Unagi: Uma Ferramenta para suporte à Análise de Requisitos em Tempo de Execução

César Henrique Bernabé[1], Pedro Pignaton Negri[1], Bruno Borlini Duarte[1], André Luiz de Castro Leal[2], Vítor E. Silva Souza[1]

[1] Núcleo de Estudos em Modelagem Conceitual e Ontologias (NEMO)
Departamento de Informática, Universidade Federal do Espírito Santo (UFES)
Vitória, ES, Brasil
{cesar.hber,brunoborlini,pedropn}@gmail.com, vitorsouza@inf.ufes.br
[2] Departamento de Matemática, Universidade Federal Rural do Rio de Janeiro (UFRRJ) – Seropédica, RJ, Brasil
andrecastr@gmail.com

Resumo Sistemas adaptativos modificam seu comportamento em tempo de execução para satisfazer seus requisitos mesmo em caso de falha. *Zanshin* é uma abordagem baseada em Engenharia de Requisitos Orientada a Objetivos (*Goal-Oriented Requirements Engineering* ou simplesmente *GORE*) para o desenvolvimento de sistemas adaptativos. Para apoiar o processo de modelagem dos requisitos dentro desta abordagem, este artigo apresenta a ferramenta *Unagi*, que provê um editor gráfico para criação de modelos *GORE* compatíveis com *Zanshin*.
Palavras-chave: requisitos, objetivos, modelos, tempo de execução, sistemas adaptativos

1 Introdução

Com o avanço da tecnologia dos computadores nas últimas décadas, identificou-se um aumento relativo no que se refere à complexidade de sistemas de software [1], visto que o crescimento da capacidade computacional propiciou maiores possibilidades na área de programação [3]. Isso motivou pesquisadores e desenvolvedores a criarem novas técnicas, baseadas principalmente em intervenção humana [1] para projetar e gerenciar softwares abrangendo os processos de projeto, construção e teste, e garantindo assim a estabilidade dos sistemas.

Porém, além de tornarem-se mais robustos, os aplicativos foram tornando-se cada vez mais escaláveis e, portanto, altamente distribuíveis, o que tornou o gerenciamento manual dos mesmos um processo praticamente inviável [1]. Nesse contexto [8], destaca-se o uso de sistemas adaptativos como uma solução viável para este problema. Do mesmo modo, à medida que sistemas vão se tornando mais enraizados no nosso dia-a-dia, acabam encontrando os mais diversos contextos de execução e, assim, faz-se importante a elaboração de softwares que possam identificar quando seus requisitos não estão sendo atendidos e, então, adaptarem-se em tempo de execução a essas situações.

Sistemas adaptativos podem assumir diferentes comportamentos de acordo com alterações de contexto, mediante falhas ou mudanças nos requisitos de desempenho [13]. Porém, poucas soluções desse tipo consideram a modelagem das características adaptativas do sistema desde a fase de Engenharia de Requisitos no processo de software. Dentro desse escopo, destacamos o *Zanshin* [12], uma abordagem que baseia-se em modelos de requisitos para projetar características adaptativas em sistemas.

Neste artigo, apresentamos o *Unagi*, uma ferramenta desenvolvida no contexto do *Zanshin*, que provê um ambiente gráfico para modelagem de requisitos usando Engenharia de Requisitos Orientada a Objetivos (*Goal-Oriented Requirements Egineering* ou *GORE*). Esse artigo apresenta os aspectos importantes da ferramenta desenvolvida nesse contexto, bem como uma breve ilustração dos elementos de *GORE* nela usados. Ao final, são dadas instruções de instalação e uso da mesma.

O restante deste artigo está assim dividido: na Seção 2 é resumida a abordagem *Zanshin*; a Seção 3 apresenta o *Unagi* e suas instruções de uso e instalação; na Seção 4 é realizada uma avaliação da ferramenta; na Seção 5 comparamos trabalhos relacionados e, por fim, a Seção 6 conclui o artigo.

2 Referencial Teórico

Zanshin [12] é uma abordagem para desenvolvimento de sistemas adaptativos que permite que se especifique, no modelo de requisitos, em quais situações o sistema deve adaptar-se e o que deve fazer para isso. *Zanshin* parte do princípio que sistemas adaptativos possuem, ainda que implícitos ou escondidos, um ciclo de retroalimentação (*feedback loop*) que operacionaliza a adaptação. A proposta principal do método é fazer com que os elementos desse ciclo se tornem explícitos nos modelos de requisitos utilizados.

Para isso, *Zanshin* utiliza-se de modelos de requisitos baseados na Engenharia de Requisitos Orientada a Objetivos (*GORE*) para representar os requisitos do sistema, adicionando novos elementos que, quando combinados com elementos clássicos desse paradigma (e.g., objetivos, tarefas, refinamentos, etc.), são capazes de representar estratégias de monitoramento e adaptação que podem ser disparadas para que o sistema em execução possa reagir a situações críticas. Esses elementos possuem ênfase nos objetivos que precisam ser resolvidos pelo sistema e, assim, passam a guiar o software quando este precisa iniciar um processo de adaptação.

GORE baseia a elaboração do modelo de requisitos de software no conceito de objetivo [16]. A Figura 1 ilustra os elementos comumente utilizados em modelos de objetivos, bem como alguns conceitos introduzidos pela abordagem *Zanshin*. Objetivos (*goals*, representados pelas ovais) são estados do mundo que deseja-se alcançar e podem ser refinados até que sejam operacionalizados por tarefas (*tasks*, hexágonos) que o sistema irá executar ou por pressuposições de domínio (*domain assumptions*, retângulos). Objetivos que não possuem em si critérios exatos de satisfação, denominados *softgoals* (nuvens), são operacionalizados por

critérios de qualidade (*quality constraints*, retângulos com cantos arredondados). Refinamentos utilizam lógica Booleana (*AND* ou *OR*) para calcular a satisfação de um elemento "pai" com base nos elementos "filhos".

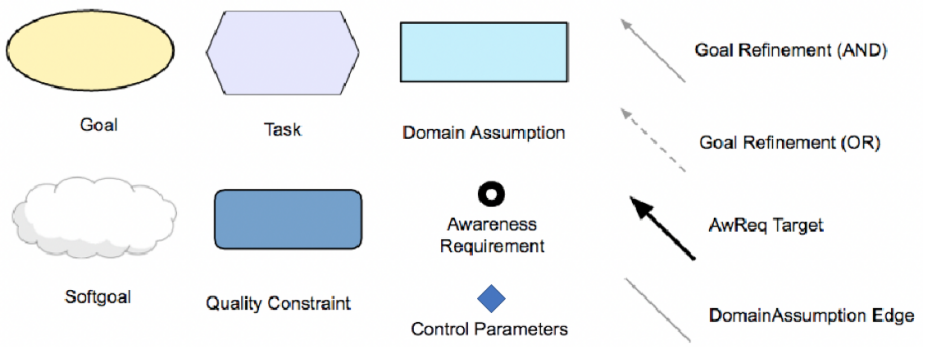

Figura 1. Elementos de um modelo *GORE* e sua representação gráfica.

Além dos elementos tradicionais de *GORE*, os elementos propostos em *Zanshin* são os Requisitos de Percepção (*Awareness Requirements* ou *AwReqs*, círculos) [14] e os Requisitos de Evolução (*Evolution Requirements* ou *EvoReqs*, sem representação gráfica) [13]. *AwReqs* são requisitos que se referem ao estado assumido por outros requisitos em tempo de execução e são representados por círculos que apontam para o elemento monitorado. *EvoReqs* descrevem como outros requisitos no modelo devem mudar/evoluir em resposta à falha de um *AwReq*. Em outras palavras, *AwReqs* representam situações nas quais os stakeholders desejam que o sistema seja capaz de adaptar-se, enquanto *EvoReqs* atuam diretamente nos requisitos do sistema por meio de estratégias de adaptação, sendo responsáveis por garantir que o sistema continue a cumprir o que foi especificado pelos stakeholders [13]. Por fim, foram adicionados os parâmetros de controle (*control parameters*, losangos) que representam parâmetros do sistema que podem ser reconfigurados para adaptação.

A Figura 2, mostra um exemplo de modelo de requisitos para um sistema de agendamento de reuniões (*meeting scheduler*), um sistema responsável por coletar os quadros de horários dos participantes de uma reunião a ser marcada, encontrar uma sala disponível, reservá-la e, por fim, confirmar a presença dos participantes. Nesse contexto, os conceitos de *GORE* são usados para definir o escopo do problema: um objetivo pode ser definido como o estado no qual reuniões são agendadas corretamente e tarefas representam os passos seguidos pelo sistema para cumprir este objetivo, como enviar e-mails aos participantes e efetuar o agendamento da reunião, enquanto pressuposições de domínio representam situações necessárias que se acredita que sejam verdadeiras, ex.: disponibilidade de salas para a reunião. Objetivos sem um critério claro de satisfação (*softgoals*), como "boa participação", possuem critérios de qualidade associados que deter-

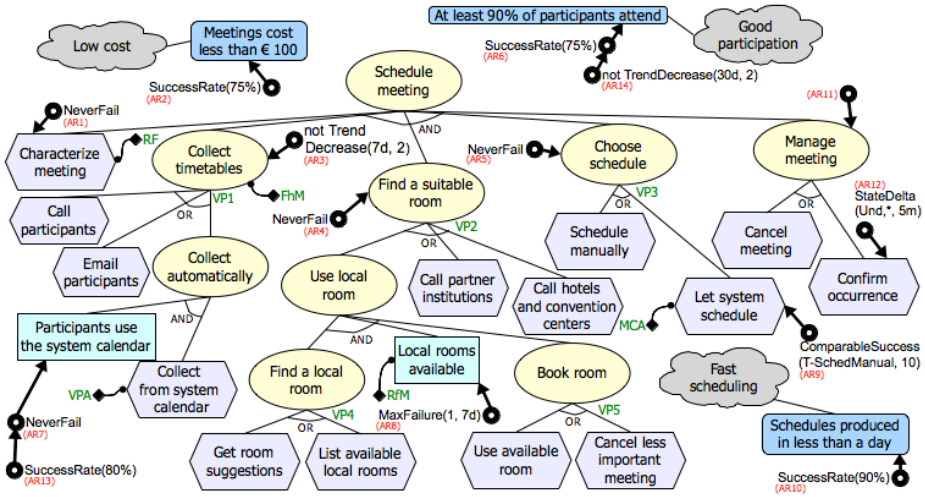

Figura 2. Exemplo de modelo de requisitos para um sistema adaptativo de agendamento de reuniões (*meeting scheduler*), retirado de [12].

minam quando são satisfeitos ou não, ex.: "pelo menos 90% dos participantes presentes". Qualquer um desses elementos podem ser monitorados pelos requisitos de percepção para garantir que os mesmos sejam atingidos. Por exemplo, o critério de qualidade "Agendamento é realizado em menos de um dia" e deve ter uma taxa de sucesso mínima de 90%.

Além do modelo de objetivos, o *Zanshin* requer que o sistema alvo seja configurado para proporcionar um registo (*log*) que indique quando os diferentes elementos do modelo *GORE* mudaram de estado (ex.: tarefa em andamento, critério de qualidade satisfeito/não satisfeito, etc.). A partir dessas ocorrências e da especificação dos *AwReqs*, um componente **Monitor** é capaz de identificar a alteração de estado previamente registrado e então desencadear um componente **Adaptador** que decide quais as operações de adaptação o sistema destino deve executar. Este componente pode ser dividido em duas partes principais: (1) um componente de adaptação baseado no padrão Evento-Condição-Ação (ECA), com possibilidades de escolha de estratégia de adaptação com base na lista de estratégias associada à falha do *AwReq* e às suas respectivas condições de aplicabilidade; (2) um componente de reconfiguração qualitativa (chamado *Qualia*), que é ativado pelo processo ECA quando a reconfiguração é selecionada como a estratégia adequada a ser aplicada. A saída é uma lista de instruções de adaptação que são enviadas ao sistema base [13]. Este último, por sua vez, deve ter implementadas ações baseadas nas instruções enviadas por *Zanshin* [15].

Os modelos de objetivos que são usados pelo *Zanshin* devem ser especificados usando a plataforma *Eclipse Modeling Framework* (*EMF*). Atualmente este processo é feito através da criação de elementos em um editor gerado automaticamente pelo próprio EMF, baseado no metamodelo de *Zanshin*, que exibe os

elementos do modelo de uma forma genérica, similar a uma árvore de diretórios em um gerenciador de arquivos. O *Unagi*, apresentado na próxima seção, vem substituir tal editor genérico por uma ferramenta mais amigável.

A tecnologia *EMF* permite que seja gerado código Java™ automaticamente a partir de metamodelos. Tal código contém instâncias de classes dos metamodelos [2], bem como a especificação de hierarquia entre essas classes. O código gerado pode ser usado para, dentre outras coisas, definir esquemas de representações gráficas para esses metamodelos sendo, por exemplo, possível criar editores gráficos para modelos de domínios específicos [17], usando plug-ins como o Graphiti[3] ou o Sirius [18]. Este último foi usado para criação da ferramenta gráfica proposta por este artigo.

O framework *Zanshin* encontra-se disponível para download e uso na plataforma de controle de versão Github. Em sua wiki,[4] o leitor interessado encontrará informações para execução das simulações de sistemas adaptativos, inclusive o mostrado na Figura 2, o *Meeting Scheduler*.

3 O Editor Gráfico *Unagi*

Baseado na tecnologia Sirius [17], que permite definir representações gráficas para elementos de modelos especificados em *EMF*, o *Unagi* traz uma interface gráfica para criação de modelos baseados em *GORE*, provendo assim suporte ao desenvolvimento de modelos de requisitos ao *Zanshin*. A Seção 3.1 descreve o uso da ferramenta, enquanto a Seção 3.2 apresenta sua arquitetura e a Seção 3.3 explica como instalar o *Unagi*.

3.1 Uso da ferramenta

Para exemplificar o uso da ferramenta, apresentamos como estudo de caso o sistema agendador de reuniões (*meeting scheduler*), que ilustra a proposta do *Zanshin* em [12] e cujo modelo de requisitos é exibido na Figura 2 e descrito na Seção 2. O *meeting scheduler* foi escolhido por ser um exemplo conhecido na comunidade de Engenharia de Requisitos [9] e para que pudéssemos comparar a saída produzida pelo *Unagi* com os arquivos utilizados pelo *Zanshin* na simulação do sistema, disponível no repositório de controle de versões do framework.[5]

A Figura 3 mostra a interface principal do *Unagi*. A partir desta tela, o modelador pode incluir elementos do modelo simplesmente arrastando o item desejado da paleta de componentes para a região do diagrama, bem como definir as associações entre os componentes selecionando o tipo de relação disponível na mesma paleta e clicando nos elementos origem e destino, nesta ordem.

Na Figura 3, todos os elementos do modelo original do *meeting scheduler* foram replicados no *Unagi*. As características de cada componente foram definidas

[3] `https://eclipse.org/graphiti/`.
[4] `https://github.com/sefms-disi-unitn/Zanshin/wiki`.
[5] `https://github.com/sefms-disi-unitn/Zanshin/wiki/Creating-a-simulation,-part-1.-the-meta-model`.

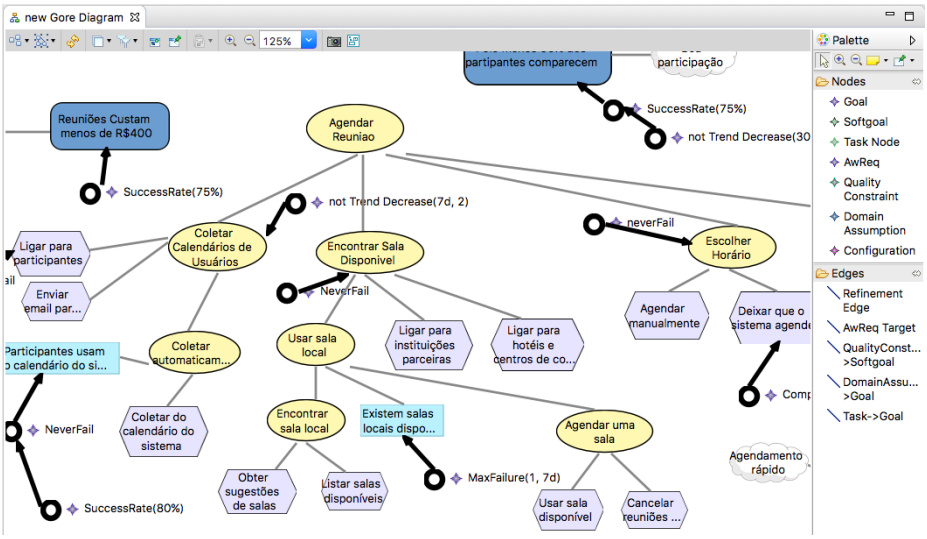

Figura 3. Interface principal do *Unagi* durante a construção do modelo *GORE* do sistema de agendamento de reuniões.

por meio da paleta de opções mostrada na parte inferior da tela (Figura 4). Essa paleta mostra configurações específicas para o elemento selecionado, dependendo do seu tipo. Nela é possível definir características como nome, alvo, objetivo pai, dentre outros.

Os *EvoReqs* inerentes a cada *AwReq* são especificados por meio de uma funcionalidade particular a esse tipo: ao clicar sobre um *AwReq*, o editor oferece a opção de criação de um subdiagrama de detalhamento dos elementos, exemplificado na Figura 5. Assim, o usuário pode determinar as estratégias de adaptação próprias a cada *AwReq* usando a mesma lógica de arrastar e soltar, selecionando as condições de resolução, estratégias de adaptação e condições de aplicabilidade na paleta específica deste tipo de diagrama, na parte direita da tela.

Construído o modelo gráfico, o usuário pode acionar o conversor do *Unagi*, que transforma os arquivos criados pelo *Sirius* para a especificação do diagrama em arquivos *XML* que podem ser lidos pelo *Zanshin*. Para executar o processo de conversão, o usuário deve clicar com o botão direito em qualquer área livre do diagrama e selecionar "Converter → Convert Diagram".

Deste modo, o usuário pode importar o arquivo no *Zanshin* e rodar as simulações a partir do diagrama construído, ilustrando a principal proposta do *Unagi*: facilitar a criação de modelos *GORE* para uso no *Zanshin*. Inicialmente só era possível criar esses modelos por meio do editor *EMF*, tarefa que poderia ser trabalhosa e complicada para usuários que não tivessem experiência com esse procedimento, já que além de demandar um trabalho maior para criação, fica difícil visualizar de forma trivial as relações e hierarquias entre os elementos,

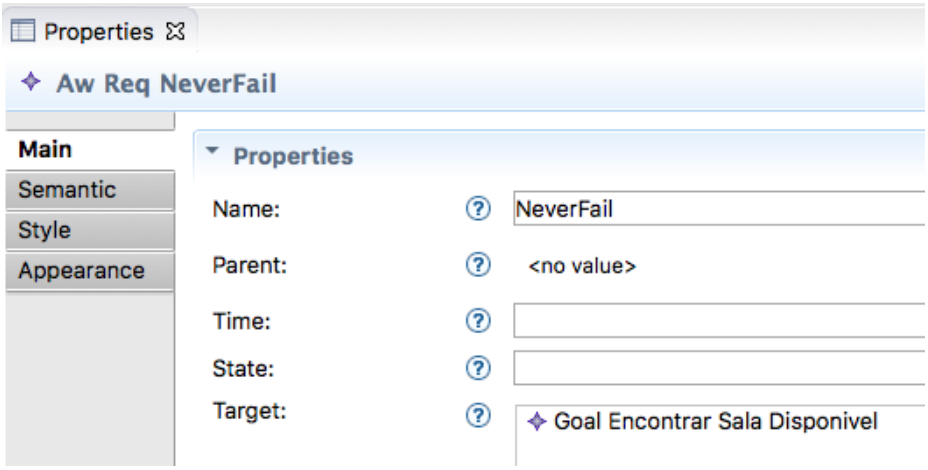

Figura 4. Interface principal do *Unagi* mostrando exemplo de paleta de opções que é exibida de acordo com o elemento selecionado.

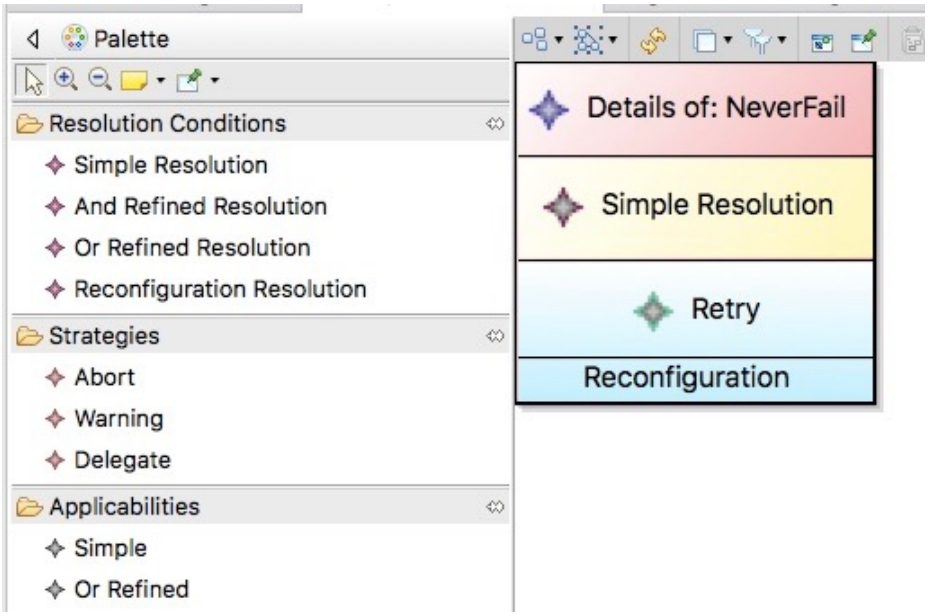

Figura 5. Interface para especificação dos *EvoReqs* de um *AwReq*.

pois todos são representados em uma lista simples. Como efeito, isso poderia desencorajar desenvolvedores de sistemas adaptativos a usarem o *Zanshin*.

O arquivo XML gerado pelo conversor para o caso de uso utilizado encontra-se disponível no repositório da ferramenta,[6] assim como o arquivos XML da simulação original do *Zanshin* também se encontram no repositório do framework.[7] O leitor interessado pode comparar os arquivos para verificar que são, de fato, compatíveis.

Por fim, é importante salientar que, apesar de fornecer uma série de facilidades, o uso do *Sirius* para especificação de metamodelos com vistas à criação de modelos gráficos pode ser uma tarefa complexa dependendo do domínio do problema, como pode ser visto em tutoriais providos pela ferramenta.[8] Porém, a escolha da plataforma Eclipse™ e seus plug-ins para geração de uma ferramenta CASE para a abordagem *Zanshin* se deu pelo fato de ambos serem baseados em *EMF*. O uso do *Sirius*, portanto, facilitou a manipulação dos metamodelos existentes, além da geração dos arquivos com as instâncias dos modelos desenhados dentro da ferramenta.

3.2 Arquitetura da ferramenta

O *Unagi* foi construído em cima do plug-in *Sirius*, que por sua vez tem como base as ferramentas de modelagem da plataforma Eclipse™ (dentre as quais se destaca o *Eclipse Modelling Framework*, ou *EMF*), baseada em Java™. A Figura 6 descreve a arquitetura da ferramenta.

Seguindo o paradigma de Desenvolvimento Orientado a Modelos [5], a construção de uma ferramenta de modelagem gráfica utilizando o *Sirius* passa pela definição de dois modelos relacionados ao domínio em questão (em nosso caso, GORE). O modelo da sintaxe abstrata, ou metamodelo, define os elementos que poderão ser criados no modelador gráfico, bem como suas propriedades e relações. O modelo da sintaxe concreta, chamado pelo *Sirius* de *viewpoint specification*, define as características gráficas dos elementos definidos no metamodelo. Ambos os modelos são especificados utilizando a sintaxe do *EMF*.

A Figura 7 mostra parte do metamodelo que define a sintaxe abstrata do editor gráfico *Unagi* (as hierarquias de AdaptationStrategy, ResolutionCondition e ApplicabilityCondition foram omitidas por serem muito extensas). Tal modelo é compatível com o utilizado por *Zanshin* para analisar os requisitos das aplicações em tempo de execução e prover instruções de adaptação.

Os elementos GORE, juntamente com os conceitos propostos em *Zanshin*, são representados pelas meta-classes Goal, Softgoal, Task, DomainAssumption e AwReq, organizados em uma hierarquia de requsitos (Requirement). As relações entre os elementos são representadas pelas associações entre os mesmos: objetivos

[6] `https://github.com/hbcesar/unagi2/tree/master/examples`.

[7] `https://github.com/sefms-disi-unitn/Zanshin/tree/master/zanshin-simulations/src/it/unitn/disi/zanshin/simulation/cases/scheduler`.

[8] `https://wiki.eclipse.org/Sirius/Tutorials/AdvancedTutorial`.

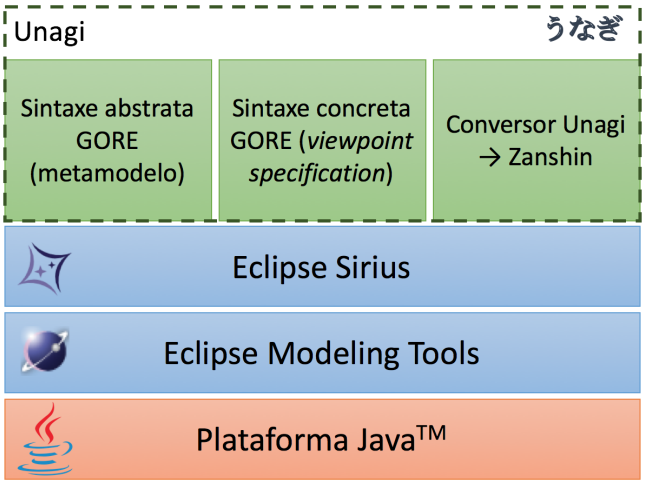

Figura 6. Descrição da arquitetura da ferramenta.

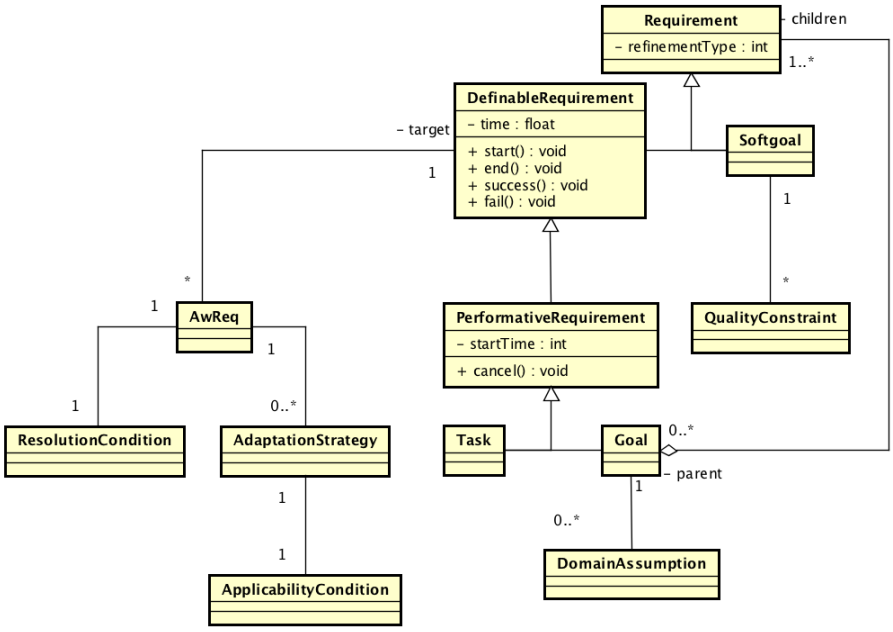

Figura 7. Metamodelo que define a sintaxe abstrata para o editor gráfico *Unagi*.

podem ser refinados (AND/OR) em outros requisitos ou em pressuposições de domínio, Softgoals são refinados em critérios de qualidade e AwReqs se referem a requisitos que podem ser monitorados.

A parte do modelo relativa ao subdiagrama de detalhamento dos *EvoReqs* associados a um *AwReq* em particular compreende ainda as meta-classes AdaptationStrategy, ResolutionCondition, ApplicabilityCondition e suas relações com a meta-classe AwReq. Um *AwReq* pode estar associado a várias estratégias de adaptação (*EvoReqs*, na prática), cada qual com sua condição de aplicabilidade. Para determinar se uma adaptação resolveu o problema, verifica-se a condição de resolução associada ao *AwReq*.

O último componente do *Unagi* é o conversor responsável por transformar o modelo gráfico em um arquivo no formato compatível com o *Zanshin*, também baseado em *EMF*. O conversor foi construído utilizando a linguagem Java™ com o apoio de bibliotecas de manipulação de arquivos XML, como as bibliotecas do pacote javax.xml.parsers. Por fim, tem-se a integração do editor gráfico com o conversor por meio da implementação de uma Chamada Java Externa (*External Java Call*), uma funcionalidade do plug-in *Sirius* que permite a invocação de serviços externos contendo código escrito em Java™.

3.3 Instruções de Instalação

O *Unagi* é distribuído na forma de um *plugin* para o *Eclipse* e pode ser instalado adicionando-o a esta IDE em sua distribuição *Eclipse Modeling Tools*, criando um *Gore Model* dentro de *Modeling Project* e configurando a *Viewpoints Selection* do Sirius. Um passo a passo deste processo encontra-se publicado na wiki do *Unagi* no Github: `https://github.com/hbcesar/unagi2/wiki/Instrucoes-de-Instalacao`. Ao fim das etapas descritas, é possível criar novos diagramas e utilizar as facilidades providas pelo *Unagi*.

4 Avaliação

Para avaliar o funcionamento do *Unagi*, usamos o editor para modelar os sistemas utilizados originalmente em [12] na avaliação do próprio framework *Zanshin*, a saber:

- O sistema de agendamento de reuniões, já apresentado nas seções 2 e 3;
- Um sistema de caixa eletrônico (*ATM Machine*), que simula, o comportamento de um caixa eletrônico de banco e, com o objetivo de provocar processos de adaptação, também simula erros de sistema e hardware [15];
- Um sistema de despacho de ambulâncias apoiado por computador (*A-CAD*), voltado para apoiar a emissão de ambulâncias baseado em disponibilidade e distância, tornando o processo mais eficiente.

Todos os modelos foram desenhados com facilidade no editor gráfico provido pelo *Unagi* e, em seguida, exportados para o formato XML do *EMF*, conforme

processo descrito na Seção 3. Os arquivos XML gerados pelo *Unagi* podem ser encontrados na página da ferramenta no Github.[9]

Em seguida, os arquivos XML foram comparados com as versões criadas para a avaliação do *Zanshin*, na qual foi utilizado o editor gerado automaticamente pelo *EMF* (vide Seção 2). Neste passo verificamos que os modelos eram equivalentes. O leitor interessado pode repetir a comparação obtendo os modelos originais na página do *Zanshin*.[10]

Finalmente, os arquivos XML gerados pelo *Unagi* foram utilizados diretamente no framework *Zanshin* para execução das simulações dos três sistemas elencados acima, de modo a verificar que o editor proposto neste artigo é compatível com o framework de adaptação. Instruções para repetição de tais simulações encontram-se também na página do *Zanshin* no Github.[11]

5 Trabalhos relacionados

A ferramenta *GATO* [11] também permite gerar arquivos *EMF* para o *Zanshin* a partir do modelo criado pelo usuário em uma ferramenta gráfica. *GATO* é uma ferramenta Web, ao contrário do *Unagi*, que é voltado para *desktop*. Ambas trabalham com uma linguagem *GORE* e suportam as extensões de adaptação propostas por *Zanshin*. Por ser construída na plataforma Web, *GATO* pode parecer mais simples e intuitiva a priori. O *Unagi*, porém, permite que o utilizador crie subdiagramas para descrever os elementos envolvidos no ciclo de retroalimentação (de adaptação), o que permite um melhor gerenciamento de diagramas de grandes proporções. Em *GATO*, as propriedades dos elementos de adaptação e evolução devem ser especificados na forma de tabela simples. Ademais, os parâmetros recorrentes a cada *AwReq* são específicos e, portanto, em *Unagi* estão representados na paleta de elementos, o que restringe o usuário ao correto uso dos mesmos, enquanto *GATO* oferece apenas uma caixa de texto na qual os parâmetros devem ser especificados, aumentando as chances de erros devido a especificação incorreta, seja por erro de digitação ou por indicação de um parâmetro não existente. Por fim, sendo baseada no Eclipse *EMF*, o *Unagi* pode se beneficiar das vantagens do desenvolvimento dirigido por modelos (*Model Driven Development*) [5].

Existem algumas ferramentas de modelagem *GORE* de propósito geral (i.e., não direcionadas a construção de sistemas adaptativos) mas, ainda assim, relacionadas. *TAOM4e* [10] visa apoiar a elicitação e modelagem de requisitos baseada em *Tropos* [6]. Por englobar o processo de elicitação de requisitos até a analise em tempo de execução, é justo que o *TAOM4e* seja comparado ao conjunto *Unagi* + *Zanshin*. A principal diferença entre eles é que enquanto *TAOM4e* é baseada em *Tropos*, *Unagi* + *Zanshin* não são baseados em nenhuma linguagem

[9] `https://github.com/hbcesar/unagi2/tree/master/examples`.

[10] `https://github.com/sefms-disi-unitn/Zanshin/tree/master/zanshin-simulations/src/it/unitn/disi/zanshin/simulation/cases`.

[11] `https://github.com/sefms-disi-unitn/Zanshin/wiki/How-to-run-Zanshin-simulations`.

específica, mas nos conceitos comuns às várias abordagens *GORE*. Assim como *Unagi*, *TAOM4e* é implementado como um plug-in do Eclipse™ que fornece facilidades para implementação de interfaces gráficas para criação de modelos, porém utilizando o plug-in GEF[12] ao invés do Sirius. As principais vantagens da ferramenta são o fato de prover design e análise de requisitos na mesma interface. Entretanto, *Zanshin* + *Unagi* proveem além de uma ferramenta de modelagem, um recurso de monitoramento e adaptação em tempo de execução.

A ferramenta *OpenOME* [7] é voltada ao desenvolvimento de modelos baseados na sintaxe de $i^\star$ [19] e também é fundamentada em *EMF*, essa ferramenta possui interface muito semelhante ao *Unagi*, como os recursos de arrastar elementos a partir de uma paleta de opções, bem como apresenta um processo de instalação e configuração muito parecidos. Além disso, possui funcionalidades que permitem uma conexão entre os processos de desenvolvimento de requisitos e as fases de desenvolvimento da arquitetura. Entretanto, destoa da proposta principal do *Unagi*, pois não oferece suporte a modelagem de sistemas adaptativos, como os elementos do ciclo de retroalimentação propostos por *Zanshin* [13] e usados em *Unagi*.

6 Conclusões

Os aspectos importantes da ferramenta *Unagi* podem ser divididos em dois: a interface de modelagem e o conversor para arquivos escritos com sintaxe adotada por *Zanshin* [12]. Na primeira, temos como principal ponto positivo o fato de a interface focar inteiramente nos elementos de *GORE* e nos elementos de adaptação, adotando a mesma simbologia proposta pelo framework. Portanto, espera-se que usuários já familiarizados com *Zanshin* sintam-se confortáveis com a ferramenta em seus primeiros momentos de uso e que novos usuários possam rapidamente se habituar tanto com o *framework* quanto com a ferramenta *CASE*.

Outro ponto forte é o fato de a interface separar os tipos de ligações que podem ser feitas entre elementos, assim evita-se que elementos que não podem se relacionar sejam erroneamente conectados. Por exemplo, é sabido que um *softgoal* não pode ser refinado em um *AwReq*, e nesse caso o usuário nunca poderá fazer esse tipo de conexão, pois não existe um conector do tipo "Softgoal → AwReq". Por fim, propriedades inerentes a cada elemento são definidas na própria tela de modelagem, evitando que o usuário tenha que trocar abas do modelador a todo momento, como acontece em algumas ferramentas do tipo.

O conversor integrado à ferramenta tem, como único objetivo, transformar o modelo atual e todos os submodelos de *AwReqs* em arquivos *XML* com sintaxe específica para uso no *Zanshin*. Tal funcionalidade facilita bastante o processo de especificação do modelo de objetivos para uso com o *framework*.

Embora o *Unagi* seja uma ferramenta funcional para criação de modelos de requisitos para sistemas adaptativos, ele ainda carece de melhoramentos para promover a Engenharia de Requisitos Orientada a Objetivos e a modelagem de

[12] `https://eclipse.org/gef/`.

sistemas adaptativos com *Zanshin*. Em um futuro breve, planeja-se integrar de forma efetiva todos os componentes do *Zanshin* e do *Unagi*, disponibilizando, assim, um único aplicativo que permita que o usuário chame o framework direto do editor de modelos ao fim da edição do diagrama (atualmente ele precisa executar o *Zanshin* em uma instância separada).

Outro trabalho futuro consiste em revisar os meta-modelos do *Unagi* (e, consequentemente, do *Zanshin*), baseando-os em ontologias bem fundamentadas como a *Runtime Requirements Ontology* [4] e a *Goal-Oriented Requirements Ontology* (em construção). Tal trabalho visa garantir ainda mais que a ferramenta promova a construção de modelos válidos e que façam sentido no mundo real.

Por fim, melhorias relacionadas à validação do diagrama em tempo de criação também deverão ser adicionadas, com o objetivo de tornar o processo de modelagem cada vez mais simples e intuitivo, bem como a implementação de diferentes tipos de visualização do mesmo diagrama, com o objetivo de ajudar a lidar com complexidades de modelos maiores.

Conforme mencionamos no início do artigo, 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. O *Unagi* tem como essência o apoio ao desenvolvimento de sistemas desse tipo, pois acredita-se que esse campo de pesquisa tornar-se-à cada vez mais promissor.

Agradecimentos

O NEMO (`http://nemo.inf.ufes.br`) é atualmente apoiado pelas agências de fomento brasileiras FAPES (# 0969/2015) e CNPq (# 485368/2013-7, # 461777/2014-2). O desenvolvimento dessa ferramenta teve início no contexto do Programa Institucional de Iniciação Científica da Universidade Federal do Espírito Santo, com apoio de recursos da CAPES.

Referências

1. Andersson, J., De Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Software engineering for self-adaptive systems, pp. 27–47. Springer (2009)
2. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the eclipse modeling framework. In: Model Driven Engineering Languages and Systems. pp. 425–439. Springer (2006)
3. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Software engineering for self-adaptive systems, pp. 48–70. Springer (2009)
4. Duarte, B.B., Souza, V.E.S., Leal, A.L.d.C., Falbo, R.A., Guizzardi, G., Guizzardi, R.S.S.: Towards an Ontology of Requirements at Runtime. In: Proc. of the 9th International Conference on Formal Ontology in Information Systems. vol. 283, pp. 255–268. IOS Press, Annecy, France (jul 2016)
5. Embley, D.W., Liddle, S.W., Pastor, O.: Conceptual-Model Programming: A Manifesto. In: Embley, D.W., Thalheim, B. (eds.) Handbook of Conceptual Modeling, pp. 3–16. Springer Berlin Heidelberg (2011)

6. Giorgini, P., Mylopoulos, J., Sebastiani, R.: Goal-oriented requirements analysis and reasoning in the Tropos methodology. Engineering Applications of Artificial Intelligence 18(2), 159–171 (mar 2005)
7. Horkoff, J., Yu, Y., Eric, S.: OpenOME: An Open-source Goal and Agent-Oriented Model Drawing and Analysis Tool. iStar 766, 154–156 (2011)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
9. van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. In: Proc. of the 2nd IEEE International Symposium on Requirements Engineering. pp. 194–203. IEEE, York, England (mar 1995)
10. Morandini, M., Nguyen, D.C., Perini, A., Siena, A., Susi, A.: Tool-supported development with tropos: The conference management system case study. In: International Workshop on Agent-Oriented Software Engineering. pp. 182–196. Springer (2007)
11. Pimentel, J.H.C.: Systematic design of adaptive systems: control-based framework. Ph.D. thesis, Universidade Federal de Pernambuco, Pernambuco, PE, Brasil (2015)
12. Souza, V.E.S.: Requirements-based Software System Adaptation. Phd thesis, University of Trento, Italy (2012)
13. Souza, V.E.S., Lapouchnian, A., Mylopoulos, J.: (Requirement) evolution requirements for adaptive systems. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 155–164. IEEE Press (2012)
14. Souza, V.E.S., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements. In: Software Engineering for Self-Adaptive Systems II, pp. 133–161. Springer (2013)
15. Tallabaci, G., Souza, V.E.S.: Engineering Adaptation with Zanshin: an Experience Report. In: Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 93–102. IEEE (may 2013)
16. Van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on. pp. 249–262. IEEE (2001)
17. Viyovic, V., Maksimovic, M., Perisic, B.: Sirius: A rapid development of DSM graphical editor. In: Intelligent Engineering Systems (INES), 2014 18th International Conference on. pp. 233–238. IEEE (2014)
18. Vujović, V., Maksimović, M., Perišić, B.: Comparative analysis of DSM Graphical Editor frameworks: Graphiti vs. Sirius. In: 23nd International Electrotechnical and Computer Science Conference. pp. 7–10 (2014)
19. Yu, E.S.: Towards modelling and reasoning support for early-phase requirements engineering. In: Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on. pp. 226–235. IEEE (1997)