe `OrdemCompraImpl`. Essa especificação permite que sejam implementadas funções de persistência e distribuição, porém não serão discutidas aqui por fugirem do escopo desse trabalho.
- As classes do tipo `EAtributte` são transformadas em atributos nas classes correspondentes
- As classes tipo `EReference` são transformadas em referências nas respectivas classes que referenciam.

Em suma, o *framework* de modelagem do Eclipse™ permite que usuários criem modelos apoiados em metamodelos, e baseando-se no modelo criado, gera partes de código de sistema (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014).

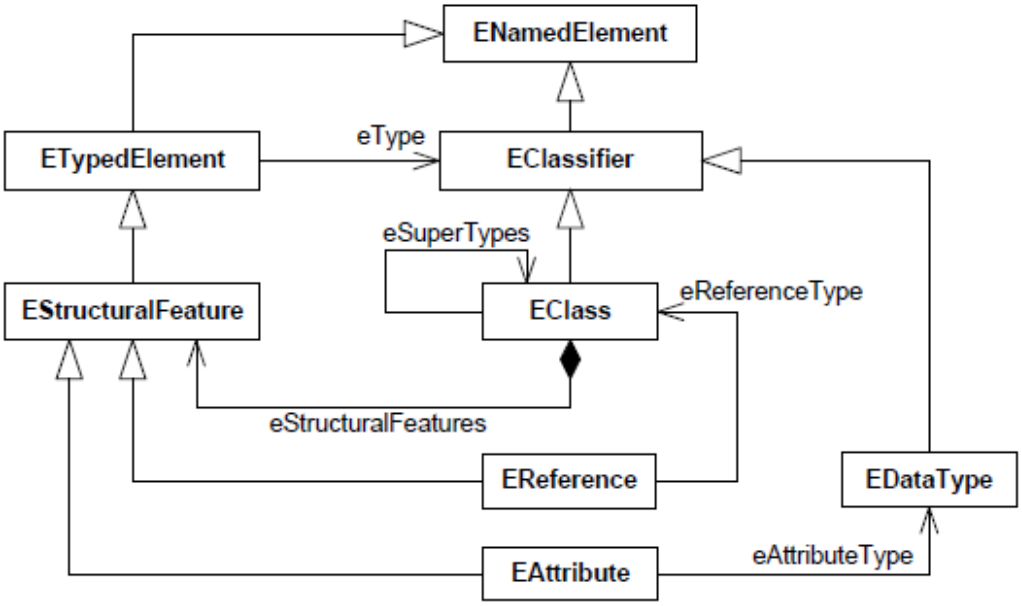

Figura 3 – Metamodelo de *Ecore* (KERN, 2008).

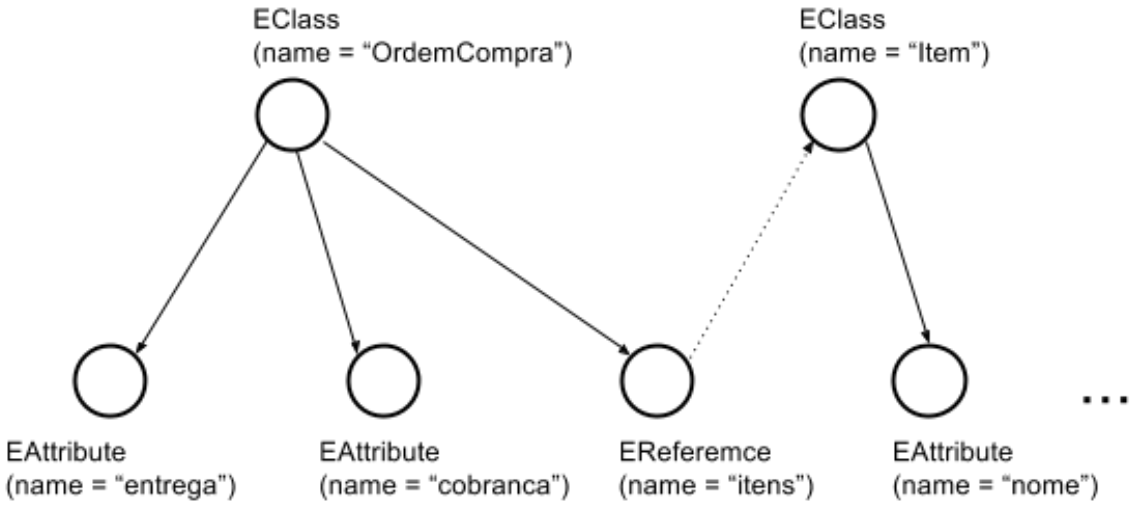

Figura 4 – Conversão do diagrama de exemplo de UML para *Ecore*.

2.1.2 Sirius

O Sirius™ é um plugin que simplifica o *framework* de Modelagem Gráfica (*Graphical Modeling Framework* ou *GMF*) do Eclipse™, reduzindo a complexidade de uso do mesmo e permitindo a produção de editores gráficos de modelos personalizados (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). O Sirius™ é construído em cima das utilidades para (des)serialização de modelos, checagem de condições e geração de editores baseados em *Ecore* providas pelo Eclipse™ (BUDINSKY, 2004). Representações criadas com Sirius™ podem ser apresentadas em diagramas, tabelas e árvores (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Em síntese, Sirius™ provê ferramentas que permitem a especificação de um modelo de um domínio qualquer em diferentes perspectivas gráficas (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014).

Por meio de modelos de especificação (*Viewpoint Specification Model* ou VSM), o *plugin* permite definir a estrutura, aparência e comportamento do metamodelo do editor a ser criado (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Os VSMs são especificados em arquivos *.odesign* (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014), e baseam-se no metamodelo de domínio descrito em *Ecore*. Um exemplo de caracterização de representação é mostrado na Figura 5, que retrata a descrição visual do exemplo do sistema de compras já citado anteriormente. Da mesma forma, um modelo criado usando essa descrição é mostrado na Figura 6. Nesse caso, pode-se ver que o *container* de fundo branco, representado como um quadrado no modelo, contém vários outros *containers* de fundo amarelo, que representam os itens de uma ordem de compra. Essas definições de cor e contenimento estão definidas na Figura 5, onde vê-se que o *container* OrdemCompra possui "*Gradient White to White*" como estilo, por exemplo, assim como é possível perceber que OrdemCompra possui Item como um de seus "filhos".

Sirius™ provê vários mecanismos que auxiliam no gerenciamento da complexidade de modelos (MADIOT; PAGANELLI, 2015):

- **Camadas**: é possível separar elementos em camadas diferentes, e ativá-las ou desativá-las.
- **Filtros**: exibir ou esconder elementos dependendo de determinada condição.
- **Personalização de Estilos**: permite modificar propriedades gráficas de elementos do diagrama.
- **Regras de Validação**: permite que verificar se o modelo está sintaticamente correto.

2.1.3 Acceleo

Acceleo é usada para processos de Transformação de Modelo para Texto (*Model to Text Transformation* ou *M2T*) (MUSSET et al., 2006a) e, apesar de ser uma lin-

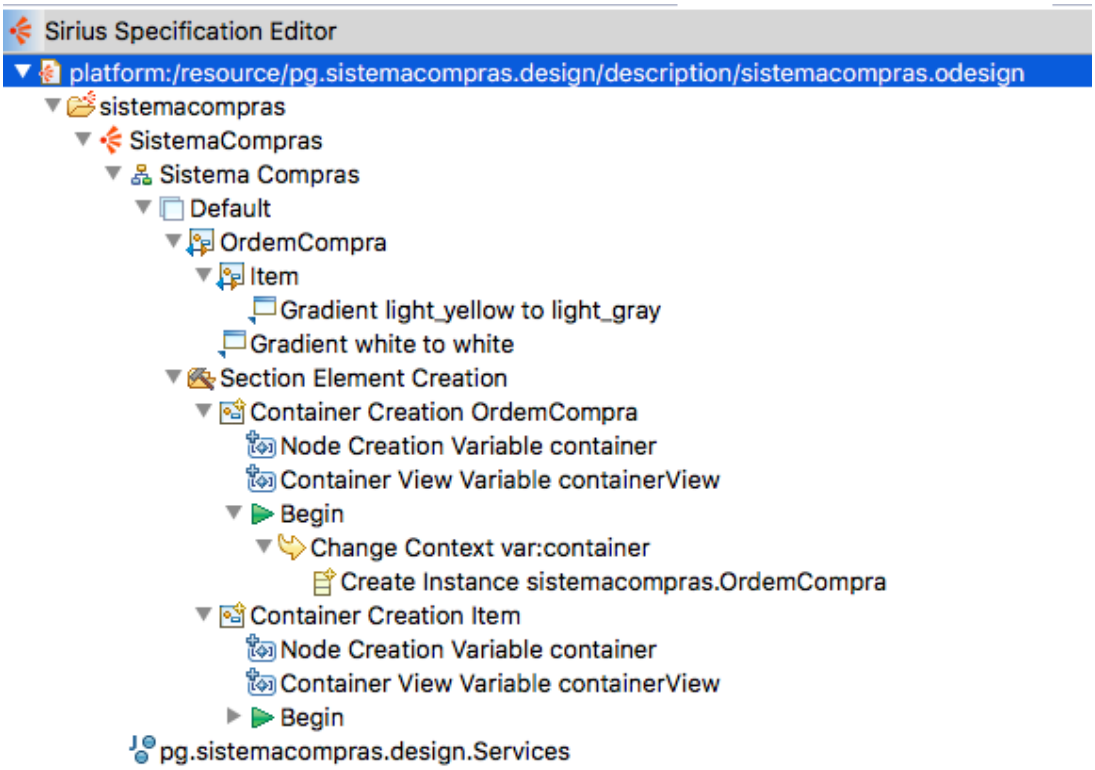

Figura 5 – Exemplo de *Viewpoint Specification Model*.

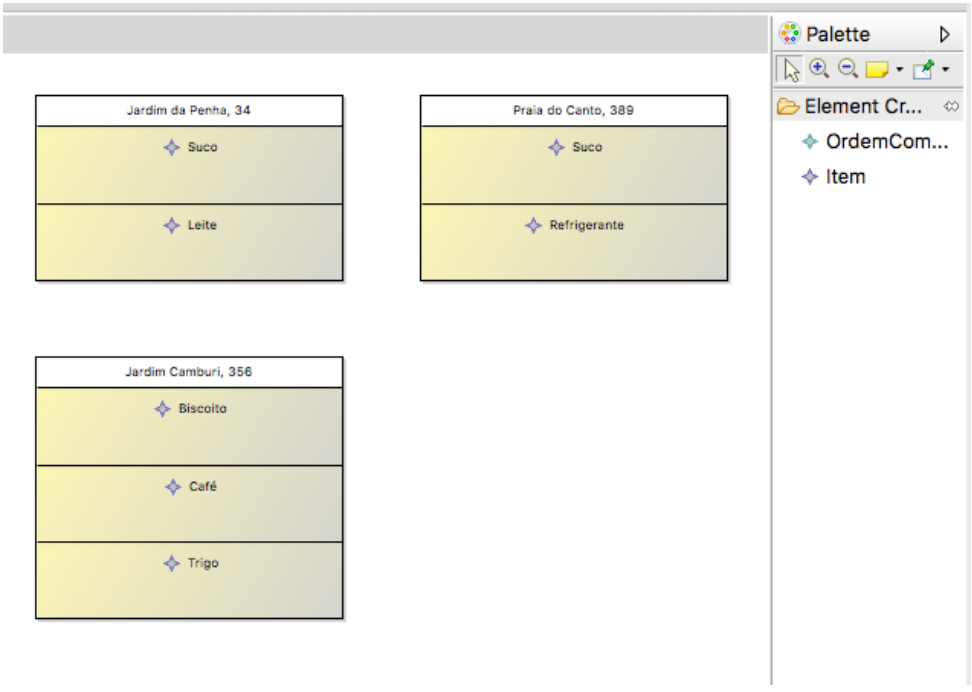

Figura 6 – Exemplo de Modelo do Sistema de Compras criado usando Sirius.

guagem de script, permite integração com linguagens de alto-nível, como por exemplo Java™ (MTSWENI, 2012). Assim também, Acceleo permite a criação de geradores automáticos de código, usando padrões e *templates*. Em adição, a linguagem é compatível com o ambiente Eclipse™ e possui um editor próprio dentro do *framework*, contando com funcionalidades como destaque de sintaxe, detecção de erros e refatoramento (ACCELEO, 2012a).

Criada em 2005, a linguagem é definida pelos autores do projeto como uma implementação pragmática do modelo *MOF Model to Text Transformation Language*, que é uma especificação para linguagens de transformação de modelos para textos criado pelo *Object Management Group* (*OMG*) (ACCELEO, 2012b).

2.2 Engenharia de Requisitos Orientada a Objetivos

A Egenharia de Software é uma área da Ciência da Computação voltada ao estudo dos processos, métodos, técnicas, ferramentas e ambientes de suporte ao desenvolvimento de software, apoiando-se principalmente nas práticas e aplicações da área de Gerência de Projetos com o objetivo de promover melhor organização, produtividade e qualidade em todo o processo de desenvolvimento de um software (FALBO, 2014).

Dentro da área de Engenharia de Software, destaca-se uma importante subárea, a área de Egenharia de Requisitos de Software, focada no processo de elicitação de requisitos, considerados fatores determinantes no sucesso do desenvolvimento de um software (FALBO, 2017). Requisitos podem ser entendidos como a definição do que o sistema pode prover, ou também entendidos como o que o sistema é capaz de fazer para atingir um determinado objetivo (PFLEEGER, 2004).

Requisitos estão diretamente ligados aos objetivos do sistema, portanto destaca-se também a Engenharia de Requisitos Orientada a Objetivos, uma subárea da Engenharia de Requisitos. Objetivos são parte importante do processo de elicitação de requisitos, seu propósito é indicar as principais necessidades que justificam a criação de um determinado sistema, demonstrando os casos em que as funcionalidades do mesmo satisfarão as necessidades elicitadas, além de dizer como o sistema deve ser construído para satisfazê-las (ROSS; SCHOMAN, 1977).

Em uma descrição geral e resumida do processo de identificação de objetivos, pode-se dizer que o potencial software é analisado nos ambientes organizacional, operacional e técnico, onde são identificados os problemas de contexto e as oportunidades de solução desses conflitos. Então, são elaborados com foco na resolução dos pontos identificados e são devidamente refinados. Por fim, requisitos são criados e refinados com o propósito de atenderem a esses objetivos levantados para o sistema.

Além de apoiar no processo de modelagem de requisitos, objetivos são usados para apoiar outros propósitos, como gerenciamento de conflitos e o processo de verificação (LAPOUCHNIAN, 2005). De acordo com Lamsweerde (2001), eles podem ser reformulados em diferentes níveis de abstração dependendo do tipo de necessidade que o sistema alvo deve atender, abrangendo desde interesses referentes a estratégias de negócios até conceitos técnicos de atividades.

A necessidade de uso de objetivos no processo de modelagem de sistemas de software vem se tornando cada vez mais clara à medida que analistas percebem que:

- Objetivos provêm critérios claros de completude dos requisitos do sistema, permitindo também que requisitos desnecessários sejam identificados e descartados (LAMSWEERDE, 2001);
- Objetivos facilitam o processo de entendimento dos requisitos pelas partes interessadas (LAMSWEERDE, 2001);
- O uso de objetivos melhora a legibilidade de documentos de especificação de requisitos, pois permite que engenheiros possam enxergar com mais clareza as alternativas de desenvolvimento do sistema, além de facilitar o processo de gerenciamento de conflitos (LAMSWEERDE, 2001);
- Objetivos dirigem parte do processo de elicitação de requisitos, facilitando a identificação de boa parte deles (LAPOUCHNIAN, 2005).

Diferentemente dos requisitos, objetivos podem precisar da cooperação entre diferentes tipos de refinamentos para que sejam atendidos de forma suficiente (DARDENNE; LAMSWEERDE; FICKAS, 1993). Em outras palavras, um objetivo diretamente relacionado ao sistema a ser criado torna-se um requisito, enquanto um objetivo sob responsabilidade de um agente do ambiente (pode ser um usuário ou outro sistema) em que o software será executado torna-se uma Pressuposição de Domínio (ou *Domain Assumption*) e, nesse caso, são satisfeitos devido a uma regra de negócio (LAMSWEERDE, 2001; LAMSWEERDE; DARIMONT; LETIER, 1998).

Objetivos funcionais podem ser classificados como objetivos rígidos (*Hard Goals*), cujo critério de satisfação pode ser atendido de forma técnica (DARDENNE; LAMSWEERDE; FICKAS, 1993) (ou seja, podem ter seu estado definido, sucesso ou falha, de acordo com certas condições) e objetivos maleáveis (*Soft Goals*), que não possuem critérios claros de satisfação, entretanto são úteis quando deseja-se comparar os melhores refinamentos ao objetivo estudado. Para que *Soft Goals* tenham um parâmetro claro de satisfabilidade, são associados a eles os Critérios de Qualidade (*Quality Constraints*). Por exemplo, um *Soft Goal* “Baixo Custo” pode ser operacionalizado pelo critério de qualidade “Custo deve ser menor que R$ 1 mil”.

Por fim, Jureta, Mylopoulos e Faulkner (2008) definem outro tipo de refinamento para especificar a satisfação de um objetivo: as tarefas (*Taks*), que são os passos a serem tomados para que um determinado objetivo seja cumprido. Em outras palavras, tarefas são definidas por funcionalidades do sistemas que, se executadas com sucesso, são consideradas satisfeitas (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012).

Objetivos relacionam-se uns com os outros por meio de refinamentos. Segundo Dardenne, Fickas e Lamsweerde (1991), também em (DARDENNE; LAMSWEERDE; FICKAS, 1993), objetivos podem ser refinados usando grafos E/OU (*AND/OR*). O critério de satisfabilidade de objetivos refinados em "E" ou "OU" segue os conceitos da lógica booleana: refinamentos do tipo "E" implicam que, para que um objetivo seja considerado satisfeito, todos os sub-objetivos refinados a partir dele devem estar em estado de sucesso, enquanto refinamentos do tipo "OU" relacionam o objetivo principal com um conjunto de alternativas, ou seja, basta que um de seus refinamentos seja atendido para que ele também seja considerado alcançado. Objetivos são refinados até atingirem um nível de granularidade em que são refinados apenas em tarefas, que podem ser completadas com sucesso por um ator (humano ou outro sistema) (SOUZA et al., 2013). No contexto do *Zanshin*, refinamentos podem acontecer entre Objetivos e outros Objetivos, *Softgoals*, Tarefas e Pressuposições de Domínio e por Tarefas em outras Tarefas.

Em questões de representação gráfica, os modelos de objetivos discutidos nesse texto são grafos ordenados que exibem as exigências das partes interessadas no topo do modelo e, abaixo, objetivos e tarefas mais refinados, adotando uma topologia do tipo árvore. A simbologia utilizada é baseada na sintaxe de $i^\star$ (YU et al., 2011), mais recentemente renomeado iStar (DALPIAZ; FRANCH; HORKOFF, 2016). Um exemplo de modelo de objetivos representando um sistema de despacho de ambulâncias é mostrado na Figura 7.

2.2.1 GORO

GORO (*Goal-Oriented Requirements Ontology*) (NEGRI et al., 2017) é uma ontologia de referência baseada no domínio de *GORE* e fundamentada em UFO (*Unified Formal Ontology*) (GUIZZARDI, 2005). Uma ontologia pode ser definida como um modelo de dados que representa um conjunto de conceitos relacionados a um domínio específico e também define o relacionamento entre esses conjuntos (GRUBER, 2009). Em outras palavras, uma ontologia é uma forma de representação do conhecimento sobre um assunto da realidade.

GORO foca principalmente no conceito de objetivos, tendo como principal meta esclarecer e explorar a natureza dos conceitos envolvidos, buscando assim aumentar a eficiência da comunicação entre os *stakeholders* de um projeto.

Essa ontologia foi construída por meio da extração de conceitos de diferentes propos-

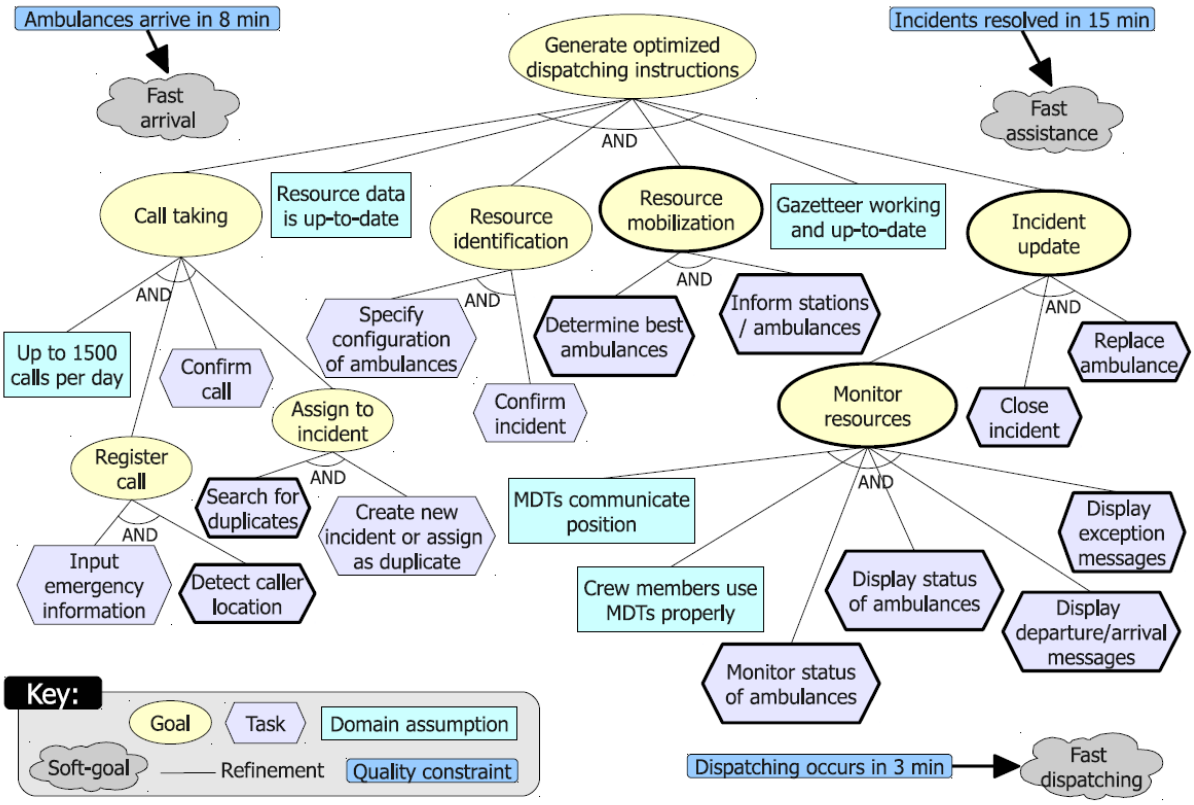

Figura 7 – Exemplo de modelos de objetivos (SOUZA, 2012).

tas GORE, como iStar (DALPIAZ; FRANCH; HORKOFF, 2016), KAOS (DARDENNE; LAMSWEERDE; FICKAS, 1993) e Techne (BORGIDA et al., 2009). Assim, os principais elementos de *GORO* são (NEGRI et al., 2017):

- **Intentions e Desires**: concentram-se na modelagem dos objetivos dos agentes, que são definidos em UFO como algo que existe na realidade e possui uma identidade particular, não tendo partes temporais mas existindo no tempo e existencialmente independente. Agentes possuem propriedades relacionadas a intenção, como Desejos (*Desires*), Crenças (*Beliefs*) e Intenções (*Intentions*). Assim, um objetivo é um conteúdo proposicional relacionado ou à intenção ou ao desejo do agente.

- **Goals e Requirements**: Um requisito é um comportamento desejado, portanto, um requisito é também um objetivo "que descreve as condições ambientais a serem alcançadas por meio de uma solução desejada, resultando na satisfação dos objetivos estratégicos subjacentes".

- **Hardgoals, Softgoals, Requisitos Funcionais e Não-Funcionais**: Adotando a conceituação de (GUIZZARDI et al., 2014), *Hardgoals* são definidos em *GORO* como requisitos que são satisfeitos mediante um dado conjunto de situações, enquanto *Softgoals* referem-se a uma definição vaga de qualidade que só podem ser consideradas satisfeitas mediante julgamento do *stakeholder*. Por fim, Requisitos Funcionais

referem-se a funções que podem manifestar determinado comportamento por meio de determinadas situações, enquanto Requisitos Não-funcionais referem-se a qualidades.

- **Assumptions**: referem-se a pressuposições de domínio das quais as partes interessadas acreditam ser uma verdade no domínio do problema.

- **Decomposition, Alternatives, Operationalization e Plans**: *Decomposition* é, como o próprio nome diz, uma decomposição de um Objetivos em Sub-objetivos, assim, diz que um objetivo é satisfeito quando todos os seus sub-objetivos são satisfeitos. Da mesma forma, diz que um objetivo pode ser atingido de diferentes maneiras, sendo essas as alternativas (*Alternatives*) de satisfação daquele. *Operationalization* é um link que conecta um Objetivo a uma Tarefa, e especifica como aquele objetivo deve ser satisfeito por meio da tarefa. Por fim, *Plans* são descritos como uma forma de fazer alguma coisa ou tarefa.

2.3 Zanshin

Apresentado por Souza (2012), *Zanshin* é um *framework* que utiliza de ciclos de retro-alimentação (*feedback loops*) para monitorar sistemas e enviar estratégias de adaptação com base em informações de modelos de requisitos orientados a objetivos enriquecidos com elementos construído especialmente para guiar o processo de adaptação. O nome *Zanshin* vem do japonês e significa "estado de completa consciência".

Em tempo de execução, os elementos do modelo de objetivos são representados por classes e instanciados cada vez que o usuário (ou sistema) busca atingir um objetivo, e então o sistema passa a enviar mensagens a essas instâncias quando há mudança de estado. Percebe-se, assim, que objetivos e pressuposições de domínio não são tratadas como invariantes que devem sempre ser atingidas, já que o sistema pode falhar ao tentar atingir seus objetivos iniciais e, então, o componente de adaptação lidará com essas falhas e tomará medidas para que os objetivos voltem a ser satisfeitos (SOUZA et al., 2013).

Assim como o monitoramento de objetivos por meio do *Feedback Loop*, o *Zanshin* também monitora Parâmetros (*Parameters*) que podem ser definidos em dois tipos. Primeiro, os Pontos de Variação (ou *Variation Points*), que representam refinamentos do tipo "OU" (*OR*). Segundo, as Variáveis de Controle (ou *Control Variables*), que são abstrações referentes a pontos de variação repetitivos ou usados em grande escala. A especificação do comportamento dessas estratégias é discutido em seções posteriores.

O metamodelo dos modelos de objetivos usados no *Zanshin* é representado na Figura 8 e especificado no código do *framework* por meio de uma tecnologia *Ecore* para descrição e suporte a modelos em tempo de execução. Nessa caso, o modelo representa os tipos de requisitos em nível de classe (SOUZA et al., 2013).

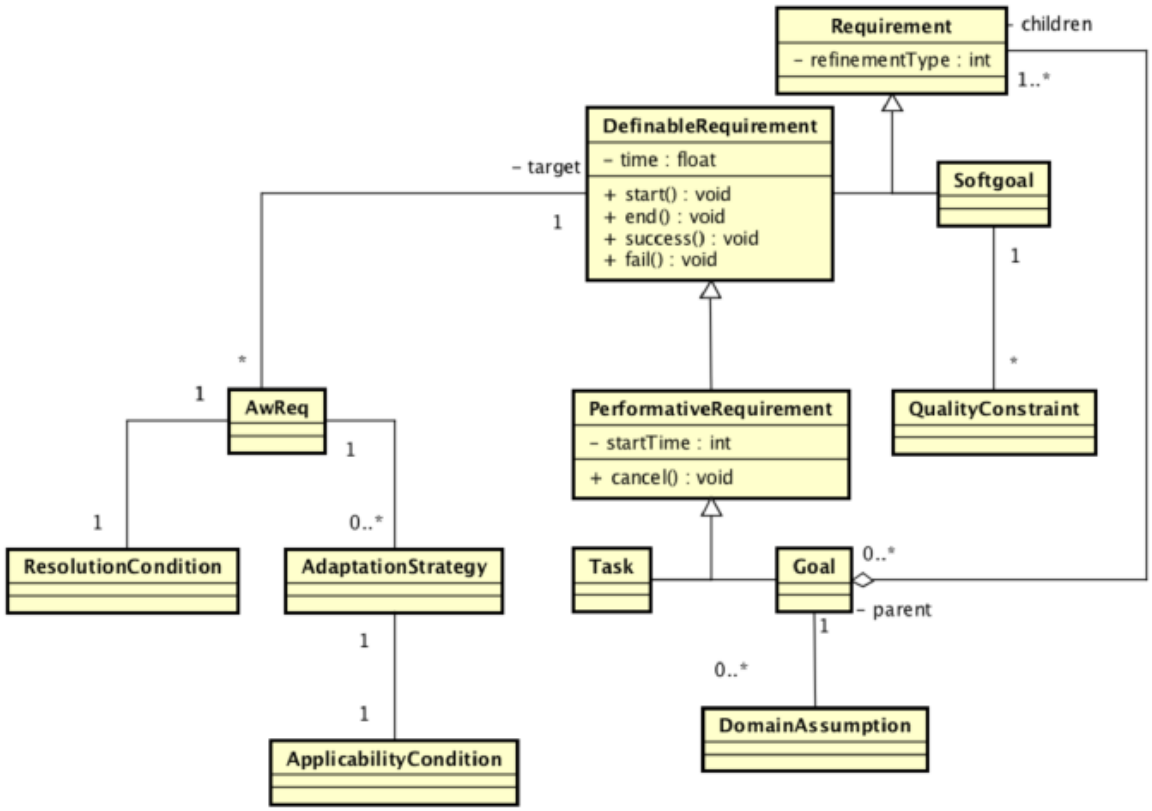

Figura 8 – Metamodelo que define a sintaxe abstrata para o *Zanshin*.

O *Zanshin* pode ser dividido em quatro componentes principais, entretanto essa seção foca na discussão de dois desses: o componente de monitoramento e o componente de adaptação.

2.3.1 Modelos de Objetivos em Tempo de Execução

Muitas vezes os requisitos de um *software* precisam ser modificados durante o ciclo de execução do mesmo. Além disso, ao longo do processo de especificação as partes interessadas no sistema podem apresentar requisitos condicionais, ou seja, que assumem diferentes configurações dependendo da ocorrência de determinada situação (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Em outras palavras, há a necessidade de sistemas que possam se automonitorar e, caso necessário, se adaptar para que seus objetivos continuem sendo satisfeitos (DALPIAZ et al., 2013). Esse tipo de sistema geralmente é composto por duas partes principais: a primeira sendo o sistema em si, que executa uma tarefa para cumprir um objetivo desejado e a segunda sendo um sistema de monitoramento do primeiro, que envia a ele instruções de modificação de suas configurações para que seus objetivos continuem sendo atendidos (SOUZA et al., 2013).

O sistema de monitoramento é construído fundamentado na premissa de que todo

software possui um ciclo de retroalimentação (*feedback loop*) (BRUN et al., 2009) e, assim, o processo de adaptação é realizado com base nesse ciclo, aplicando controladores de resposta que monitoram o comportamento do sistema e enviam estratégias de adaptação (SOUZA et al., 2013). O módulo adaptador verifica, de acordo com as saídas do sistema alvo, se os objetivos internos a esse estão sendo atendidos e, para isso, necessita importar o modelo de objetivos (SOUZA et al., 2013) enriquecido de elementos que indicam os requisitos a serem observados e as estratégias de adaptação relativas, que serão discutidos adiante.

Modelos de sistemas adaptativos incluem requisitos autoconscientes, ou seja, definidos em relação ao sucesso, falha ou qualidade de serviço de outros requisitos (SOUZA et al., 2013). Assim, esses são considerados requisitos "especiais", já que sua operacionalização está relacionada à mudança de outros requisitos (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Ademais, o comportamento do sistema é caracterizado por eventos que ocorrem em tempo de execução e que estão diretamente ligados a instâncias de objetivos (DALPIAZ et al., 2013). Finalmente, é importante observar que essa abordagem é considerada orientada a objetivos já que os requisitos mencionados são derivados do refinamento de objetivos elicitados para o sistema.

Requisitos autoconscientes são divididos por Souza, Lapouchnian e Mylopoulos (2012) em dois tipos principais: Requisitos de Percepção (*Awareness Requirements* ou *AwReqs*) (SOUZA et al., 2013) e Requisitos de Evolução (*Evolution Requirements* ou *EvoReqs*). *AwReqs* são requisitos que referem-se ao estado de outros requisitos em tempo de execução, representando situações onde as partes interessadas desejam que o sistema se adapte (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012), e podem se referir a qualquer tipo de elemento, sejam objetivos, *Softgoals*, tarefas ou pressuposições de domínio. Ainda mais, *AwReqs* indicam o quão critico um requisito pode ser ao descrever o grau de tolerância a falhas do mesmo (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Antes da execução de um sistema, os requisitos estão em estado "Não Decidido" (*Undecided*), e então pode assumir os estados "Sucesso" (*Succeeded*), "Falha" (*Failed*), e no caso de objetivos e tarefas, "Cancelado" (*Canceled*) (SOUZA et al., 2013). É fácil notar que o processo de elicitação de requisitos de percepção só acontece depois que o modelo de objetivos é levantado, e assim como o processo de construção de objetivos, *AwReqs* são sistematicamente criados.

EvoReqs são requisitos que modificam o espaço de comportamento do sistema, permitindo que novas alternativas de requisitos sejam usadas, baseando-se em um conjunto pré-definido de etapas de evolução para os requisitos monitorados (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Isto é, *EvoReqs* são requisitos que especificam uma série de operações primárias em relação a outros requisitos diante de determinadas situações, dizendo ao sistema como adaptar-se (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Instruções para adicionar, remover ou modificar o estado de um objetivo (em nível de instância), desfazer as ações de uma execução que resultou em falha são exemplos de

diretivas de adaptação (SOUZA et al., 2013).

Em suma, *AwReqs* especificam quando um determinado objetivo precisa de mudanças para continuar a ser atendido, enquanto *EvoReqs* especificam como executar tais mudanças. A seguir, o modelo de exemplo apresentado na seção 2.2 é novamente apresentado, porém com novos requisitos de adaptação que são devidamente discutidos na próxima sessão.

2.3.2 Um Exemplo de Uso do *Zanshin*

Na Figura 9 é apresentado o modelo visual completo do sistema de despacho de ambulâncias adaptativo (*Adaptive Computer-aided Ambulance Dispatch* ou *A-CAD*). Nele, observa-se o objetivo principal "Gerar Instruções de Despacho Otimizadas", representado por uma oval, que é imediatamente refinado em outros objetivos e em uma pressuposição de domínio (retângulo, vide legenda na Figura 10). O refinamento entre o objetivo raiz e seus filhos imediatos é do tipo "E" e portanto, para que o objetivo principal seja considerado satisfeito, todos as suas decomposições de primeiro grau precisam ser satisfeitas. Então, verifica-se que o primeiro nível de refinamento do objetivo principal é composto dos objetivos "Gerenciar Chamadas" (*"Call Talking"*), "Identificação de Recursos" (*"Resource Identification"*), "Mobilização de Recursos" (*"Resource Mobilization"*), "Obtenção de Mapas" (*"Map Retrieval"*), "Receber atualizações sobre o Incidente" (*"Incident Update"*) e da pressuposição de domínio "Dados sobre recursos está sempre atualizado" (*"Resource data is up-to-date"*).

O processo de refinamento do modelo então segue até que todos os objetivos sejam completamente decompostos em tarefas ou pressuposições de domínio. *Softgoals* são simbolizados por nuvens e refinados em restrições de qualidade representados por retângulos com cantos arredondados. Exemplificando, o *Softgoal* "Chegada Rápida" (*Fast Arrival*) é operacionalizado por "Ambulâncias chegam em oito minutos" (*Ambulances arrive in 8 min*), assim, tem-se um critério claro de satisfação para um objetivo que antes possuía diversos tipos de interpretação, porém, agora, sabe-se que uma ambulância chega rapidamente se consegue estar no local do acidente em menos de oito minutos a partir da chamada.

Os requisitos de percepção são representados por um círculo oco. Por exemplo, o *AwReq* identificado por "AR15", indica que o objetivo "Registrar Chamados" deve "Nunca falhar" (`NeverFail`). Os *EvoReqs* referentes a cada um dos *AwReqs* não são representados nesse modelo, e devem ser especificados em forma de sequência de operações sobre os elementos do modelo de objetivos (essa escolha de representação visa aprimorar a legibilidade do modelo). Além dos *AwReqs*, são especificados também os parâmetros de controle (*Control Parameters*), representados por losangos, que indicam parâmetros do sistema que podem ser reconfigurados durante a adaptação. Todas as formas de

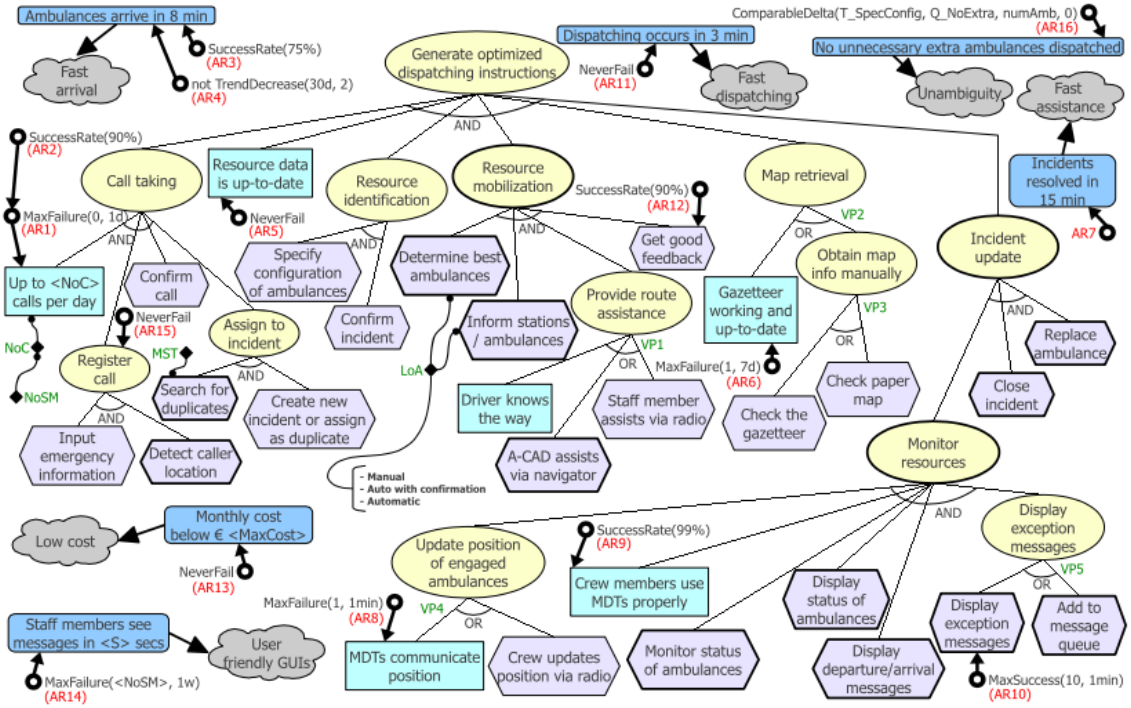

Figura 9 – Exemplo de modelos de objetivos de um sistema de despacho de ambulâncias (SOUZA, 2012).

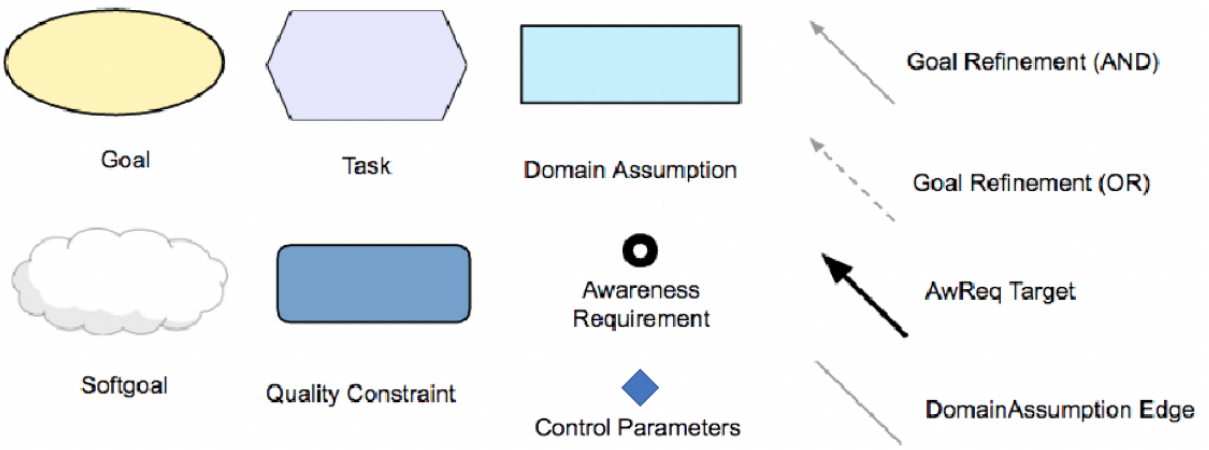

Figura 10 – Representação gráfica de elementos de *GORE*.

Tabela 1 – Tabela de especificação das estratégias de adaptação de AR15.

AR15	NeverFail(G_RegCall)	1. Retry(5000) 2. RelaxDisableChild(T_DetectCaller)

representação estão resumidas na Figura 10.

Tomando como exemplo o *AwReq* "AR15", podemos definir seus respectivos Requisitos de Evolução. Primeiramente, se o objetivo vier a falhar, define-se a primeira estratégia de adaptação como "Tentar Novamente em 5 segundos no máximo uma vez" (`RetryStrategy(5000)`), caso falhe mais do que uma vez, aplica-se outra estratégia "Relaxar Condições ao Desabilitar Filho" (`RelaxDisableChild(TDetectCaller)`), que desativa o requisito "Detectar Localização da Ligação", também aplicado no máximo uma vez. Essas decisões são sumarizadas na Tabela 1.

O processo de especificação de como as estratégias de evolução são realizadas pelo sistema é discutido na próxima seção.

2.3.3 Monitoramento

O módulo de monitoramento necessita que o sistema alvo implemente funcionalidades de registro (*logs*). Por meio do registro, o sistema pode detectar mudanças nos estados das instâncias de um *AwReq* e assim notificar o serviço de adaptação sobre essas ocorrências.

Para identificar o modelo de objetivos do sistema alvo, o componente de monitoramento necessita da especificação do modelo do domínio (também em *Ecore*). Por meio dele, o *Zanshin* criará instâncias desses objetivos estendendo as classes carregadas do metamodelo de *GORE* referentes ao tipo de cada um deles (Figura 8). Por exemplo, ao criar a tarefa "Especificar configuração de ambulâncias", o sistema criará uma instancia dessa tarefa específica, que fará referência à classe `Task` criada no momento em que o metamodelo foi importado. A classe de tarefa especializa a classe `PerformativeRequirement` que é uma especialização `DefinableRequirement` (ver Figura 11).

Performative Requirements definem os tipos de requisitos que são ou podem ser refinados em tarefas e, assim, possuem ações que são executadas pelo sistema ou seus usuários. *Definable Requirements* são os requisitos que devem possuir um estado definido em algum momento da execução, como por exemplo os objetivos e pressuposições de domínio.

Assim que o *Zanshin* cria as classes dos requisitos do sistema alvo, o processo de monitoramento apoia-se nos métodos definidos nas metaclasses `DefinableRequirement` e `PerformativeRequirement` (SOUZA, 2012) para realizar a monitoração, os métodos disponíveis são:

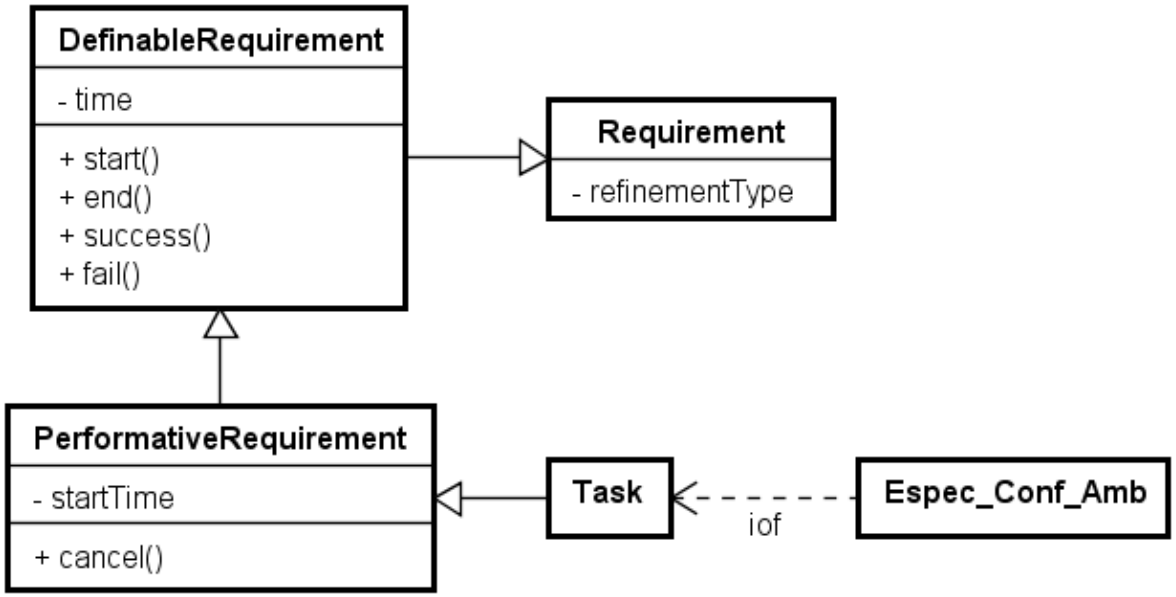

Figura 11 – Exemplo de um elemento do modelo de domínio específico instanciado em *Zanshin*.

- `start()`: esse método é chamado nas classes de critérios de qualidade e pressuposições de domínio imediatamente antes da análise de seu estado de satisfação. Para o caso de tarefas, esse método é chamado assim que um usuário inicia uma tarefa, e então todos os "ancestrais" desse elemento também têm esse método invocado;
- `success()`: chamado quando um requisito é satisfeito e propagado para todos os pais desse objetivo para "avisar" que o filho chegou ao estado de sucesso;
- `fail()`: segue a mesma lógica dos item anteriores, propagando a "falha" de um requisito aos seus superiores;
- `cancel()`: para o caso de requisitos como objetivos e tarefas, esse método é usado quando um procedimento é cancelado pelo usuário, usando da mesma lógica de propagação dos métodos anteriores para informar o ocorrido aos pais;
- `end()`: chamado assim que um requisito muda para os estados de sucesso, falha ou cancelamento.

Pela descrição dos métodos acima, é possível entender como o processo de monitoramento funciona: basicamente os requisitos monitorados possuem métodos que podem ser chamados em cada caso que assumem, seja de sucesso, falha ou cancelamento. Assim que chamados, os métodos invocam os mesmos procedimentos nos elementos pais, permitindo que a atualização de um requisito se propague a todos os elementos relacionados, atualizando assim todo o modelo.

Logo que o componente monitor detecta uma mudança de estado em um requisito, ou seja, assim que qualquer um dos métodos acima é invocado, ele imediatamente envia uma notificação sobre essa mudança ao módulo de adaptação, que então inicia o processo definido para aquele determinado requisito de percepção (SOUZA, 2012).

2.3.4 Adaptação

Em termos de implementação, o algoritmo do módulo de adaptação cria uma sessão para cada *AwReq* que está sendo monitorado e então gera uma fila de eventos que se referem a estratégias adaptativas. Assim, essa linha de eventos pode ser usada para verificar se uma estratégia é ou não aplicável a determinada situação.

Primeiramente, deve-se verificar o estado de um objetivo apontado por um *AwReq*, para isso é checada a condição de resolução daquele. Então, caso uma avaliação considere que o objetivo está em estado de falha, o sistema segue (de acordo com uma ordem pré-definida) a lista de estratégias de adaptação disponíveis, procurando alguma que tenha sua condição de aplicabilidade verdadeira. Caso encontre, aplica a estratégia definida por aquela condição de aplicabilidade e então retorna ao estado inicial, onde a checagem da satisfabilidade do objetivo é realizada novamente. Caso o objetivo ainda não tenha sido satisfeito, o processo começa novamente, obedecendo restrições como, por exemplo, o número de tentativas de aplicação de uma estratégia, definido individualmente. Caso não encontre uma condição para aplicar uma estratégia, o sistema ativa o método "abortar" (`abort()`) (SOUZA et al., 2013).

Ao final, quando uma sessão de adaptação é considerada resolvida, a mesma deve ser terminada e, se for necessário que o processo seja aplicado novamente, uma nova sessão é criada. Entretanto, se uma sessão termina sem ter resolvido o problema, o *framework* continuará trabalhando nela do ponto em que parou assim que receber uma nova requisição para adaptação daquele mesmo *AwReq*. Porém, algumas estratégias também podem forçar que a sessão seja reiniciada quando executada (SOUZA et al., 2013). Essa processo é conhecido como Evento-Condição-Ação (*Event-Condition-Action* ou *ECA*) (MORIN et al., 2009).

O código mostrado na Listagem 2.1 resume o processo *ECA* para realizar a seleção de estratégias de adaptação:

Listagem 2.1 – Código do processo ECA

```
1 processEvent(ar : AwReq) {
2   session = findOrCreateSession(ar.class);
3   session.addEvent(ar);
4   solved = ar.condition.evaluate(session);
5   if(solved) break;
6
7   ar.selectedStrategy = null;
```

```
8   for each s in ar.strategies {
9     appl = s.condition.evaluate(session);
10    if (appl) {
11      ar.selectedStrategy = s;
12      break ;
13    }
14  }
15
16  if (ar.selectedStrategy == null)
17    ar.selectedStrategy = ABORT;
18
19  ar.selectedStrategy.execute(session);
20  ar.condition.evaluate(session);
21 }
```

O algoritmo da Listagem 2.1 inicia obtendo a sessão de adaptação referente à classe do *AwReq* requisitado (caso não haja uma sessão existente, uma nova é criada). Obtida a sessão de adaptação, o algoritmo tem então acesso à lista de eventos de aplicabilidade referentes e, então, adiciona o objeto do *AwReq* à sessão e imediatamente verifica o estado da mesma (por meio da condição de resolução), parando caso tenha retornado estado de sucesso. Caso contrário, o processo continua procurando por uma estratégia que seja aplicável, checando se a condição de aplicabilidade é verdadeira; caso seja, interrompe o processo e dá à sessão de adaptação uma nova chance de verificar o estado do *AwReq*. Caso todas as condições sejam falsas e nenhuma estratégia seja selecionada, seleciona a estratégia padrão (`abort()`) e termina o processo (SOUZA, 2012).

Para exemplificar esse processo, tomemos novamente o *AwReq* "AR15", que garante que o requisito "Registrar chamada" deve nunca falhar. Caso seja detectado pelo processo de monitoramento que esse requisito apresenta estado de falha, o módulo monitor imediatamente ativa o componente de adaptação, que segue o processo ECA para aplicar estratégias ao "AR15". Pelo fragmento de XML mostrado na Listagem 2.2, vê-se que a condição de resolução desse requisito é do tipo `SimpleResolutionCondition` (considera o objetivo solucionado de acordo com o estado de seus filhos), e que a primeira estratégia a ser selecionada é `RetryStrategy`, ou seja, tentar novamente (em 5s, de acordo com a propriedade `time`), entretanto essa estratégia possui a condição de aplicabilidade `MaxExecutionsPerSessionApplicabilityCondition`, o que significa que ela só pode ser aplicada um determinado número de vezes naquela sessão (nesse caso apenas uma vez, de acordo com o atributo `maxExecutions`). Caso essa condição seja falsa, a próxima estratégia refere-se a desativar um dos filhos desse objetivo (`RelaxDisableChildStrategy`), no caso a tarefa "Detectar localização da chamada", e possui a mesma condição de aplicabilidade. Se nenhuma dessas condições puderem ser satisfeitas o sistema aborta e, caso contrário, seleciona a primeira estratégia aplicável e verifica novamente o estado do objetivo. A condição `SimpleResolutionCondition` dita que um objetivo é satisfeito apenas se seus filhos estiverem em estado de sucesso (respeitando a regra booleana do refinamento).

Tabela 2 – Tabela de Requisitos de Adaptação.

Regra	**Efeito**
Abort()	Sistema deve falhar de forma prevista, exibindo mensagem de falha, por exemplo.
Delegate(a)	Delegar tarefa a um outro ator do sistema.
RelaxDisableChild(r, l, child)	Para de considerar o estado de um requisito em determinada execução do sistema.
Replace(r, copy, l, newReq)	Substitui requisito.
Retry(copy, time)	Tenta novamente em determinado período de tempo.
StrengthenEnableChild(r, l, child)	Volta a considerar estado de um requisito.
Warning(a)	Avisa um ator sobre o estado atual do sistema.

Listagem 2.2 – Estratégias de adaptação de AR15

```
1 <awreqs xsi:type="acad:AR15">
2   <condition xsi:type="eca:SimpleResolutionCondition"/>
3     <strategies xsi:type="eca:RetryStrategy" time="5000">
4       <condition xsi:type="eca:MaxExecutionsPerSessionApplicabilityCondition"
            maxExecutions="1"/>
5     </strategies>
6     <strategies xsi:type="eca:RelaxDisableChildStrategy" child="//@rootGoal/
          @refinements.0/@refinements.0/@refinements.1">
7       <condition xsi:type="eca:MaxExecutionsPerSessionApplicabilityCondition"
            maxExecutions="1"/>
8     </strategies>
9 </awreqs>
```

É importante salientar que as classes referentes a estratégias de adaptação, assim como as condições de aplicabilidade e resolução podem ser estendidas para casos mais complexos, envolvendo inclusive a interação humana (SOUZA, 2012). Uma lista completa das estratégias de adaptação disponíveis por padrão no *Zanshin* é mostrada na Tabela 2.

3 Análise do Metamodelo do *Zanshin*

Esta seção trata do processo de reavaliação do metamodelo operacional do sistema *Zanshin*, anteriormente já apresentado na Figura 8.

Durante estudos realizados sobre a implementação do *framework Zanshin*, foram detectadas algumas oportunidades de melhoria relacionadas principalmente ao metamodelo de objetivos utilizado pelo sistema. A maioria das adequações identificadas referem-se a dois motivos: ou o metamodelo não era restrito o suficiente, deixando que algumas situações indesejadas pudessem ser modeladas; ou o metamodelo não refletia fielmente alguns conceitos de *GORE* que foram mais bem elaborados em discussões sobre o tema. Então, o metamodelo original foi retrabalhado levando também em consideração as definições propostas pela ontologia *GORO*, que permitiu definir elementos com mais precisão e adicionar restrições que antes não existiam. A Subseção 3.1 trabalha oportunidades de melhoria do metamodelo original, enquanto a Subseção 3.2 demonstra como soluções para essas propostas de melhoria foram elaboradas.

O código fonte da versão original do *Zanshin* pode ser encontrado em <https://github.com/sefms-disi-unitn/Zanshin>, assim como a versão modificada para adequação à nova proposta de metamodelo pode ser encontrada em <https://github.com/hbcesar/zanshin>.

3.1 Revisão do Metamodelo

A partir da análise de cada elemento do metamodelo antigo, foram encontradas as seguintes oportunidades de melhoria:

1. **Requirement**: o fato desse elemento estar no topo da hierarquia e todos os outros elementos serem especializações dele faz com que suas características sejam herdadas por todos os outros componentes do modelo. Porém, nota-se que algumas características na verdade são inerentes a apenas alguns elementos. Por exemplo: *Requirement* possui a relação de agregação *Parent* para *Children*, permitindo que cada elemento tenha zero ou um "Pai" e um ou vários "Filhos". Entretanto, a agregação pai/filho é indesejada nos elementos *AwReq*, *DomainAssumption* e *QualityConstraint*, visto que esses não são refinados, apenas possuem "alvos".

2. **DeinableRequirement** e **Softgoal**: *Softgoal* possui o mesmo tipo de comportamento de um *DefinableRequirement* e, portanto, seria esperado que aquele também fosse uma especialização deste. Assim, com essa modificação seria desnecessário a

utilização de duas classes (*Requirement* e *DefinableRequirement*) na composição do modelo, que poderiam perfeitamente ser reduzidas a apenas uma.

3. **Softgoal** e **QualityConstraint**: O modelo permite que um *Softgoal* contenha nenhuma operacionalização, o que não faria sentido pois assim não seria possível identificar quando esse elemento foi bem sucedido ou não.

4. **DefinableRequirement** e **AwReq**: os elementos de adapatação, ao se referirem a qualquer outro tipo de elemento e tratarem especificamente do elemento ao qual se referem, deveriam ter, na verdade, uma relação de composição com o elemento mais alto da hierarquia de especializações, permitindo então que todos os outros elementos do modelo contivessem *AwReqs*.

5. **Goal** e **Task**: *Tasks* são refinamentos de *Goal* e por isso seria importante que houvesse algum tipo de relacão direta entre os dois elementos, onde fosse possível identificar que quando um objetivo fosse operacionalizado por tarefas e quais seriam essas tarefas. Além disso, identificou-se uma oportunidade de melhoria da nomenclatura de "*Goal*" para "*HardGoal*", refletindo uma melhor impressão sobre o papel do elemento e as diferenças entre ambos.

6. **DomainAssumption**: a forma como *DomainAssumptions* se relacionam com os outros elementos poderia ser refinada, o objetivo aqui é que qualquer tipo de elemento que for passível de ser operacionalizado (*PerformativeRequirement*) pudesse conter uma lista de referências a pressuposições de domínio.

3.2 Proposta

Após levantamento das melhorias possíveis, foi elaborado o metamodelo apresentado na Figura 12. Primeiramente, pode-se observar que as especialiações e relações de agregação e composição entre os elementos foram refinadas, assim, identifica-se que:

- A partir de agora, todos os elementos são especializações de *GOREElement*, que passa a englobar características das antigas classes *Requirement* e *DefinableRequirement*, resolvendo então o problema 2.

- A agregação pai/filho agora ocorre em dois tipos e acontece em um elemento mais específico: um requisito pode ser decomposto (*decompose*) em outros, referindo-se assim a refinamentos do tipo "E", ou refinado por meio da agregação denominada "alternative", que trata de alternativas a um requisito, ou seja, refinamentos do tipo "OU". Dessa forma, o metamodelo pode retratar melhor os tipos de refinamentos admitidos e quais são, exatamente, os componentes que os aceitam. Além do mais, a partir desta melhoria os refinamentos acontecem somente nos elementos que

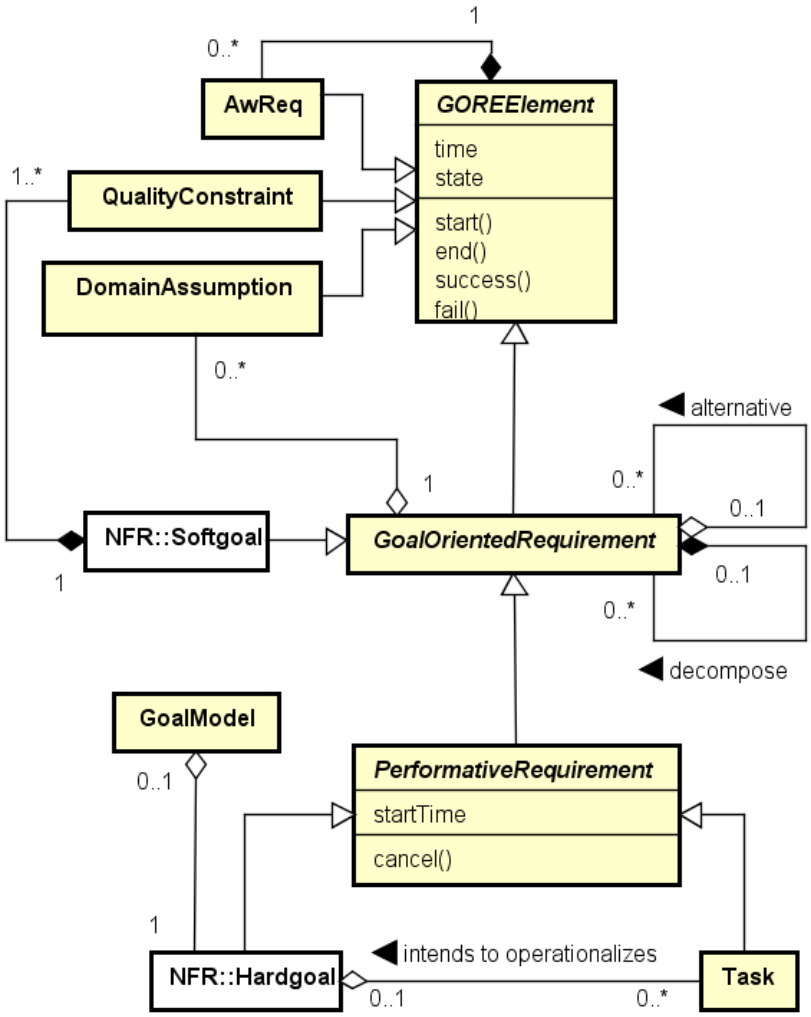

Figura 12 – Proposta de evolução do metamodelo do *framework Zanshin*.

especializam *GoalOrientedRequirement*, ou seja, somente *Hardgoals*, *Softgoals* e *Tasks*. Essa modificação soluciona o problema 1.

- *Softgoal* e *QualityConstraint* passam a se relacionar por meio de relação de decomposição e, portanto, também fica mais evidente que todo *Softgoal* deve ser operacionalizado por pelo menos um *QualityConstraint*, dando fim, assim, ao problema 3.

- Nessa proposta, *AwReqs* têm uma relação direta de composição com o *GOREElement* logo, define-se que qualquer tipo de requisito no modelo pode conter elementos de adaptação, assim resolve-se o problema 4.

- No novo metamodelo, foi criada uma relação direta entre *Tasks* e *Hardgoals* (denominados anteriormente por *Goals*), assim fica explícito o relacionamento entre esses dois elementos, sinalizando que objetivos são refinados em tarefas, que podem ser refinadas apenas nelas mesmas. Soluciona-se então o problema 5.

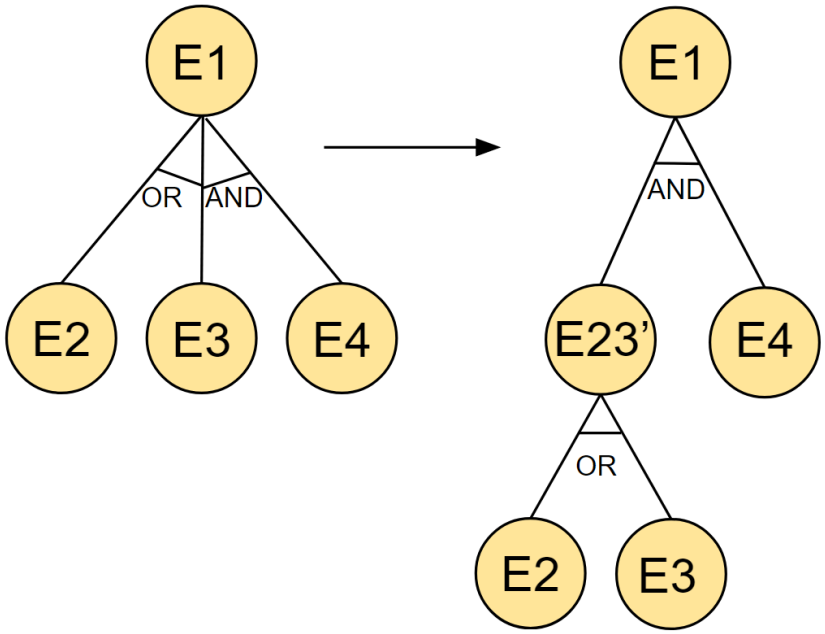

Figura 13 – Exemplificação da representação de refinamentos no novo metamodelo.

- Propõe-se, por fim, uma relação de agregação entre *DomainAssumption* e *GoalOrientedRequirement*, assim fica claro que *Domain Assumptions* são elementos que interagem apenas com *Hardgoals*, *Softgoals* e *Tasks*, findando o problema 6.

Uma observação importante que deve ser feita refere-se ao fato de que um elemento só pode conter refinamentos exclusivamente do tipo "E" ("AND") ou exclusivamente do tipo "OU" ("OR"). Essa decisão tem por objetivo facilitar a leitura e interpretação do diagrama, e não compromete a representatividade do modelo pois assume-se que, caso necessário, o modelador pode agrupar refinamentos do mesmo tipo em um objetivo, como exemplificado na Figura 13.

4 Unagi

O editor gráfico *Unagi* foi desenvolvido com o principal objetivo de facilitar a criação de arquivos de especificação do modelo de domínio para uso no *Zanshin*, considerando que até então não havia nenhuma ferramenta que automatizasse esse processo, assim o usuário precisava escrever o arquivo XML e *Ecore* manualmente.

Usando um ferramental disponível na plataforma Eclipse™ para criação de editores gráficos usando EMF, além da utilização de aspectos de desenvolvimento dirigido por modelos, o processo de desenvolvimento do *Unagi* pode ser dividido em duas fases: a criação do editor gráfico usando Sirius™ e o desenvolvimento do conversor de diagramas apoiado em código gerado automaticamente por técnicas MDD (*Model-Driven Development*). As próximas seções dão detalhes sobre o processo de implementação da ferramenta: a Seção 4.1 explica como o editor gráfico foi elaborado, usando as tecnologias já discutidas nesse texto, e então a Seção 4.2 explica as funcionalidades da ferramenta modeladora. Por fim, a Seção 4.3 demonstra como o componente responsável pela conversão do modelo visual para os arquivos nos padrões do *Zanshin* foi construído.

O código fonte do plugin, bem como o arquivo de instalação da ferramenta podem ser encontrados em <https://github.com/hbcesar/unagi>.

4.1 Criação do Editor Gráfico

A primeira parte do desenvolvimento da ferramenta consistiu na criação do editor gráfico para modelagem de diagramas de objetivos para sistemas adaptativos. Devido ao fato de *Zanshin* ser um sistema que também se baseia nas ferramentas da plataforma Eclipse™, aliado ao fato dessa plataforma também prover todos os utensílios necessários para o desenvolvimento de editores de diagramas, como *Ecore*, EMF e o plugin Sirius™, foi fácil decidir que para o desenvolvimento do *Unagi* também seriam usados os mesmos recursos, com objetivo de permitir maior compatibilidade com o *Zanshin* em trabalhos futuros.

Inicialmente, o mesmo modelo *Ecore* usado para operacionalização dos requisitos no *Zanshin* foi usado para geração automática de código dentro do Eclipse™ EMF. É importante salientar que junto com as classes específicas de domínio, o EMF também gera código automático que serve como base para criação de editores e para criação de processos de teste e validação (Figura 14). Em outras palavras, o EMF gera classes relativas ao metamodelo que podem ser instanciadas em tempo de execução da ferramenta gráfica. Assim, após completa especificação do *Ecore*, foi gerado código automático que serviu

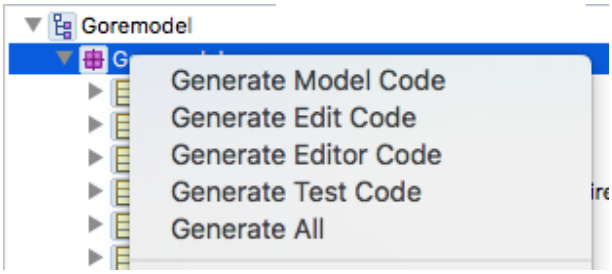

Figura 14 – Geração automática de código no Eclipse™.

como base para o processo inicial de implantação do editor. O código gerado pode então ser executado como uma aplicação Eclipse™ e a partir daí pode-se criar um editor de modelos personalizado usando o Sirius™.

Rodando na instância do Eclipse™ a partir do código de editor gerado automaticamente, o plugin Sirius™ permite que sejam definidos pontos de perspectiva com diferentes especificações de representação gráfica para os objetos do modelo *Ecore*. Essa especificação é realizada a partir da criação de Projetos de Especificação de Perspectiva (*Viewpoint Specification Project* ou *VSP*), onde são definidos como os elementos serão representados, como suas instâncias serão armazenadas nas classes do metamodelo *Ecore*, qual o comportamento do modelo quando relações são criadas, dentre outros.

A princípio, cria-se dentro do modelo de VSP uma camada (*layer*) que representa a perspectiva a ser representada, nela são especificados os elementos do modelo *Ecore* que serão representados, aqui chamados de *Nodes*, que pode ter um estilo padrão ou um estilo condicionado a determinada situação, além disso, pode-se usar formas geométricas predefinidas ou importar imagens externas.

Como visto na Figura 15, o elemento *Goal* tem como representação padrão uma elipse amarela. Essa representação pode ser mais bem configurada por meio da paleta de propriedades da mesma (Figura 16), onde podem ser especificadas características visuais como cor da linha e tamanho do nome.

Na Figura 15 também pode-se observar que além da representação dos *Nodes*, é necessário configurar a criação das linhas de ligação entre os elementos, pois por meio dessas são definidos os refinamentos entre os componentes do modelo. Em `childReqEdge` é especificado que se um objeto possui tipo de refinamento “OU” (“OR”), deve ser usado o tipo de linha tracejada, entretanto, se for do tipo “E” (“AND”), usa-se linha contínua. A linguagem usada para escrita de termos condicionais é conhecida como *Acceleo Query Language* (MUSSET et al., 2006b), e será posteriormente discuta nesse capítulo.

Além da caracterização dos elementos que serão representados em uma perspectiva, o Sirius™ também necessita que sejam configuradas propriedades de instanciação dos elementos, assim, além dos *Nodes* de representação, devem ser criados os *Nodes* de criação, mostrados na Figura 17, onde podem ser observados, por exemplo, a criação do *Node Goal*.

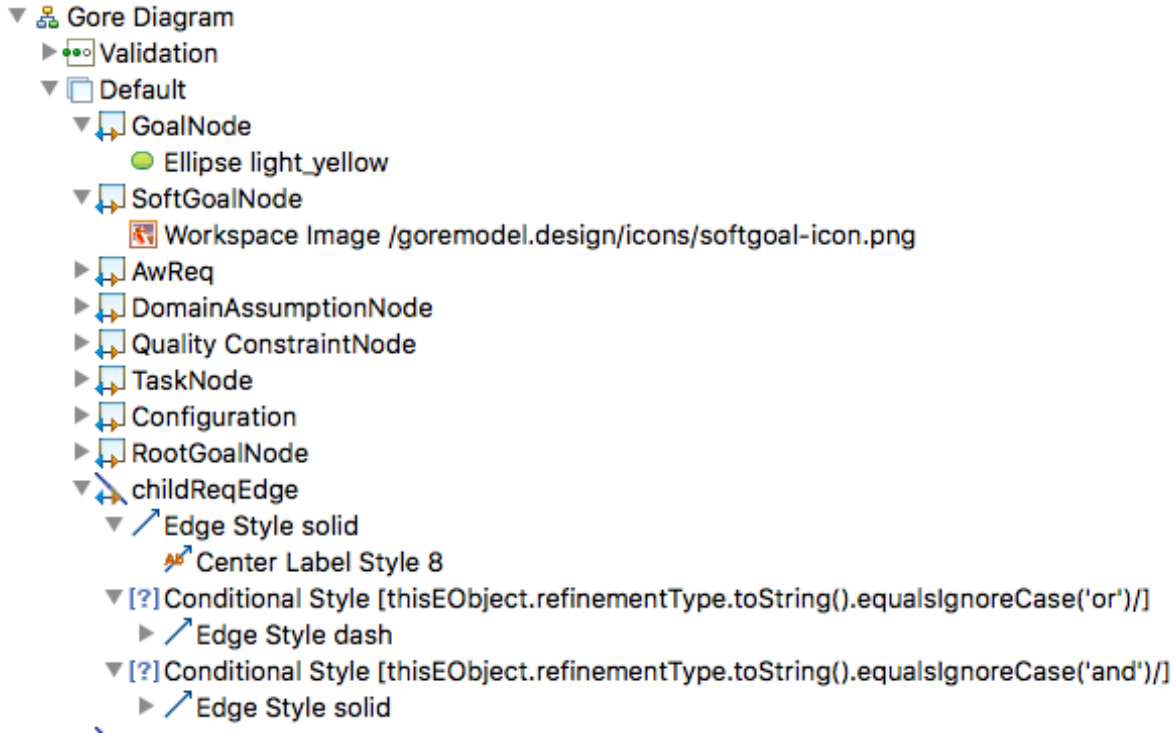

Figura 15 – Sirius Viewpoint Specification Project.

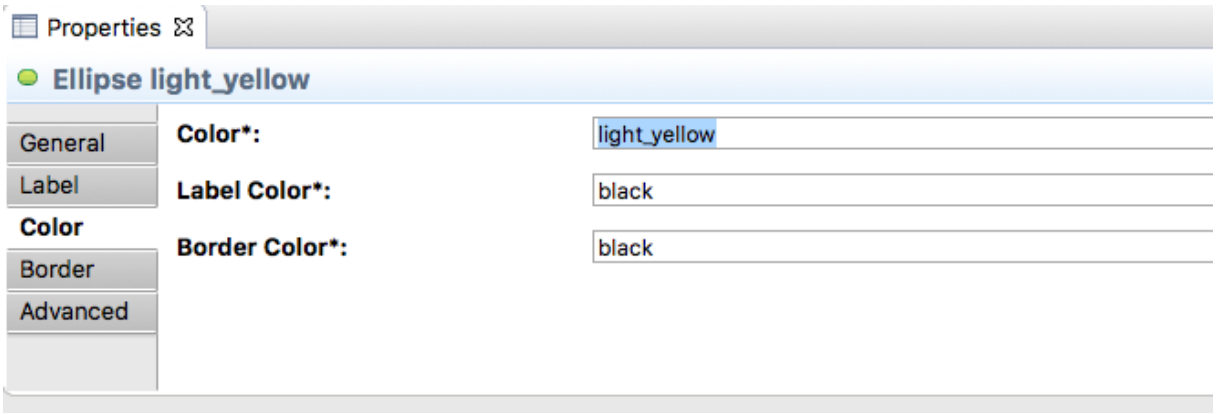

Figura 16 – Propriedades de Representação de Elemento em Sirius™.

Nessa etapa é necessario que primeiramente seja especificado em que tipo de instância do metamodelo o novo objeto será armazenado. Para o *Node* de exemplo pode-se observar que há duas ocasiões:

- Caso o elemento for o elemento pai, ou seja, a raiz da árvore de representação, deve ser armazenado em uma variável especial da classe *GoalModel*, de nome *rootGoal*. Esse caso é verificado ao checar se a variável ainda não foi definda (`Case[oclIsUndefined(container.rootGoal)]`).
- Caso o elemento for refinamento de qualquer nível do elemento pai, então é guardado na variável *children* da classe *GoalModel*.

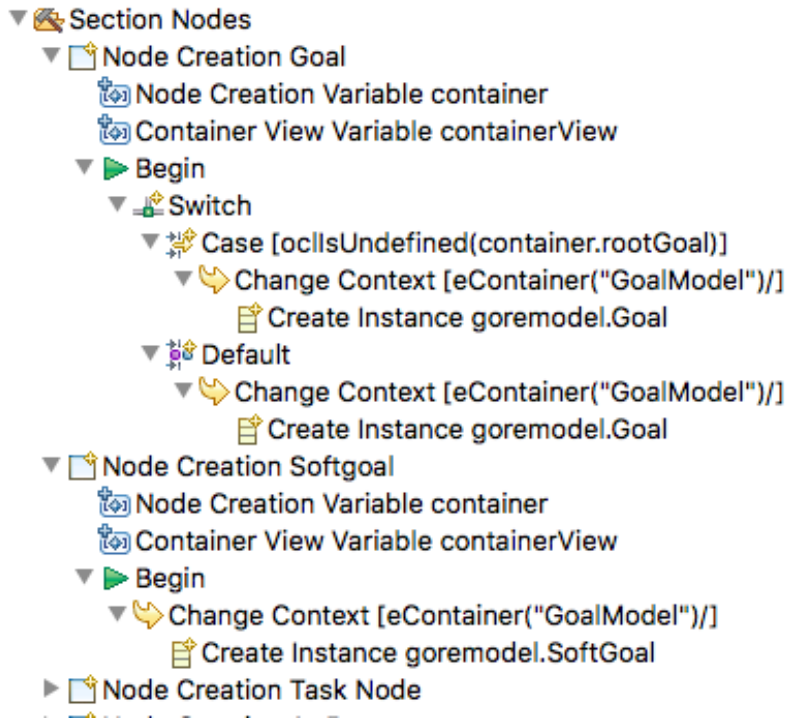

Figura 17 – Criação de Nodes no Sirius^{TM}- Exemplo de criação de nó.

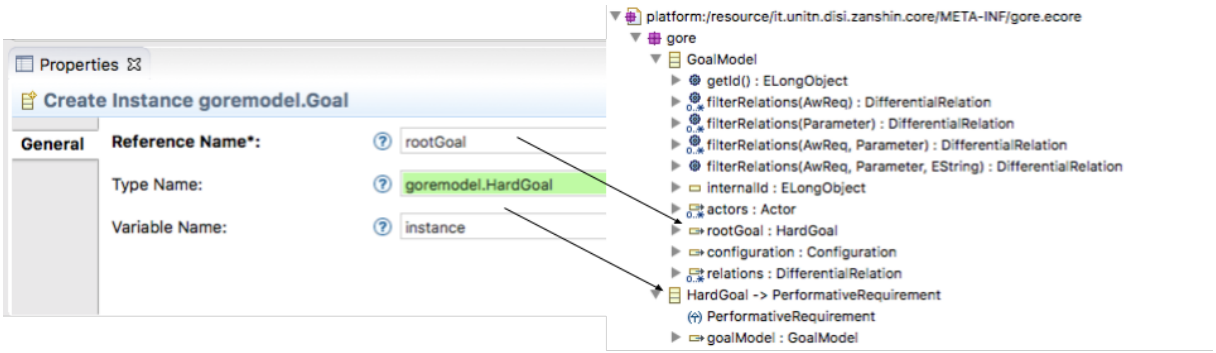

Figura 18 – Criação de Nodes no Sirius^{TM}- Exemplo de criação de instancia de nó.

Decididos os casos, é necessário criar uma instância da classe relativa ao novo elemento. Esse processo é definido por meio das propriedades da opção *Create Instance* (Figura 17), que são definidas de acordo com o modelo *Ecore*, como pode ser visto na Figura 18. O campo *referenceName* deve se referir à variável da classe de contenimento que será usada para armazenar o novo elemento, enquanto *Type Name* refere-se à classe do novo componente do modelo.

Por fim, foi modelada uma perspectiva diferente para especificação das estratégias de adaptação dos *AwReqs*, permitindo que cada um tenha sua propria representação em um diagrama separado, com o objetivo de evitar que o modelo principal ficasse "poluído" por excesso de informação.

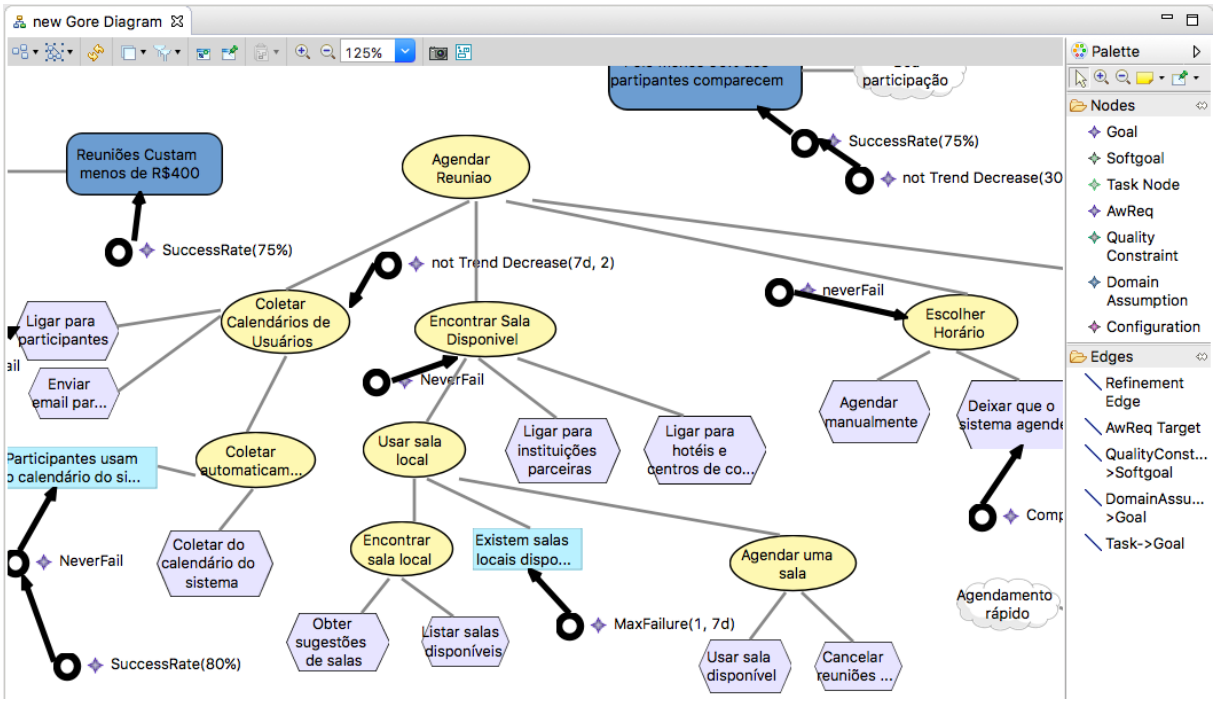

Figura 19 – Editor Unagi.

4.2 O Editor Gráfico

Após todo o processo de detalhamento da representação dos elementos do editor, é possivel então criar um modelo de especificação de perspectiva (*Viewpoint Specification Model*) que permite a criação do diagrama usando recursos de arrastar e soltar, considerados intuitivos por estarem presentes na maioria dos editores gráficos atuais. A Figura 19 mostra o editor em execução, onde pode ser vista a área de modelagem, bem como a paleta de elementos que podem ser criados e também a paleta com tipos de refinamentos disponíveis para modelagem. Esses refinamentos podem ser selecionados e então desenhados apenas clicando no elemento de origem e destino, nesta ordem. As propriedades relativas a cada elemento do modelo podem ser modificadas por meio da paleta de opções que aparece na área inferior do editor (Figura 20).

Para acessar a perspectiva que permite o detalhamento dos *EvoReqs* referentes a cada *AwReq* é necessário simplesmente um clique duplo sobre o Requisito de Percepção desejado. O editor então oferece a opção de criação de um subdiagrama de detalhamento dos elementos, mostrado na Figura 21. Assim, o usuário pode determinar as Condições de Resolução, as Condições de Aplicabilidade e as Estratégias de Adaptação usando a mesma lógica de arrastar e soltar, selecionando os elementos na parte direita da tela. É importante dizer que o modelo só aceita a criação de um elemento específico após seu elemento "pai" ter sido criado, por exemplo, só é possível criar uma Estratégia de Adaptação após ter definido uma Condição de Aplicabilidade.

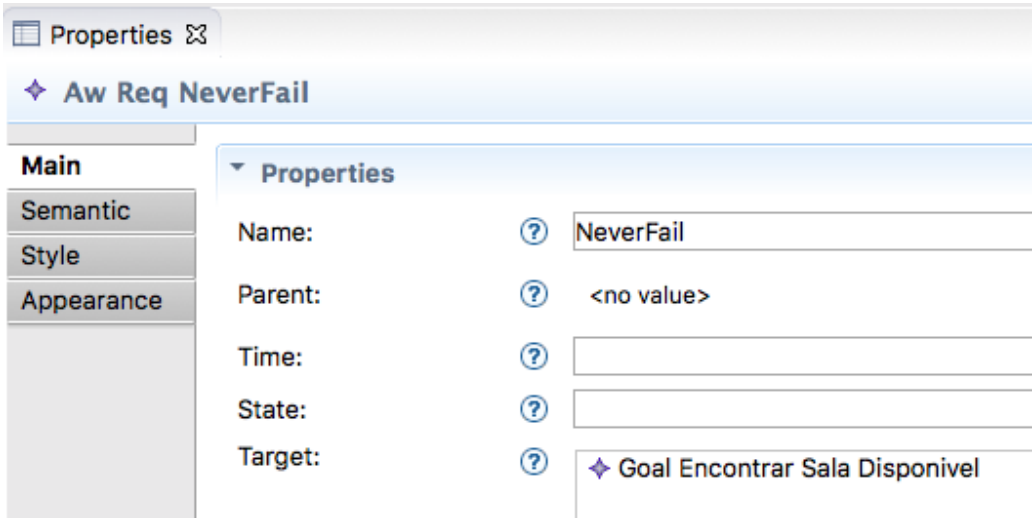

Figura 20 – Editor Unagi - Paleta de Opções.

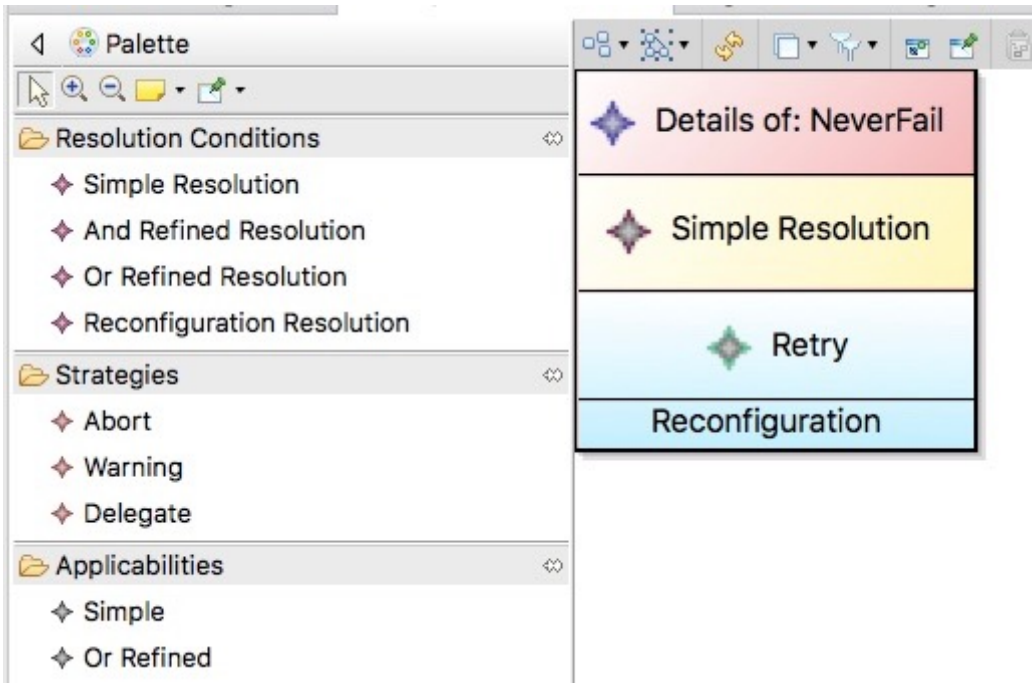

Figura 21 – Editor Unagi - Subdiagrama de Especificação de *EvoReqs*.

4.3 Conversor

O conversor do Unagi foi desenvolvido usando linguagem Acceleo. O processo de transformação consiste em percorrer a árvore de elementos do editor gráfico e, baseando-se no *template* implementado usando a linguagem, os objetos são transformados um a um em linhas do texto dos arquivos XML formatadas aos padrões do *Zanshin*. De uma forma mais detalhada, o editor procura inicialmente pelo elemento raiz, gera sua linha com as características específicas, checa pela existência de refinamentos e: se houver, percorre a lista de elementos e a lista de refinamentos deles recursivamente, até que sejam atingidas as folhas da árvore. Ao fim, os arquivos que podem ser importados no *Zanshin* são armazenados na pasta "zanshin-src" criada dentro da pasta do projeto de modelagem. Caso o conversor encontre algum erro no processo, um arquivo de nome "ERROR.txt" é gerado com informações sobre o ocorrido e, se disponível, um link para página de ajuda.

5 Validação

Nesse capítulo são discutidas as etapas de verificação e validação de ambas as realizações no contexto desse trabalho: a adaptação do *Zanshin* à nova proposta de metamodelo e a criação da ferramenta *Unagi*.

5.1 Zanshin

O processo de validação do *Zanshin* teve foco nas simulações de adaptação que já estavam disponíveis. O *framework* disponibiliza alguns exemplos de modelos de sistemas que podem ser usados para simular situações em que seriam necessárias operações de adaptação sobre os objetivos elicitados. Entre essas simulações disponíveis podemos citar o sistema de agendamento de reuniões (*Meeting Scheduler*) e o sistema de despacho de ambulâncias (*A-CAD*).

Nessa fase, os arquivos *Ecore* e XML de especificação dos metamodelos dos sistemas citados foram reescritos para referenciarem corretamente os elementos do novo metamodelo. Primeiramente, o metamodelo anterior referia-se aos refinamentos do objetivo principal como “children”, e na nova proposta são referenciados como “refinements”, portanto o primeiro passo foi a modificação do nome da referencia, como mostrado nas listagens 5.1 e 5.2.

Listagem 5.1 – Trecho de XML representando o A-CAD no metamodelo antigo

```
1 <rootGoal xsi:type="scheduler:G_SchedMeet">
2   <children xsi:type="scheduler:T_CharactMeet"/>
3     <children xsi:type="scheduler:G_CollectTime" refinementType="or">
4       ...
5     </children>
6   ...
```

Listagem 5.2 – Trecho de XML representando o A-CAD no novo metamodelo

```
1 <rootGoal xsi:type="scheduler:G_SchedMeet">
2   <refinements xsi:type="scheduler:T_CharactMeet">
3     <awreqs xsi:type="scheduler:AR1">
4       <condition xsi:type="it.unitn.disi.zanshin.model:SimpleResolutionCondition
            "/>
5       <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time
            ="5000">
6         <condition xsi:type="it.unitn.disi.zanshin.model:
              MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="2"/>
7       </strategies>
8     </awreqs>
9   </refinements>
10  <refinements xsi:type="scheduler:G_CollectTime" refinementType="or">
```

```
11  ...
```

Além da substituição do nome para referenciar os refinamentos, pode-se identificar que agora os *AwReqs* são definidos dentro do escopo do componente ao qual se referem. No processo anterior, os *AwReqs* deviam ser especificados separadamente e conter um campo de referência para o elemento ao qual apontavam, como mostrado na listagem 5.3. A principal vantagem dessa modificação foi o aumento do nível de legibilidade do código do arquivo de especificação já que, como pode ser percebido em 5.3, o alvo do *AwReq*, definido pela variável "target", referenciava a linha na qual o elemento estava descrito. Assim como os refinamentos que deixaram de ser referenciados por "children", os *AwReqs* também passaram a receber uma nova forma de referenciamento: "awreqs".

Listagem 5.3 – Trecho de XML representando o ACAD no metamodelo original

```
1  <children xsi:type="scheduler:AR1" target="//@rootGoal/@children.0">
2    <condition xsi:type="it.unitn.disi.zanshin.model:SimpleResolutionCondition"/>
3    <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time="5000">
4      <condition xsi:type="it.unitn.disi.zanshin.model:
           MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="2"/>
5    </strategies>
6  </children>
```

Seguindo a mesma motivação dos elementos anteriores, a referência para *Domain Assumptions* passou de "children" para "assumptions", caracterizando assim uma separação dos diferentes tipos de refinamentos do modelo facilitando, então, tanto o processo de implementação quanto o de leitura dos arquivos de especificação.

Ao final do processo de reescrita dos arquivos dos modelos dos sistemas *Meeting Scheduler* e *A-CAD*, ambos foram importados para dentro do Eclipse™, onde o *Zanshin* é executado, e pôde-se verificar que o processo de simulação, que não teve nenhuma parte do código modificada, rodou perfeitamente com a nova versão do *Zanshin* e produziu as saídas esperadas para todos os casos.

Os arquivos completos dos modelos antigos podem ser encontrados no repositório original do *framework*, [1] enquanto as novas versões dos mesmos encontram-se em apêndice a esse texto.

5.2 Unagi

Para o processo de validação do *Unagi* foi usado método similar ao processo de validação da nova versão do *Zanshin*: após validação das alterações do *framework*, ambos sistemas *Meeting Scheduler* e *A-CAD* foram graficamente modelados no editor da ferramenta. Ao fim do processo, foi usado o conversor para criar os arquivos XML de

[1] https://goo.gl/vthgOk

especificação dos modelos de cada um dos sistemas.

Os arquivos gerados pelo *Unagi* foram então copiados para o sistema de simulação do *Zanshin* e foi executada simulação para cada um dos sistemas, comparando a saída gerada com a saída original. Verificou-se então a validade dos modelos gerados pela ferramenta ao garantir que ambas as saídas eram compatíveis.

Para fins de verificação e distribuição, os arquivos gerados pelo *Unagi* estão disponíveis como exemplos na página do respositório git da ferramenta. [2]

[2] https://github.com/hbcesar/unagi2

6 Considerações Finais

Este capítulo apresenta as conclusões do trabalho realizado, mostrando suas contribuições na Seção 6.1. Por fim, na Subseção 6.2 são apresentadas suas limitações e perspectivas de trabalhos futuros.

6.1 Conclusões

A proposta de um novo metamodelo para o *Zanshin* tem como objetivo demonstrar a importância do uso de modelos bem definidos, pois acredita-se que linguagens de modelagem foram criadas para diminuir ao máximo a ambiguidade na possíveis interpretações de comunicação de ideias. Sendo assim, o uso de metamodelos concisos e reforçados por meio do uso de definições precisas é o passo fundamental para eliminar possíveis erros de modelagem e interpretação desses modelos.

A ferramenta *Unagi* pode ser dividida em duas partes: o ambiente de modelagem e o módulo de conversão. A primeira parte adota a mesma sintaxe para especificação de modelos usada na literatura disponível sobre *Zanshin*, portanto espera-se que usuários já familiarizados com o *framework* sintam-se confortáveis ao usar o modelador. Entretanto, espera-se que novos usuários também sintam-se à vontade ao usar o *Unagi* devido a intuitividade da sintaxe adotada, bem como a simplicidade da interface gráfica do editor. Também espera-se que o módulo conversor impulsione o uso do *Zanshin* ao prover uma forma simples e fácil para criação dos arquivos de especificações dos modelos específicos.

O uso de sistemas adaptativos aparece como solução para a atual conjuntura enfrentada pela Engenharia de Software para o desenvolvimento de sistemas complexos e distribuídos. Tanto o *Zanshin* quanto o *Unagi* têm como essência o apoio ao desenvolvimento de sistemas desse tipo, pois acredita-se que esse campo de pesquisa tornar-se-á cada vez mais promissor.

6.2 Limitações e Perspectivas Futuras

Atualmente, algumas checagens de consistência e, principalmente, opções que permitam uma personalização mais refinada da sintaxe esbarram em limitações da tecnolgia do plugin Sirius^{TM}, como por exemplo opção de auto organização eficiente do modelo (atualmente essa função existe, mas não funciona corretamente). Entretanto, novas funcionalidades e melhorias vêm sendo adicionadas ao plugin constantemente e acredita-se que, em breve, muitas das atuais limitações não serão mais um problema. Em um futuro

próximo, espera-se que o módulo conversor consiga importar os arquivos gerados automaticamente para o *Zanshin*, possibilitando assim que o usuário possa criar e simular os processos do modelo específico com ainda mais facilidade e praticidade. Assim como pretende-se trabalhar continuamente na melhoria da ferramenta e na criação de regras de análise sintática, para que os modelos produzidos sejam de qualidade cada vez maior.

Referências

ACCELEO. 2012. <http://wiki.eclipse.org/Acceleo>. Acesso: 07/07/2017. Citado na página 22.

ACCELEO. 2012. <https://www.eclipse.org/acceleo/>. Acesso: 07/07/2017. Citado na página 22.

ANDERSSON, J. et al. Modeling dimensions of self-adaptive software systems. In: *Software engineering for self-adaptive systems*. [S.l.]: Springer, 2009. p. 27–47. Citado na página 12.

ATKINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. *IEEE software*, IEEE, v. 20, n. 5, p. 36–41, 2003. Citado na página 16.

BORGIDA, A. et al. Techne: A (nother) requirements modeling language. *Computer Systems Research Group. Toronto, Canada: University of Toronto*, 2009. Citado na página 25.

BRUN, Y. et al. Engineering self-adaptive systems through feedback loops. In: *Software engineering for self-adaptive systems*. [S.l.]: Springer, 2009. p. 48–70. Citado 2 vezes nas páginas 12 e 28.

BUDINSKY, F. *Eclipse modeling framework: a developer's guide*. [S.l.]: Addison-Wesley Professional, 2004. Citado na página 20.

DALPIAZ, F. et al. Runtime goal models: Keynote. In: IEEE. *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*. [S.l.], 2013. p. 1–11. Citado 2 vezes nas páginas 27 e 28.

DALPIAZ, F.; FRANCH, X.; HORKOFF, J. istar 2.0 language guide. *arXiv preprint arXiv:1605.07767*, 2016. Citado 3 vezes nas páginas 13, 24 e 25.

DARDENNE, A.; FICKAS, S.; LAMSWEERDE, A. van. Goal-directed concept acquisition in requirements elicitation. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 6th international workshop on Software specification and design*. [S.l.], 1991. p. 14–21. Citado na página 24.

DARDENNE, A.; LAMSWEERDE, A. V.; FICKAS, S. Goal-directed requirements acquisition. *Science of computer programming*, Elsevier, v. 20, n. 1-2, p. 3–50, 1993. Citado 3 vezes nas páginas 23, 24 e 25.

FALBO, R. A. *Engenharia de Software*. [s.n.], 2014. 141 p. Disponível em: <http://www.inf.ufes.br/~falbo/files/ES/Notas_Aula_Engenharia_Software.pdf>. Citado na página 22.

FALBO, R. A. *Engenharia de Software*. [s.n.], 2017. 71 p. Disponível em: <http://www.inf.ufes.br/~falbo/files/ER/Notas_Aula_Engenharia_Requisitos.pdf>. Citado na página 22.

GRUBER, T. Ontology. *Encyclopedia of database systems*, Springer, p. 1963–1965, 2009. Citado na página 24.

GUIZZARDI, G. *Ontological foundations for structural conceptual models*. [S.l.]: CTIT, Centre for Telematics and Information Technology, 2005. Citado na página 24.

GUIZZARDI, R. S. et al. An ontological interpretation of non-functional requirements. In: *FOIS*. [S.l.: s.n.], 2014. v. 14, p. 344–357. Citado na página 25.

JURETA, I.; MYLOPOULOS, J.; FAULKNER, S. Revisiting the core ontology and problem in requirements engineering. In: IEEE. *International Requirements Engineering, 2008. RE'08. 16th IEEE*. [S.l.], 2008. p. 71–80. Citado na página 24.

JURETA, I. J. et al. Core ontology for requirements engineering. *Inf. Manage. Res. Unit, Univ. Namur, Belgium, Tech Rep*, 2007. Citado na página 13.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, IEEE, v. 36, n. 1, p. 41–50, 2003. Citado na página 12.

KERN, H. The interchange of (meta) models between metaedit+ and eclipse emf using m3-level-based bridges. In: *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*. [S.l.: s.n.], 2008. v. 2008. Citado 2 vezes nas páginas 7 e 19.

LAMSWEERDE, A. V. Goal-oriented requirements engineering: A guided tour. In: IEEE. *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. [S.l.], 2001. p. 249–262. Citado na página 23.

LAMSWEERDE, A. V.; DARIMONT, R.; LETIER, E. Managing conflicts in goal-driven requirements engineering. *IEEE transactions on Software engineering*, IEEE, v. 24, n. 11, p. 908–926, 1998. Citado na página 23.

LAPOUCHNIAN, A. Goal-oriented requirements engineering: An overview of the current research. *University of Toronto*, p. 32, 2005. Citado na página 23.

MADIOT, F.; PAGANELLI, M. Eclipse sirius demonstration. In: *P&D@ MoDELS*. [S.l.: s.n.], 2015. p. 9–11. Citado na página 20.

MORIN, B. et al. Models@ run. time to support dynamic adaptation. *Computer*, IEEE, v. 42, n. 10, 2009. Citado na página 33.

MTSWENI, J. Exploiting uml and acceleo for developing semantic web services. In: IEEE. *Internet Technology And Secured Transactions, 2012 International Conference for*. [S.l.], 2012. p. 753–758. Citado na página 22.

MUSSET, J. et al. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, v. 2, 2006. Citado na página 20.

MUSSET, J. et al. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, v. 2, 2006. Citado na página 41.

NEGRI, P. P. et al. Towards an ontology of goal-oriented requirements. 2017. Citado 2 vezes nas páginas 24 e 25.

PFLEEGER, S. L. *Engenharia de software: teoria e prática*. [S.l.]: Prentice Hall, 2004. Citado na página 22.

ROSS, D. T.; SCHOMAN, K. E. Structured analysis for requirements definition. *IEEE transactions on Software Engineering*, IEEE, n. 1, p. 6–15, 1977. Citado na página 22.

SELIC, B. The pragmatics of model-driven development. *IEEE software*, IEEE, v. 20, n. 5, p. 19–25, 2003. Citado 2 vezes nas páginas 16 e 17.

SOUZA, V. E. S. *Requirements-based software system adaptation*. Tese (Doutorado) — University of Trento, 2012. Citado 10 vezes nas páginas 6, 7, 12, 25, 26, 30, 31, 33, 34 e 35.

SOUZA, V. E. S. et al. Requirements-driven software evolution. *Computer Science-Research and Development*, Springer, v. 28, n. 4, p. 311–329, 2013. Citado 3 vezes nas páginas 26, 29 e 33.

SOUZA, V. E. S.; LAPOUCHNIAN, A.; MYLOPOULOS, J. (requirement) evolution requirements for adaptive systems. In: IEEE PRESS. *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. [S.l.], 2012. p. 155–164. Citado 4 vezes nas páginas 12, 24, 27 e 28.

SOUZA, V. E. S. et al. Awareness requirements. In: *Software Engineering for Self-Adaptive Systems II*. [S.l.]: Springer, 2013. p. 133–161. Citado 3 vezes nas páginas 24, 27 e 28.

STEINBERG, D. et al. *EMF: eclipse modeling framework*. [S.l.]: Pearson Education, 2008. Citado na página 17.

VIYOVIĆ, V.; MAKSIMOVIĆ, M.; PERISIĆ, B. Sirius: A rapid development of dsm graphical editor. In: IEEE. *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. [S.l.], 2014. p. 233–238. Citado 3 vezes nas páginas 16, 17 e 20.

VUJOVIĆ, V.; MAKSIMOVIĆ, M.; PERIŠIĆ, B. Comparative analysis of dsm graphical editor frameworks: Graphiti vs. sirius. In: *23nd International Electrotechnical and Computer Science Conference ERK, Portorož, B*. [S.l.: s.n.], 2014. p. 7–10. Citado 3 vezes nas páginas 16, 18 e 20.

YU, E. et al. 1 social modeling for requirements engineering: An introduction. *Social Modeling for Requirements Engineering*, MIT Press, p. 3–10, 2011. Citado na página 24.

Apêndices

APÊNDICE A – Especificação textual do *ACAD*

Listagem A.1 – Especifcação do Sistema ACAD

```
<?xml version="1.0" encoding="UTF-8"?>
<acad:AcadGoalModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" xmlns:acad="https://raw.githubusercontent.com/hbcesar/zanshin/master/zanshin-simulations/src/it/unitn/disi/zanshin/simulation/cases/acad/acad.ecore" xmlns:eca="https://raw.githubusercontent.com/hbcesar/zanshin/master/it.unitn.disi.zanshin.core/META-INF/eca.ecore">
  <rootGoal xsi:type="acad:G_GenDispatch">
    <refinements xsi:type="acad:G_CallTaking">
      <assumptions xsi:type="acad:D_MaxCalls"/>
      <refinements xsi:type="acad:G_RegCall">
        <refinements xsi:type="acad:T_InputInfo"/>
        <refinements xsi:type="acad:T_DetectLoc"/>
        <awreqs xsi:type="acad:AR15">
          <condition xsi:type="eca:SimpleResolutionCondition"/>
          <strategies xsi:type="eca:RetryStrategy" time="5000">
            <condition xsi:type="eca:MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="1"/>
          </strategies>
          <strategies xsi:type="eca:RelaxDisableChildStrategy" child="//@rootGoal/@refinements.0/@refinements.0/@refinements.1">
            <condition xsi:type="eca:MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="1"/>
          </strategies>
        </awreqs>
      </refinements>
      <refinements xsi:type="acad:T_ConfirmCall"/>
      <refinements xsi:type="acad:G_AssignIncident">
        <refinements xsi:type="acad:T_SearchDuplic"/>
        <refinements xsi:type="acad:T_CreateOrAssign"/>
      </refinements>
    </refinements>
    <assumptions xsi:type="acad:D_DataUpd"/>
    <refinements xsi:type="acad:G_ResourceId">
      <refinements xsi:type="acad:T_SpecConfig"/>
      <refinements xsi:type="acad:T_ConfIncident"/>
    </refinements>
    <refinements xsi:type="acad:G_ResourceMob">
      <refinements xsi:type="acad:T_DetBestAmb"/>
      <refinements xsi:type="acad:T_InformStat"/>
      <refinements xsi:type="acad:G_RouteAssist" refinementType="or">
        <assumptions xsi:type="acad:D_DriverKnows"/>
        <refinements xsi:type="acad:T_AcadAssists"/>
        <refinements xsi:type="acad:T_StaffAssists"/>
```

```
37        </refinements>
38        <refinements xsi:type="acad:T_Feedback"/>
39      </refinements>
40      <refinements xsi:type="acad:G_ObtainMap" refinementType="or">
41        <assumptions xsi:type="acad:D_GazetUpd"/>
42        <refinements xsi:type="acad:G_ManualMap" refinementType="or">
43          <refinements xsi:type="acad:T_CheckGazet"/>
44          <refinements xsi:type="acad:T_CheckPaper"/>
45        </refinements>
46      </refinements>
47      <refinements xsi:type="acad:G_IncidentUpd">
48        <refinements xsi:type="acad:G_MonitorRes">
49          <refinements xsi:type="acad:G_UpdPosition" refinementType="or">
50            <assumptions xsi:type="acad:D_MDTPos"/>
51            <refinements xsi:type="acad:T_RadioPos"/>
52          </refinements>
53          <assumptions xsi:type="acad:D_MDTUse"/>
54          <refinements xsi:type="acad:T_MonitorStatus"/>
55          <refinements xsi:type="acad:T_DispStatus"/>
56          <refinements xsi:type="acad:T_DispDepArriv"/>
57          <refinements xsi:type="acad:G_DispExcept" refinementType="or">
58            <refinements xsi:type="acad:T_Except"/>
59            <refinements xsi:type="acad:T_ExceptQueue"/>
60          </refinements>
61        </refinements>
62        <refinements xsi:type="acad:T_CloseIncident"/>
63        <refinements xsi:type="acad:T_ReplAmb"/>
64      </refinements>
65
66      <!-- Softgoals. -->
67      <refinements xsi:type="acad:S_FastArriv">
68        <constraints xsi:type="acad:Q_AmbArriv"/>
69      </refinements>
70      <refinements xsi:type="acad:S_FastDispatch">
71        <constraints xsi:type="acad:Q_Dispatch">
72          <awreqs xsi:type="acad:AR11" incrementCoefficient="2">
73            <condition xsi:type="eca:ReconfigurationResolutionCondition"/>
74            <strategies xsi:type="eca:ReconfigurationStrategy" algorithmId="
                  qualia">
75              <condition xsi:type="eca:ReconfigurationApplicabilityCondition"/>
76            </strategies>
77          </awreqs>
78        </constraints>
79      </refinements>
80      <refinements xsi:type="acad:S_FastAssist">
81        <constraints xsi:type="acad:Q_IncidResolv"/>
82      </refinements>
83      <refinements xsi:type="acad:S_LowCost">
84        <constraints xsi:type="acad:Q_MaxCost"/>
85      </refinements>
86      <refinements xsi:type="acad:S_UserFriendly">
87        <constraints xsi:type="acad:Q_MaxTimeMsg"/>
88      </refinements>
89  </rootGoal>
90
91  <!-- System parameters. -->
92  <configuration>
```

```
93      <parameters xsi:type="acad:CV_MST" type="ncv" unit="10" value="60" metric="
            integer"/>
94    </configuration>
95
96    <!-- Indicator / parameter differential relations. -->
97    <relations indicator="//@rootGoal/@refinements.6/@constraints.0/@awreqs.0"
          parameter="//@configuration/@parameters.0" lowerBound="0" upperBound="180"
          operator="ft" />
98  </acad:AcadGoalModel>
```

APÊNDICE B – Especificação textual do *Meeting Scheduler*

Listagem B.1 – Especifcação do Sistema Meeting Scheduler

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <scheduler:SchedulerGoalModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI
    " xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:scheduler="https
    ://raw.githubusercontent.com/hbcesar/zanshin/master/zanshin-simulations/src/it
    /unitn/disi/zanshin/simulation/cases/scheduler/scheduler.ecore" xmlns:it.unitn
    .disi.zanshin.model="https://raw.githubusercontent.com/hbcesar/zanshin/master/
    it.unitn.disi.zanshin.core/META-INF/eca.ecore">
3   <rootGoal xsi:type="scheduler:G_SchedMeet">
4     <refinements xsi:type="scheduler:T_CharactMeet">
5       <awreqs xsi:type="scheduler:AR1">
6         <condition xsi:type="it.unitn.disi.zanshin.model:
              SimpleResolutionCondition"/>
7         <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time
              ="5000">
8           <condition xsi:type="it.unitn.disi.zanshin.model:
                MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="2"/>
9         </strategies>
10      </awreqs>
11    </refinements>
12    <refinements xsi:type="scheduler:G_CollectTime" refinementType="or">
13      <refinements xsi:type="scheduler:T_CallPartic"/>
14      <refinements xsi:type="scheduler:T_EmailPartic"/>
15      <refinements xsi:type="scheduler:G_CollectAuto">
16        <assumptions xsi:type="scheduler:D_ParticUseCal"/>
17        <refinements xsi:type="scheduler:T_CollectCal"/>
18      </refinements>
19    </refinements>
20    <refinements xsi:type="scheduler:G_FindRoom" refinementType="or">
21      <refinements xsi:type="scheduler:G_UseLocal">
22        <refinements xsi:type="scheduler:G_FindLocal" refinementType="or">
23          <refinements xsi:type="scheduler:T_GetSuggest"/>
24          <refinements xsi:type="scheduler:T_ListAvail"/>
25        </refinements>
26        <assumptions xsi:type="scheduler:D_LocalAvail"/>
27        <refinements xsi:type="scheduler:G_BookRoom" refinementType="or">
28          <refinements xsi:type="scheduler:T_UseAvail"/>
29          <refinements xsi:type="scheduler:T_CancelLess"/>
30        </refinements>
31      </refinements>
32      <refinements xsi:type="scheduler:T_CallPartner"/>
33      <refinements xsi:type="scheduler:T_CallHotel"/>
34      <awreqs xsi:type="scheduler:AR4">
35        <condition xsi:type="it.unitn.disi.zanshin.model:
              ReconfigurationResolutionCondition"/>
36        <strategies xsi:type="it.unitn.disi.zanshin.model:ReconfigurationStrategy
              " algorithmId="qualia">
```

```
37          <condition xsi:type="it.unitn.disi.zanshin.model:
                ReconfigurationApplicabilityCondition"/>
38        </strategies>
39      </awreqs>
40    </refinements>
41    <refinements xsi:type="scheduler:G_ChooseSched" refinementType="or">
42      <refinements xsi:type="scheduler:T_SchedManual"/>
43      <refinements xsi:type="scheduler:T_SchedSystem"/>
44    </refinements>
45    <refinements xsi:type="scheduler:G_ManageMeet" refinementType="or">
46      <refinements xsi:type="scheduler:T_CancelMeet"/>
47      <refinements xsi:type="scheduler:T_CancelMeet"/>
48    </refinements>
49    <refinements xsi:type="scheduler:S_LowCost">
50      <constraints xsi:type="scheduler:Q_CostLess100"/>
51    </refinements>
52    <refinements xsi:type="scheduler:S_GoodPartic">
53      <constraints xsi:type="scheduler:Q_Min90pctPart"/>
54    </refinements>
55    <refinements xsi:type="scheduler:S_FastSched">
56      <constraints xsi:type="scheduler:Q_Sched1Day"/>
57    </refinements>
58  </rootGoal>
59
60  <configuration>
61    <parameters xsi:type="scheduler:CV_RfM" unit="1" value="5" metric="integer"/>
62  </configuration>
63
64  <relations indicator="//@rootGoal/@refinements.0/@awreqs.0" parameter="//
        @configuration/@parameters.0" lowerBound="0" upperBound="10"/>
65 </scheduler:SchedulerGoalModel>
```