UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO

Iagoh Ribeiro Lima

Criação de uma DSL e Integração das Ferramentas *Unagi* e *Zanshin* para Apoiar o Desenvolvimento de Sistemas Adaptativos baseados em Requisitos

Vitória, ES

2021

Iagoh Ribeiro Lima

Criação de uma DSL e Integração das Ferramentas *Unagi* e *Zanshin* para Apoiar o Desenvolvimento de Sistemas Adaptativos baseados em Requisitos

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

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Engenharia de Computação

Orientador: Prof. Dr. Vítor E. Silva Souza

Coorientador: Me. César Henrique Bernabé

Vitória, ES

2021

Iagoh Ribeiro Lima

Criação de uma DSL e Integração das Ferramentas *Unagi* e *Zanshin* para Apoiar o Desenvolvimento de Sistemas Adaptativos baseados em Requisitos/ Iagoh Ribeiro Lima. – Vitória, ES, 2021-

52 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Vítor E. Silva Souza

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Colegiado do Curso de Engenharia de Computação, 2021.

1. Engenharia de Requisitos. 2. Zanshin DSL. I. Iagoh Ribeiro Lima. II. Universidade Federal do Espírito Santo. III. Criação de uma DSL e Integração das Ferramentas *Unagi* e *Zanshin* para Apoiar o Desenvolvimento de Sistemas Adaptativos baseados em Requisitos

CDU 02:141:005.7

Iagoh Ribeiro Lima

Criação de uma DSL e Integração das Ferramentas *Unagi* e *Zanshin* para Apoiar o Desenvolvimento de Sistemas Adaptativos baseados em Requisitos

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

Vitória, ES
2021

Dedico este trabalho aos meus pais e à minha família, em especial minha mãe e minha namorada que não soltaram minha mão em momento algum durante essa jornada.

Agradecimentos

Primeiramente, agradeço aos meus pais, Leuzeni e Lucimar, que mesmo com pouca instrução acadêmica, desde pequeno me mostraram que a educação era o melhor caminho a se percorrer. Aos meus irmãos Gustavo, Lara e Luma, por confiarem no meu potencial e sempre acreditarem que eu poderia ir além. Mais uma vez, aos meus pais e irmãos, meu muito obrigado, vocês foram essenciais em toda a jornada da minha vida.

À minha companheira, Nathália Luiza, que foi força onde eu achei que não existia mais. Me acompanhando desde o início da jornada da graduação, viveu todos os momentos ao meu lado me dando suporte e, quando mais precisei, soube abrir mão do nosso tempo juntos, com muito carinho e confiança, para que eu pudesse alcançar voos mais altos. Você é a razão pela qual acordo todos os dias e enfrento as dificuldades que aparecem de cabeça erguida. Obrigado por sempre me guardar em seu abraço e me incentivar a ir mais longe. Nunca se esqueça que você é meu maior motivo de orgulho e você me inspira a ser alguém melhor. Obrigado por tanto.

A todos os meus familiares, sou muito grato por toda força que me deram desde criança. Em especial, agradeço à prima Suellen por estar comigo desde sempre, sem largar minha mão, compartilhando o amor pela tecnologia e me acolhendo com todo o conforto em seu lar nos primeiros anos de graduação. Obrigado também à prima Perla, que trilhou essa caminhada comigo, vibrando nas vitórias, sorrindo de nervoso nos momentos difíceis e sempre dizendo: “Você consegue”. Obrigado também aos meus sogros, que como pais, me incentivaram e estiveram do meu lado em todos os momentos.

Aos meus amigos. Primeiramente obrigado ao Pablo por também ter me acolhido nos primeiros anos de faculdade e ter compartilhado vários conhecimentos de elétrica, isso foi muito valioso e serei eternamente grato. Aos amigos que construí durante a graduação e em minha jornada profissional, em especial os amigos do Núcleo de Cidadania Digital (NCD), projeto de extensão da Ufes que me ensinou muito e que me trouxe amigos que levarei para a vida. Aos amigos que fiz durante o estágio da Intel na Alemanha: Júlio, Lucas (Bertão), Alison, Nicholas, Renato (Roy), Selina, Jéssica, Lea e Gabriel (Uri), por toda experiência compartilhada e todos os momentos bons durante minha estadia nesse país que está em meu coração. Por fim, ao amigo Willian Abreu (Will), por ter acreditado em mim e ter me ajudado a chegar até a Alemanha.

Ao professores do Departamento de Informática e Elétrica por terem contribuído para a minha formação. Vivemos em um momento que a educação parece não ser a prioridade, mas vocês se reinventam a cada dia, tornando a sala de aula em ambiente de esperança para o futuro. Muito obrigado e sinto orgulho da profissão de vocês.

Ao meu orientador Vitor Souza, por ter se prontificado a me orientar, mesmo estando com muito trabalho. Muito obrigado pelo conhecimento compartilhado, disponibilidade e atenção durante meu tempo de Iniciação Científica e Projeto de Graduação. Sua organização, paciência e conhecimentos são inspiradores. Mais uma vez, obrigado.

Ao co-orientador César Henrique Bernabé, pelas incansáveis reuniões de desabafo e orientação. Obrigado por topar essa aventura desde a Iniciação Cientifica, sempre com leveza e muita dedicação. Seu amor pelo o que faz é motivo de inspiração. Levarei sua amizade para minha vida. Obrigado, César.

Por fim, obrigado Deus. Quando me restaram forças, Ele ouviu minhas preces me confortando. Obrigado por me iluminar nos momentos difíceis e ter me guiado até onde estou hoje.

"You can't connect the dots looking forward, you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future."

(Steve Jobs)

Resumo

Sistemas adaptativos possuem a capacidade de tomar decisões para se ajustar e se reconfigurar quando necessário. Neste contexto, o *Zanshin* (SOUZA, 2012) é uma abordagem que apoia o desenvolvimento e simulação de sistemas adaptativos utilizando um controlador. O *Unagi* (BERNABÉ, 2017) é uma ferramenta CASE que implementa um editor gráfico para modelos de requisitos para sistemas adaptativos usando o formato esperado pelo controlador *Zanshin*. No entanto, estas ferramentas encontram-se separadas e exigem conhecimento de sua estrutura interna para serem combinadas. Este trabalho tem como proposta a junção de ambas ferramentas para facilitar a criação de modelos no *Unagi* e, posteriormente, simulação no *Zanshin*. Além disso, tem como proposta evoluir o *Unagi* de modo que o mesmo construa e execute simulações de modelos a partir de uma linguagem específica de domínio (DSL ou *Domain-Specific Language*). Para isso, na etapa de junção das ferramentas, utilizaram-se tecnologias como Java™ e XML, para desenvolver botões e seus respectivos comandos de controle do *Zanshin* na barra de ferramentas do Sirius, que compõe o *Unagi*, dentro do Eclipse™. Já na etapa de desenvolvimento da DSL, foram utilizadas as tecnologias Xtext e Xtend para o completo desenvolvimento da linguagem, indo desde a definição da sintaxe até a geração de códigos realizada por ela. Assim, com esse trabalho, houve a integração dos sistemas *Unagi* e *Zanshin*, além da criação da DSL, facilitando a utilização do *Zanshin* para os usuários interessados em Engenharia de Requisitos com foco em Sistemas Adaptativos.

Palavras-chaves: Sistemas Adaptativos, Engenharia de Requisitos, Unagi, Zanshin, Desenvolvimento Orientado a Modelos, DSL.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UML	Linguagem de Modelagem Unificada, do inglês *Unified Modeling Language*
MDD	Desenvolvimento orientado a modelos, do inglês *Model-Driven Development*
GMF	Framework de Modelagem Gráfica, do inglês *Graphical Modeling Framework*
DSL	Linguagem Específica de Domínio, do inglês *Domain-Specific Languages*
IBM	*International Business Machines Corporation*
EMF	Framework de Modelagem do Eclipse, do inglês *Eclipse Modeling Framework*
JDT	Ferramenta de desenvolvimento Java, do inglês *Java development tools*
RMI	*Remote Method Invocation*
OSGi	*Open Services Gateway Initiative*
XML	*Extensible Markup Language*
NEMO	*Núcleo de Estudos em Modelagem Conceitual e Ontologias*
RCP	*Rich Client Applications*
RAP	*Remote Application Platform*

Sumário

1 Introdução

A sociedade hoje, em tempo integral, depende de sistemas complexos com o mínimo de supervisão e manutenção realizada por humanos durante sua fase operacional. O crescimento da complexidade e o tamanho desses sistemas fazem com que seja essencial projetá-los e implementá-los de maneira versátil, flexível e robusta, já que a robustez e confiabilidade tornam-se centrais nesse caso (SABATUCCI; SEIDITA; COSSENTINO, 2018).

O manifesto de computação autônoma da IBM (KEPHART; CHESS, 2003) sugere um direcionamento para lidar com a complexidade do software através da auto-adaptação. Sistemas auto-adaptativos ou, simplesmente, sistemas adaptativos (neste trabalho utilizaremos esse termo), modificam seu comportamento em tempo de execução devido a alterações no sistema, em seus requisitos ou no ambiente em que é implantado (ANDERSSON et al., 2009).

Neste contexto, *Zanshin* (SOUZA, 2012) é uma abordagem para desenvolvimento de sistemas adaptativos fundamentada na premissa que para projetar características adaptativas em sistemas é necessário modelar os requisitos do "ciclo de retroalimentação" (*feedback loop*) que operacionaliza tal adaptação. O *Zanshin* conta com um controlador que implementa o *feedback loop*, elemento que retro-alimenta o sistema enviando ações de adaptação a partir de *logs* que indiquem falhas em requisitos monitorados (SOUZA; MYLOPOULOS, 2013).

A abordagem conta também com uma ferramenta que dá suporte à modelagem de requisitos de sistemas adaptativos chamada *Unagi* (BERNABÉ, 2017). Seguindo o paradigma de Desenvolvimento Orientado a Modelos (*Model-Driven Development* ou MDD), o qual visa elevar o nível de abstração concentrando-se na criação e exploração de modelos de domínio, tal ferramenta oferece produção automática de código por meio da interpretação de modelos visuais (SELIC, 2003; VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Assim, o *Unagi* tem como propósito gerar arquivos de definição de requisitos adaptativos a partir de diagramas criados pelo usuário. Os arquivos gerados pelo *Unagi* podem ser importados no *Zanshin*. No entanto, o *Unagi* não integra-se com o *Zanshin*, sendo necessário inserir manualmente tais arquivos na base de código do controlador, o que requer suporte de desenvolvedores capacitados.

Além disso, o *Unagi* por si só não gera todos os arquivos necessários para a execução de uma simulação no *Zanshin*, fazendo com que o usuário necessite desenvolver esses arquivos manualmente, exigindo um conhecimento da linguagem Java™. Diante disso, a DSL textual vem para gerar esses arquivos de códigos automaticamente, tornando a

utilização do *Zanshin* uma experiência completa a partir do uso dessas ferramentas.

Este trabalho concentra-se em torno da integração das abordagens *Unagi* e *Zanshin*, explorando a arquitetura do *Unagi* para isso, além da criação de uma linguagem específica de domínio textual (DSL ou *Domain-Specific Language*) (BETTINI, 2016), de modo que auxilie pesquisadores e interessados no uso dessas ferramentas no campo da Engenharia de Requisitos para Sistemas Adaptativos.

1.1 Motivação e Justificativa

Tendo em vista o crescimento da complexidade dos sistemas atuais, a comunidade de Engenharia de Software passou a buscar novas formas de projetar e gerenciar sistemas e serviços. Um caminho de estudo que se mostrou promissor, no qual o sistema tem a capacidade de se adaptar a falhas sem qualquer intervenção humana, é o de Sistemas Adaptativos (BRUN et al., 2009).

Dentro deste contexto, foi proposta a abordagem *Zanshin* (SOUZA, 2012) para o desenvolvimento de sistemas adaptativos baseados no uso de modelos de requisitos adaptáveis em tempo de execução, incluindo a implementação de um controlador que permite a simulação de um sistema adaptativo baseado nesses modelos. Como ferramenta para apoiar esta abordagem, foi desenvolvido o *Unagi* (BERNABÉ, 2017), que permite a modelagem gráfica/diagramática dos requisitos do sistema, convertendo e exportando o resultado em um formato capaz de ser lido pelo controlador *Zanshin*.

Essas duas ferramentas são baseadas na mesma plataforma (Eclipse™)[1], porém encontram-se separadas, requerendo um esforço manual e conhecimento da estrutura interna das ferramentas, além de entendimento da linguagem de programação Java™, por parte de quem desejar realizar experimentos com *Zanshin*.

1.2 Objetivos

O objetivo geral desse trabalho é integrar as ferramentas *Zanshin* e *Unagi*, além da construção de uma linguagem específica de domínio textual (DSL ou *Domain-Specific Language*), que facilite a geração dos arquivos necessários para uma simulação *Zanshin*, que atualmente o *Unagi* não gera. Desta maneira, possibilitar a construção de uma simulação de sistema e sua execução através do *Zanshin* em um mesmo ambiente.

Dessa forma, pretende-se tornar fácil a tarefa de criação de modelos e simulação com o uso dessas ferramentas integradas por pessoas interessadas no campo da Engenharia de Requisitos Orientada a Objetivos com foco em desenvolvimento de sistemas adaptativos.

[1] <https://www.eclipse.org/>

A partir do objetivo geral citado anteriormente, tem-se os seguintes objetivos específicos:

- Estudo dos conceitos que cercam o *framework Zanshin* e a ferramenta CASE *Unagi*, como Engenharia de Requisitos, Sistemas Adaptativos, Desenvolvimento Orientado a Modelos, tecnologias utilizadas (em especial a Sirius™), etc.;
- Estudo de trabalhos relacionados ao tema, como as ferramentas GATO (PIMENTEL, 2015), *TAOM4e* (MORANDINI et al., 2007), *OpenOME* (HORKOFF; YU; ERIC, 2011), *piStar* (PIMENTEL; CASTRO, 2018), *Objectiver*[2] e *RE Tools* (SUPAKKUL; CHUNG, 2012);
- Estudo do uso das tecnologias Xtext e Xtend para o desenvolvimento de uma DSL;
- Desenvolvimento de novas interfaces gráficas e funcionalidades que permitam o acionamento e comunicação com o controlador *Zanshin* a partir do editor *Unagi*;
- Proposta de uma DSL para que seja possível o desenvolvimento e execução de simulações de modelos no *Unagi* sem a necessidade de escrever código em linguagem de programação;
- Realização de testes dos artefatos desenvolvidos utilizando como exemplos os sistemas já modelados anteriormente para o *Zanshin*.

1.3 Metodologia

Inicialmente, como forma de embasamento acerca da abordagem *Zanshin* (SOUZA, 2012), do desenvolvimento orientado a modelos (MDD) (PASTOR et al., 2008; ALMEIDA, 2006), dos softwares para a criação de ferramentas seguindo a abordagem MDD e da ferramenta CASE *Unagi* (BERNABÉ et al., 2017), consultaram-se artigos previamente fornecidos pelo orientador. Além disso, foi utilizado o livro *Implementing Domain-Specific Languages with Xtext and Xtend: Second Edition* (BETTINI, 2016) para se aprofundar no desenvolvimento de uma DSL.

Posteriormente, com o intuito de aproximar-se da plataforma Eclipse™, usaram-se tutoriais (VOGEL, 2018a; VOGEL, 2018b; VOGEL, 2016). Na fase de integração do *framework Zanshin* e da ferramenta CASE *Unagi*, utilizaram-se arquivos disponibilizados nos repositórios do GitHub dos autores Vítor Estêvão Silva Souza[3] e César Henrique Bernabé,[4] respectivamente.

[2] <http://www.objectiver.com/index.php?id=25>
[3] <https://github.com/sefms-disi-unitn/Zanshin/wiki>
[4] <https://github.com/hbcesar/unagi-tool>

Já na etapa de implementação do trabalho, utilizou-se a plataforma Eclipse™ e o módulo do editor gráfico da ferramenta CASE *Unagi* que emprega o Sirius™[5], um *plugin* que simplifica o *Framework* de Modelagem Gráfica (GMF) do Eclipse™.

Ainda na fase de desenvolvimento, com o objetivo de adicionar botões e seus respectivos comandos na barra de ferramentas *tab-bar* do Sirius™ (SIRIUS, 2017), utilizaram-se tutoriais[6,7] disponíveis no sítio eletrônico do Eclipse™, juntamente com as tecnologias XML e Java™.

Para a evolução do *Unagi* por meio do desenvolvimento de uma DSL, utilizou-se o framework Xtext[8] e a linguagem de programação Xtend,[9] responsáveis pela implementação de linguagem específica de domínio (DSL) no Eclipse™. Em conjunto à essas tecnologias, também foi usado o ANTLRWorks[10] como ferramenta de *debug* para a gramática da linguagem desenvolvida.

1.4 Organização do Texto

O trabalho está dividido em cinco capítulos principais, além da introdução. Abaixo segue uma breve descrição dos próximos capítulos.

- **Capítulo 2** – Referencial Teórico: discute sobre Desenvolvimento Orientado a Modelos, com foco no *Eclipse Modeling Framework*, que emprega essa abordagem e que é um *framework* utilizado pelos softwares *Unagi* e *Zanshin* em seu desenvolvimento. Também, nesse capítulo é abordado o Sirius™, ferramenta que foi explorada para realizar a integração das ferramentas *Unagi* e *Zanshin*, além de discutir sobre o *Unagi* e *Zanshin* em si. Por fim, discute sobre o conceito de *Domain Specific Language* (DSL) e ferramentas que são utilizadas para desenvolver uma;

- **Capítulo 3** – Integração das ferramentas *Unagi* e *Zanshin*: nesse capítulo é detalhado como foi desenvolvida a integração das ferramentas *Unagi* e *Zanshin*, além de apresentar a arquitetura dessas ferramentas após a integração;

- **Capítulo 4** – Zanshin DSL: descreve como foi desenvolvida a DSL para o *Zanshin*, chamada de Zanshin DSL. Descreve também seu funcionamento e como foi desenvolvido o gerador de códigos utilizado por ela;

- **Capítulo 5** – Validação: descreve o processo de validação da primeira e segunda

[5] <https://www.eclipse.org/sirius/>
[6] <https://www.eclipse.org/sirius/doc/developer/extensions-provide_tabbar_extensions.html>
[7] <https://www.eclipse.org/forums/index.php/t/1067173/>
[8] <https://www.eclipse.org/Xtext/>
[9] <https://www.eclipse.org/xtend/>
[10] <https://www.antlr3.org/>

partes do trabalho, que são a integração do *Unagi* e *Zanshin* e o desenvolvimento da DSL, respectivamente;

- **Capítulo 6** – Considerações Finais: apresenta a conclusão do trabalho, desafios encontrados durante sua construção e os pontos de melhorias a serem trabalhados no futuro.

2 Referencial Teórico

Este capítulo apresenta os principais conceitos teóricos acerca do *framework Zanshin* e da ferramenta CASE *Unagi*, além de descrever os conceitos que fundamentam o desenvolvimento da integração dessas duas ferramentas e os artefatos utilizados para a criação de uma DSL que auxilia no desenvolvimento de simulações para o *Zanshin*.

A Seção 2.1 apresenta um breve resumo sobre Desenvolvimento Orientado a Modelos (*Model-Driven Development* ou MDD), assim como as principais ferramentas que foram utilizadas durante o desenvolvimento do *Unagi* e que auxiliaram na integração com o *Zanshin*, como as funcionalidades EMF de modelagem do Eclipse™ e o *plugin* Sirius™. A Seção 2.2 aborda o sistema *Zanshin* e seus detalhes. A Seção 2.3 apresenta a ferramenta CASE *Unagi* e as particularidades do seu Editor Gráfico e Conversor. Por fim, a Seção 2.4 descreve o que é uma DSL e as ferramentas necessárias para sua implementação.

2.1 Desenvolvimento Orientado a Modelos

Pelo contexto histórico de desenvolvimento de software, os níveis de abstração e reusabilidade dos sistemas vêm sendo elevados. Assim, como forma de reduzir a complexidade, desenvolvedores de software estão criando abstrações que os ajudam a focar no conteúdo do desenvolvimento ao invés das especifidades da tecnologia de criação adotada (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014).

O Desenvolvimento Orientado a Modelos (*Model-Driven Development* ou MDD) pode diminuir a lacuna conceitual que existe entre o problema e os domínios de implementação no desenvolvimento de softwares complexos, já que no MDD um sistema é um modelo consistente com seu meta-modelo (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014). Assim, dentro desse contexto, o modelo é a ligação entre o domínio do problema e o domínio da solução. O meta-modelo define conceitos abstratos, que são utilizados para definir o modelo do sistema (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014).

Dessa forma, o MDD tem o objetivo de melhorar a produtividade e a manutenção do software, aumentando o nível de abstração do código-fonte (PERIŠIĆ, 2014). Ele parte do princípio de que o desenvolvimento de software deve se concentrar na produção de modelos e não de códigos. Isso traz a vantagem de tornar possível expressar modelos utilizando conceitos que são muito menos ligados à tecnologia de implementação e que estão muito mais próximos do domínio do problema, fazendo com que haja uma maior facilidade de especificar, entender e manter esses modelos (SELIC, 2003).

2.1.1 Eclipse Modeling Framework

O *Framework* de Modelagem do Eclipse (*Eclipse Modeling Framework* ou EMF) é um recurso de estruturação e geração de código que permite definir um modelo de dados a partir de uma estrutura Java, XML ou UML. Tendo uma dessas três estruturas, utilizando EMF como intermédio, é possível gerar as outras, assim como as classes de implementação correspondentes (STEINBERG et al., 2008). Além disso, a unificação dessas tecnologias torna um modelo de EMF a representação comum de alto nível que engloba todas elas, independentemente de qual tecnologia é utilizada para defini-lo (STEINBERG et al., 2008).

Pode-se utilizar o EMF como introdução à modelagem para programadores Java. Ele consegue promover uma ligação entre os modeladores e os programadores, relacionando conceitos de modelagem com as representações Java (STEINBERG et al., 2008). Assim, é considerado um recurso de criação de aplicativos Java com base em definições de modelagem simples (STEINBERG et al., 2008).

Com todos esses recursos que o EMF proporciona e comparando-o ao Eclipse™, que é uma plataforma para integração a nível de componentes e interface do usuário, o EMF pode ser considerado como uma base para o compartilhamento de dados e para a interoperabilidade entre ferramentas e aplicativos no Eclipse™ (STEINBERG et al., 2008).

2.1.2 Sirius

O *plugin* Sirius™ é construído sobre o *Framework* de Modelagem Gráfica (*Graphical Modeling Framework* ou GMF) do Eclipse™, simplificando o mesmo e permitindo a criação de editores gráficos de modelos (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Sirius™ suporta três distintas representações: diagramas (modeladores gráficos), tabelas e árvores (representações hierárquicas), mas novas representações podem ser adicionadas através de programação (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014).

Utilizando o Eclipse™, os usuários possuem duas maneiras de executar um editor criado: como um *plugin* para o Eclipse™ ou como um editor independente. Em ambos os casos, qualquer editor desenvolvido com o Sirius™ é aberto na perspectiva de modelagem, que fornece todas as visões, assistentes e menus necessários (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). O *Unagi* é uma ferramenta CASE criada com uso do Sirius™.

A Figura 1 apresenta um exemplo da barra de ferramentas (*tab-bar*) do Sirius™ que compõe o *Unagi*. A *tab-bar* permite a adição de novos botões utilizando tecnologia XML e Java™ para auxiliar nas atividades do editor. Isso possibilitou que, nesse trabalho, fosse possível manipular a *tab-bar* do *Unagi* para integração com o *Zanshin*.

Figura 1 – *Tab-bar* do Sirius^{TM} que compõe o *Unagi*.

2.2 Zanshin

O *Zanshin* (SOUZA, 2012) baseia-se em modelos de objetivos para projetar características adaptativas em sistemas por meio de novos tipos de requisitos, que definem o chamado "ciclo de retroalimentação" (*feedback loop*). Para isso, ele provê um controlador para sistemas adaptativos que, a partir dos modelos de objetivos e de informações coletadas em tempo de execução, envia ao sistema indicações de como se adaptar a situações indesejáveis, além de um simulador que faz o papel do sistema adaptativo, enviando ao controlador dados relativos a falhas e verificando as instruções de adaptação recebidas (SOUZA; MYLOPOULOS, 2013; TALLABACI; SOUZA, 2013).

Na especificação de modelos de requisitos no *Zanshin*, é utilizado o *Eclipse Modeling Framework* (EMF). Estes modelos são especificados através de meta-modelos de EMF e, no *bundle* de simulação do *Zanshin*, são estendidos para prover classes que representam os requisitos e, posteriormente, serem lidos pelo *Zanshin* (SOUZA et al., 2013). Por exemplo, para cada requisito do modelo de objetivo da Figura 2, há uma classe EMF.

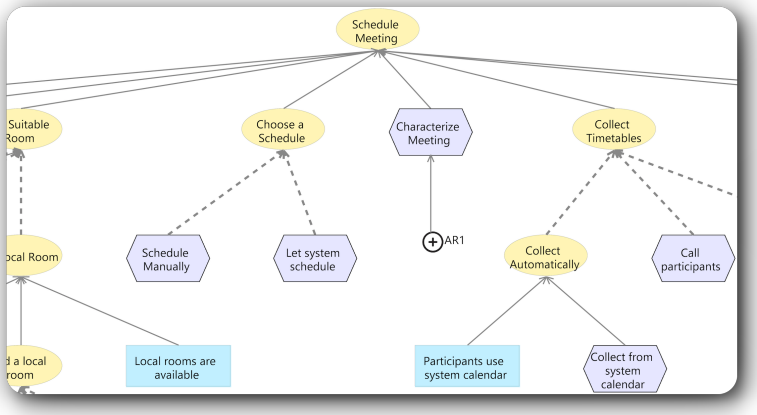


Figura 2 – Fragmento do Modelo Meeting Scheduler criado no Unagi.

A representação dessas classes é feita em tempo de execução. Além disso, são instanciados à medida que o usuário (ou sistema) busca atingir um objetivo e, ao ocorrer mudança de estado, o sistema passa a enviar mensagens para essas instâncias (BERNABÉ, 2017). Isso evidencia a possibilidade do sistema falhar ao tentar atingir seus objetivos iniciais, assim, não tratando objetivos e pressuposições como invariantes que devem ser sempre atingidas (BERNABÉ, 2017). O componente de adaptação terá a função de corrigir essas falhas a partir de tomadas de decisões, fazendo com que os objetivos sejam

atingidos (SOUZA et al., 2013).

Para ocorrer uma simulação no *Zanshin*, ele deve ser executado como um *server* (Zanshin - Server) dentro de um *container* OSGi,[1] que permite aceitar conexões de aplicações externas (chamadas de *target systems*). Além disso, ele possui um módulo de simulação (Zanshin Simulation), que é uma aplicação Java™ executada separadamente e que efetua as simulações conectando-se com o Zanshin - Server. As simulações no *Zanshin* devem ser implementadas em Java no módulo de simulação após a construção do modelo em EMF, encaixando-se no *framework* deste módulo, que espera que sejam providas uma classe que representa o sistema adaptativo (i.e., o *target system*) e uma ou mais classes que implementem a simulação.

Por exemplo, após a construção do modelo da Figura 2, e tendo seus arquivos EMF também incluídos no módulo de simulação, uma classe representando o sistema *Meeting Scheduler* adaptativo seria definida como um SimulatedTargetSystem, conforme a Listagem 2.1. Esta classe recebe um arquivo de *lock* que é usado pelo *framework* para sincronizar as *threads* de simulação e deve implementar os métodos que espera-se que sejam chamados durante as simulações. Neste caso, espera-se que nas simulações que serão implementadas o *Zanshin* envie as mensagens initiate, abort e applyConfig para o sistema adaptativo como resposta do *feedback loop*.

Em seguida, para facilitar a implementação das diferentes simulações, é implementada uma classe abstrata para simulações do *Meeting Scheduler*, derivada da classe AbstractSimulation, exibida na Listagem 2.2. Para que cada simulação já herde questões comuns a todas elas, esta classe define o caminho para os arquivos de modelo, constantes para os elementos de modelo que serão referenciados e o objeto *lock* que será usado, além de implementar o método que registra um sistema adaptativo junto ao *Zanshin*, mantendo a identidade do sistema e de cada sessão de uso em atributos também usados pelas simulações específicas.

Por fim, uma simulação específica é exemplificada na Listagem 2.3. Toda a simulação é construída no método doInit(), que adiciona partes de simulação (instâncias de SimulationPart) à coleção parts, herdada de uma classe ancestral. Esta simulação: (1) cria uma sessão de uso do *Zanshin*; (2) simula a falha da tarefa *Characterize Meeting* e aguarda a resposta do *Zanshin*; (3) novamente simula a mesma falha e aguarda resposta; (4) simula uma terceira falha na execução desta tarefa; e (5) finalmente reconhece que o *Zanshin* indicou abortar a adaptação após muitas tentativas. É importante ressaltar que esta sequência de passos simulados está de acordo com o que se espera do modelo construído (i.e., da especificação do *AwReq* AR1), bem como em sincronia com o que a classe SchedulerSimulatedTargetSystem responde a cada instrução recebida pelo *Zanshin*.

[1] *Open Services Gateway Initiative*, <https://www.osgi.org/>

Listagem 2.1 – Definição do *Meeting Scheduler* como um sistema adaptativo a ser simulado no *Zanshin*.

```
1  package it.unitn.disi.zanshin.simulation.cases.scheduler;
2
3  import java.util.Map;
4  import it.unitn.disi.zanshin.simulation.SimulatedTargetSystem;
5
6  public class SchedulerSimulatedTargetSystem extends SimulatedTargetSystem {
7    private Object lock;
8
9    public SchedulerSimulatedTargetSystem(Object lock) {
10     this.lock = lock;
11   }
12
13   @Override
14   public void initiate(Long sessionId, String reqName) {
15     super.initiate(sessionId, reqName);
16
17     // Initiate is the last command of the Retry strategy, thus we should notify
18     // the simulation to continue, as this strategy is selected to deal with the
19     // failures of AR1.
20     synchronized (lock) {
21       lock.notifyAll();
22     }
23   }
24
25   @Override
26   public void abort(Long sessionId, String awreqName) {
27     super.abort(sessionId, awreqName);
28
29     // After two attempts of the Retry strategy, Zanshin replies with an Abort,
30     // also for the AR1 simulation.
31     synchronized (lock) {
32       lock.notifyAll();
33     }
34   }
35
36   @Override
37   public void applyConfig(Map<String, String> newConfig) {
38     super.applyConfig(newConfig);
39
40     // ApplyConfig is the last command of the Reconfiguration strategy, thus we
41     // should notify the simulation to continue, as this strategy is selected to
42     // deal with failures of AR4.
43     synchronized (lock) {
44       lock.notifyAll();
45     }
46   }
47 }
```

Listagem 2.2 – Classe abstrata com os elementos comuns a todas as simulações referentes ao *Meeting Scheduler*.

```
1  package it.unitn.disi.zanshin.simulation.cases.scheduler;
2
3  import it.unitn.disi.zanshin.simulation.Logger;
4  import it.unitn.disi.zanshin.simulation.SimulationUtils;
5  import it.unitn.disi.zanshin.simulation.cases.AbstractSimulation;
6  import java.io.IOException;
7
8  public abstract class AbstractSchedulerSimulation extends AbstractSimulation {
9    private static final Logger log = new Logger(AbstractSchedulerSimulation.class)
         ;
10   protected static final String BASE_PATH = AbstractSchedulerSimulation.class.
         getPackage().getName().replace('.', '/') + '/';
11   protected static final String META_MODEL_FILE_PATH = BASE_PATH + "scheduler.
         ecore";
12   protected static final String MODEL_FILE_PATH = BASE_PATH + "model.scheduler";
13   protected static final String T_CHARACT_MEET = "T_CharactMeet";
14   protected static final String D_LOCAL_AVAIL = "D_LocalAvail";
15   protected static final String T_CALL_PARTNER = "T_CallPartner";
16   protected static final String T_CALL_HOTEL = "T_CallHotel";
17   protected static Object lock = new Object();
18   protected String targetSystemId;
19   protected Long sessionId;
20
21   protected void registerTargetSystem() throws IOException {
22     // Registers the A-CAD as target system in Zanshin, if not already registered
         .
23     log.info("Registering the Meeting Scheduler as a target system in Zanshin..."
         );
24     targetSystemId = SimulationUtils.registerTargetSystem(zanshin, new
         SchedulerSimulatedTargetSystem(lock), META_MODEL_FILE_PATH,
         MODEL_FILE_PATH);
25     log.info("Target system registered as: {0}", targetSystemId);
26   }
27
28   @Override
29   public Object getLock() {
30     return lock;
31   }
32 }
```

2.3 Unagi

Para a realização de uma simulação no *Zanshin* é necessária a criação de arquivos de código manualmente. A ferramenta CASE *Unagi*, por sua vez, veio com o intuito de facilitar a simulação de projetos por parte dos usuários do *Zanshin*. Dessa forma, auxilia na elaboração de arquivos de especificação do modelo de domínio. Essa automatização feita pelo *Unagi* não havia sido realizada antes por nenhuma outra solução. Atualmente, arquivos XML e *Ecore*, que são modelos de objetivos e que antes eram necessários serem escritos pelos utilizadores do *Zanshin*, são gerados automaticamente pelo *Unagi*.

Os modelos de objetivos utilizados pelo *Zanshin* são gerados por meio do editor gráfico do *Unagi*, usando os mesmos componentes da plataforma Eclipse™ na qual o *Zanshin* é baseado, facilitando a compatibilidade dessas ferramentas. O Sirius™ (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014), *plug-in* utilizado para o desenvolvimento desse editor e

Listagem 2.3 – Simulação de falha do *AwReq* AR1 que monitora *Characterize Meeting*.

```
1  package it.unitn.disi.zanshin.simulation.cases.scheduler;
2
3  import it.unitn.disi.zanshin.simulation.Logger;
4  import it.unitn.disi.zanshin.simulation.cases.SimulationPart;
5
6  public class SchedulerAR1FailureSimulation extends AbstractSchedulerSimulation {
7    private static final Logger log = new Logger(SchedulerAR1FailureSimulation.
       class);
8
9    @Override
10   protected void doInit() throws Exception {
11     registerTargetSystem();
12
13     parts.add(new SimulationPart() { // Adds the first part of the simulation.
14       @Override
15       public boolean shouldWait() { return true; }
16
17       @Override
18       public void run() throws Exception {
19         // Creates a user session, as if someone were using the Meeting Scheduler
20         sessionId = zanshin.createUserSession(targetSystemId);
21         log.info("Created a new user session with id: {0}", sessionId);
22
23         // Simulates a failure in task "Characterize meeting".
24         log.info("Meeting organizer tries to characterize meeting but fails!");
25         zanshin.logRequirementStart(targetSystemId, sessionId, T_CHARACT_MEET);
26         zanshin.logRequirementFailure(targetSystemId, sessionId, T_CHARACT_MEET);
27       }
28     });
29
30     parts.add(new SimulationPart() { // Second part.
31       @Override
32       public boolean shouldWait() { return true; }
33
34       @Override
35       public void run() throws Exception {
36         // Simulates another failure in task "Characterize meeting".
37         log.info("Tries *again* to Characterize meeting but it fails!");
38         zanshin.logRequirementStart(targetSystemId, sessionId, T_CHARACT_MEET);
39         zanshin.logRequirementFailure(targetSystemId, sessionId, T_CHARACT_MEET);
40       }
41     });
42
43     parts.add(new SimulationPart() {  // Third part.
44       @Override
45       public boolean shouldWait() { return true; }
46
47       @Override
48       public void run() throws Exception {
49         // Simulates another failure in task "Characterize meeting".
50         log.info("Tries *one more time* to Characterize meeting but it fails!");
51         zanshin.logRequirementStart(targetSystemId, sessionId, T_CHARACT_MEET);
52         zanshin.logRequirementFailure(targetSystemId, sessionId, T_CHARACT_MEET);
53       }
54     });
55
56     // Fourth and last part (being the last one, it should not wait).
57     parts.add(new SimulationPart() {
58       @Override
59       public boolean shouldWait() {
60         return false;
61       }
62
63       @Override
64       public void run() throws Exception {
65         // Simulates another failure in task "Characterize meeting".
66         log.info("Abort! Today is not a good day to schedule meetings...");
67       }
68     });
69   }
70 }
```

que tem como base as ferramentas de modelagem da plataforma Eclipse (em especial o *EMF*), possui uma barra de ferramentas (*tab-bar*) que pode ser explorada para contribuir na evolução de ferramentas, o que foi utilizado na integração entre *Unagi* e *Zanshin*.

Além do editor gráfico que permite a modelagem de objetivos e requisitos adaptativos, o *Unagi* possui um conversor desenvolvido na linguagem *Acceleo*, responsável por transformar o modelo gráfico em arquivos compatíveis com o *Zanshin*.

2.3.1 Editor Gráfico

O editor gráfico do *Unagi* consiste em um módulo responsável pela modelagem de diagramas de objetivos para sistemas adaptativos. O *Unagi*, que é composto pelo módulo de editor gráfico e conversor, foi construído sob a plataforma Eclipse™ e tecnologias de editores de diagramas como Ecore, EMF e o *plugin* Sirius™. Essa escolha se deu partindo do princípio de que o *Zanshin* foi construído utilizando a mesma plataforma, e que esta provê tecnologias diversas de desenvolvimento, o que proporciona uma maior compatibilidade entre *Zanshin* e *Unagi* (BERNABÉ, 2017).

O editor gráfico admite a criação do diagrama utilizando recursos de arrastar e soltar, da mesma forma que é encontrada em outros editores atuais, o que o torna intuitivo (BERNABÉ, 2017). Conforme pode ser visto na Figura 3, esse editor possui uma área de modelagem, além de um espaço de elementos e outro com os tipos de refinamento que podem ser utilizados para criação de um modelo (BERNABÉ, 2017). Os elementos disponíveis na paleta de opção (à direita da tela) são arrastados para a área de modelagem, e os refinamentos são escolhidos de acordo com a modelagem e desenhados clicando no elemento de origem e destino, necessariamente nessa sequência. Além disso, é possível realizar a mudança dos atributos de cada elemento através do espaço de opções disponível na área inferior do editor (Figura 4) (BERNABÉ, 2017).

Dentro da área de modelagem, é possível acessar a visão que favorece a descrição dos *EvoReqs* (*Evolution Requirements* ou Requisitos de Evolução) ligados a cada *AwReq* (*Awareness Requirements* ou Requisitos de Percepção) através de um clique duplo sobre o *AwReq* requerido. Dessa forma, o editor possibilita a opção de gerar um subdiagrama de detalhamento dos elementos, como é visto na Figura 5 (BERNABÉ, 2017). Com isso, é possível que o usuário defina algumas configurações, como as Condições de Resolução, as Condições de Aplicabilidade e as Estratégias de Adaptação empregando a mesma lógica de arrastar e soltar (BERNABÉ, 2017). Vale lembrar que o modelo admite apenas a elaboração de um elemento próprio após a criação de seu elemento "pai" (BERNABÉ, 2017).

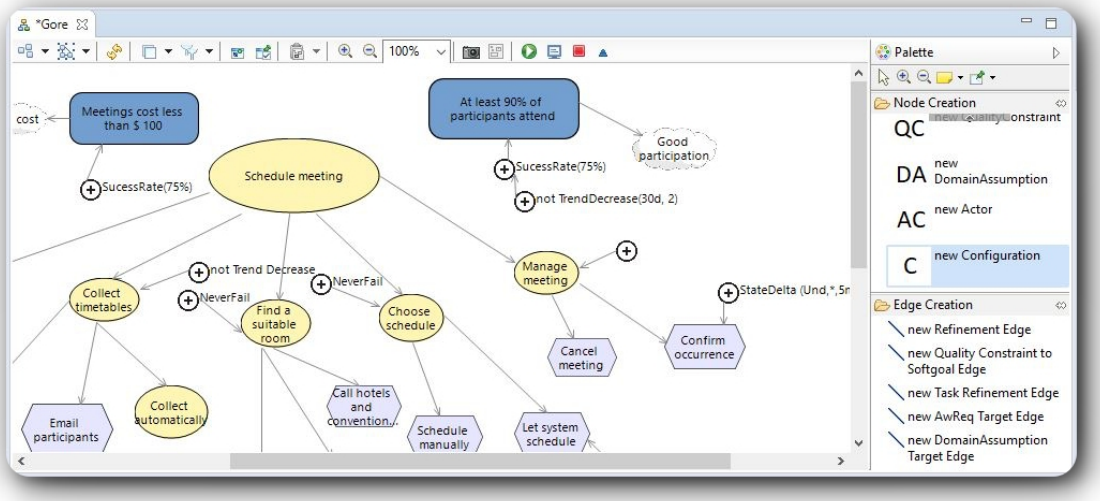


Figura 3 – Editor *Unagi*.

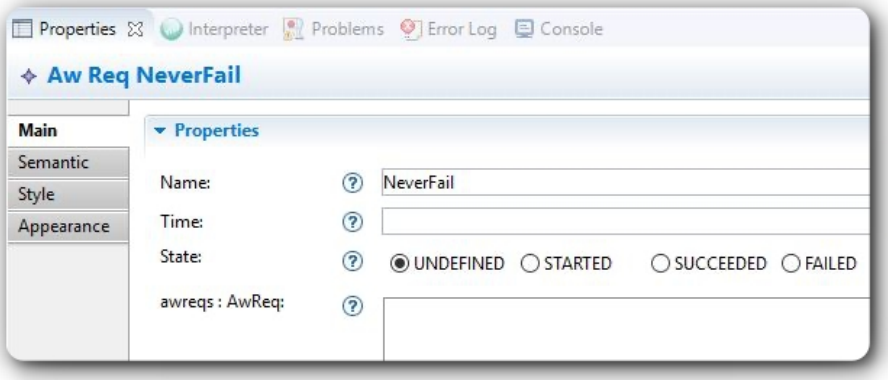


Figura 4 – Editor *Unagi* - Espaço de Opções.

2.3.2 Conversor

O conversor do *Unagi* é um gerador de código que utiliza a linguagem Acceleo para a sua implementação. O Acceleo parte do princípio de utilizar instâncias de modelos baseados em EMF para a conversão de outra linguagem. Sendo assim, o conversor do *Unagi* foi implementado de modo que o processo de conversão se baseia em analisar a árvore de elementos do editor gráfico, convertendo cada um dos objetos em linhas do texto dos arquivos XML formatadas de acordo com os padrões do *Zanshin* (BERNABÉ, 2017).

Para isso, o processo de conversão se inicia na busca do elemento inicial pelo editor, cria sua linha com as características próprias, procura pela existência de refinamentos e, caso haja, verifica a lista de elementos e a lista de refinamentos deles recursivamente, até que as folhas da árvore sejam alcançadas (BERNABÉ, 2017). Por último, os arquivos gerados da conversão e que podem ser importados no *Zanshin* são armazenados no pacote

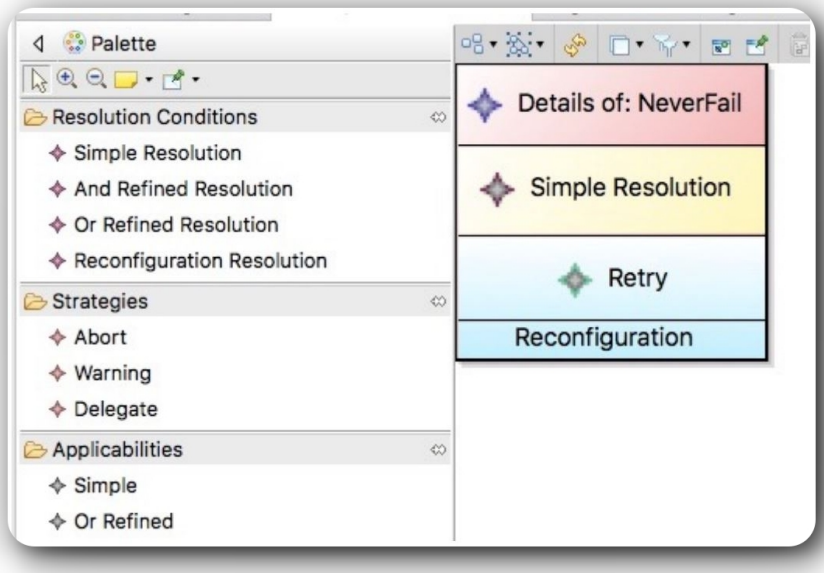


Figura 5 – Editor *Unagi* - Subdiagrama de Especificação de *EvoReqs* (BERNABÉ, 2017).

de conversão escolhido pelo usuário, ou na sub-pasta "zanshin-gen", criada dentro da pasta do projeto de modelagem caso o usuário não tenha especificado um pacote. Para ter conhecimento de falha ou sucesso da conversão, um arquivo de nome "log.txt" é gerado com essas informações.

2.4 Linguagens Específicas de Domínio

Uma linguagem específica de domínio (DSL ou *Domain-Specific Language*), diferente das linguagens de programação de uso geral, como Java e C, é voltada para a solução de um problema próprio, sem fornecer recursos para solução de problemas gerais (BETTINI, 2016). As DSL's vêm com propósito de facilitar a criação de soluções completas com menos código, além de separar os conceitos técnicos dos conceitos de domínio, reduzindo os custos de manutenção (BETTINI, 2016).

Para a criação de uma DSL no Eclipse™, existem ferramentas de apoio, como o framework Xtext e a linguagem de programação Xtend. A primeira é um *framework* completo para implementação rápida de DSL, possuindo infraestrutura de linguagem completa, desde o analisador (*Parser*) até o interpretador (*Interpreter*), inclusive a associação com o Eclipse™ IDE (BETTINI, 2016). Já a segunda é uma linguagem de programação de propósito geral implementada em Xtext, utilizada para o desenvolvimento de aplicações em Java, Web ou Android, e que também pode ser utilizada para criação de geradores de código. Além disso, o Xtend pode ser considerada como uma linguagem alternativa da linguagem Java (BETTINI, 2016).

2.4.1 Xtext

O Xtext[2] é um framework do Eclipse™ que possibilita o desenvolvimento fácil de uma linguagem de programação ou uma DSL (BETTINI, 2016). Para a implementação de uma DSL, o Xtext somente exige a criação de uma especificação de gramática baseada em ANTLR (*ANother Tool for Language Recognition*) (BETTINI, 2016). Em ANTLR, a gramática é definida em apenas um arquivo, não necessitando a separação entre a sintaxe e a especificação léxica. A partir desse arquivo, o *parser* é gerado automaticamente.

No desenvolvimento de uma Linguagem de Programação, tem-se o Lexer, que é um *scanner* responsável pela análise lexical de uma linguagem, o Parser, que é o analisador sintático de uma linguagem, e o modelo AST, que são estruturas de dados de uma linguagem representadas em árvore. O Xtext é capaz de gerar todos esses elementos, além de outros elementos complexos na criação de uma linguagem/compilador (BETTINI, 2016). Assim, para a criação de uma DSL utilizando esse *framework*, o desenvolvedor concentra a maior parte do esforço na especificação da gramática, deixando que os outros elementos da linguagem sejam automaticamente resolvidos pelo Xtext.

2.4.2 ANTLRWorks

Como ferramenta para *debug* de gramática, o ANTLRWorks[3] vem como uma ótima solução para os desenvolvedores encontrarem inconsistências em suas gramáticas desenvolvidas utilizando o Xtext. O Xtext por si só não possibilita a realização de *debug* amigável para o usuário da ferramenta, o que dificulta a busca de solução de erros no desenvolvimento por parte do desenvolvedor.

O ANTLRWorks é um ambiente de desenvolvimento completo para gramáticas ANTLR (BOVET, 2021). Auxiliando iniciantes na criação de linguagens de programação, essa ferramenta ajuda na eliminação de inconsistências gramaticais (BOVET, 2021). Além disso, essa ferramenta facilita a manutenção e legibilidade da gramática, já que a torna mais palpável ao desenvolvedor (BOVET, 2021).

2.4.3 Xtend

O Xtend é um flexível e expressivo dialeto de Java™ (XTEND, 2021), possibilitando assim a utilização de qualquer biblioteca Java existente. Alternativamente à linguagem Java, o Xtend, que é considerado uma linguagem de propósito geral, pode ser utilizado para o desenvolvimento de aplicações Java, Web ou Android (BETTINI, 2016).

Uma das possibilidades desse dialeto é a escrita de geradores de código, assim ele pode ser utilizado para conversão do modelo EMF gerado de uma DSL desenvolvida

[2] <https://www.eclipse.org/Xtext/>
[3] <https://www.antlr3.org/works/>

com o Xtext em outra linguagem de programação, como Java, por exemplo (BETTINI, 2016). No Eclipse™, um projeto Xtext já é configurado para utilizar o Xtend, facilitando a utilização dessas duas ferramentas para a criação de uma DSL completa.

2.5 Trabalhos Relacionados

No contexto específico em que o *Unagi* foi desenvolvido (abordagens baseadas em requisitos para sistemas adaptativos), existem ferramentas de apoio similares propostas na literatura. A ferramenta GATO (PIMENTEL, 2015) também permite gerar arquivos para o controlador *Zanshin* a partir de um modelo desenhado em uma ferramenta Web, no entanto GATO não integra o controlador e o simulador *Zanshin*, que devem ser executados manualmente.

O *TAOM4e* (MORANDINI et al., 2007) apoia a elicitação e modelagem de requisitos na linguagem Tropos e foi estendido para incluir suporte aos elementos de modelagem de requisitos para sistemas adaptativos da abordagem Tropos4AS (MORANDINI; PERINI; MARCHETTO, 2011). O editor inclui a ferramenta *t2x* que gera código a partir do modelo desenhado para a plataforma de implementação de agentes *Jadex*. No entanto, o *TAOM4e* não integra a execução e simulação do sistema em uma única ferramenta.

Algumas ferramentas apoiam a modelagem GORE de forma geral, ou seja, não são direcionadas a modelagem de sistemas adaptativos mas focam em modelagem de objetivos. *OpenOME* (HORKOFF; YU; ERIC, 2011) e *piStar* (PIMENTEL; CASTRO, 2018) são voltados ao desenvolvimento de modelos na linguagem $i^\star$, enquanto *Objectiver*[4] foca na linguagem *KAOS* (DARDENNE; LAMSWEERDE; FICKAS, 1993). *RE Tools* (SUPAKKUL; CHUNG, 2012) possui funcionalidades que permitem modelagem GORE nas linguagens $i^\star$, *KAOS* e *NFR* (MYLOPOULOS; CHUNG; NIXON, 1992). O *Unagi* não é baseado em nenhuma linguagem específica (apesar de usar uma sintaxe concreta similar à $i^\star$), mas nos conceitos comuns às várias abordagens GORE, e integra-se com o *Zanshin* com foco no desenvolvimento de sistemas adaptativos.

[4] <http://www.objectiver.com/index.php?id=25>

3 Integração das ferramentas *Unagi* e *Zanshin*

A primeira contribuição deste trabalho foi a integração das ferramentas *Unagi* e *Zanshin*, explorando a *tab-bar* do Sirius™, *framework* utilizado para construção do *Unagi*. Com isso, o processo de modelagem de um sistema e a utilização do mesmo para a realização de uma simulação no *Zanshin* tornou-se mais intuitiva e prática, já que agora essas duas ferramentas fazem parte de um mesmo ambiente.

A integração desses sistemas é apresentada nas seções a seguir. A Seção 3.1 explica a atual arquitetura do sistema após a integração, e então a Seção 3.2 explica como foi desenvolvida tal integração.

3.1 Arquitetura Geral do Sistema

A integração ocorre por meio do módulo do editor gráfico que o *Unagi* possui, conforme descrito na Figura 6. Esse módulo é construído em cima do Sirius e definido em dois modelos (BERNABÉ et al., 2017). O primeiro modelo é o da sintaxe abstrata, que estabelece os elementos que poderão ser criados no modelador gráfico, juntamente com suas propriedades e relações. O segundo modelo, por sua vez, define as características gráficas dos elementos determinados na sintaxe abstrata e é chamado de sintaxe concreta. Ambos modelos utilizam a sintaxe do EMF.

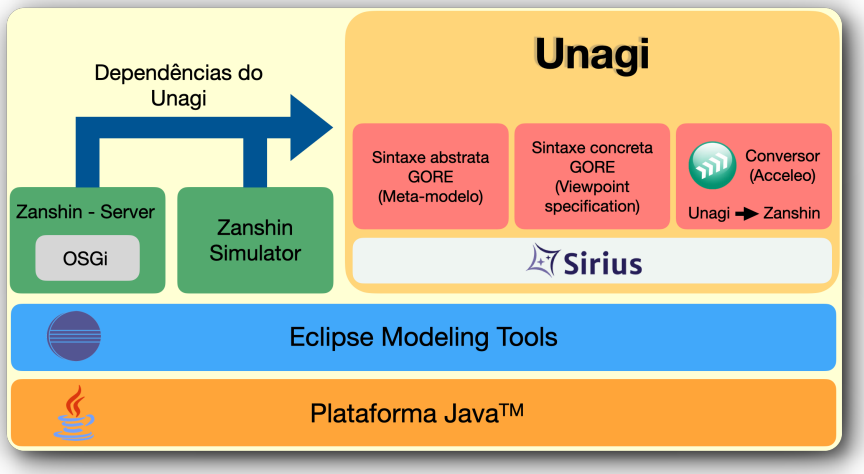


Figura 6 – Descrição da arquitetura da integração das ferramentas

Figura 7 – *Tab-bar* do *Sirius* no *Unagi* com os botões de controle do *Zanshin* (destacados em vermelho).

Tabela 1 – Significado dos botões adicionados na *tab-bar* do Sirius no *Unagi*.

Ícones	Significado
	Executar o servidor RMI do *Zanshin*
	Inicializar o Console de Simulação do *Zanshin*
	Parar o servidor RMI do *Zanshin*
	Converter e Carregar arquivos convertidos pelo *Unagi* para o *Zanshin*

O edito gráfico do *Unagi* possui uma barra de ferramentas (*tab-bar*) do Sirius (SIRIUS, 2017), na qual foram adicionados os botões de controle do *Zanshin*, tendo como resultado a Figura 7. Os comandos associados a cada novo botão são programados na ferramenta utilizando a linguagem Java™ e o componente JDT Debug.[1]

O comportamento de cada botão é resumido na Tabela 1. A partir do *Unagi*, portanto, é possível executar e interromper o controlador *Zanshin* (que executa dentro de um *container* OSGi), iniciar o simulador *Zanshin* (uma aplicação Java simples), além de converter e carregar no controlador *Zanshin* automaticamente o modelo construído no editor *Unagi*. A comunicação com o controlador *Zanshin* por parte do *Unagi* e do simulador é feita por meio da tecnologia RMI.[2]

3.2 Integração das Ferramentas

A integração das ferramentas ocorre por meio da modificação do módulo *gore.design* do *Unagi*. No arquivo de configuração desse módulo, é adicionada uma extensão de modificação de menu (do tipo menus) que aponta para a *tab-bar* do *Sirius* (Figura 8). Para cada *command* adicionado nessa extensão, há uma outra extensão do tipo handlers (Figura 9) que, por sua vez, aponta para outra extensão do tipo command (Figura 10), além de apontar para a classe na linguagem Java que define a ação do botão, conforme mostrado na Figura 9.

Assim, para cada botão adicionado nessa *tab-bar*, foi desenvolvido um arquivo Java™ aplicando configurações referentes à tecnologia utilizada para o sistema ou ação que o botão representava. Na Listagem 3.1 é mostrado, como exemplo, um fragmento do

[1] <https://goo.gl/jprtyY>, <http://bit.do/eJmGL>
[2] *Java Remote Method Invocation*, <https://docs.oracle.com/javase/tutorial/rmi/>

código da ação do botão que executa o servidor RMI do *Zanshin*.

Listagem 3.1 – Exemplo de código, em Java™, da ação do botão que executa o servidor RMI do *Zanshin*.

```
1  package unagi;
2  //...
3
4  public class Server extends AbstractHandler {
5    /**
6     * Create a OSGi Framework launch configuration, save and run
7     */
8    private void launch() {
9      try {
10       ILaunchManager manager = DebugPlugin.getDefault().getLaunchManager();
11       ILaunchConfigurationType type = manager.getLaunchConfigurationType("org.
             eclipse.pde.ui.EquinoxLauncher");
12       ILaunchConfigurationWorkingCopy wc = type.newInstance(null, "Zanshin");
13       wc.setAttribute("append.args", true);
14       wc.setAttribute("automaticAdd", true);
15       //...
16       wc.setAttribute("clearConfig", false);
17       wc.setAttribute("configLocation", "${workspace_loc}/.metadata/.plugins/org.
             eclipse.pde.core/Zanshin");
18       //...
19       wc.setAttribute("useDefaultConfigArea", false);
20       wc.setAttribute("workspace_bundles", "it.unitn.disi.zanshin.adaptation.
             qualia@default:default,"
21           + "it.unitn.disi.zanshin.adaptation@default:default,"
22           + "it.unitn.disi.zanshin.controller@default:default," + "it.unitn.disi.
                 zanshin.core@3:default,"
23           + "it.unitn.disi.zanshin.logging@2:default," + "it.unitn.disi.zanshin.
                 monitoring@default:default");
24       wc.setAttribute("deselected_workspace_plugins", "gore.unagi.converter");
25       ILaunchConfiguration config = wc.doSave();
26       config.launch(ILaunchManager.RUN_MODE, null);
27
28     } catch (Exception e) {
29
30       e.printStackTrace();
31     }
32
33   }
34
35   @Override
36   public Object execute(ExecutionEvent event) throws ExecutionException {
37     launch();
38     return null;
39   }
40 }
```

A execução do *Zanshin* ocorre por meio da criação de uma configuração de execução OSGi no Eclipse de forma automática, procedimento que anteriormente deveria ser feito manualmente.[3] Já o simulador *Zanshin*, que antes precisava ser executado separadamente, agora foi incorporado ao *Unagi* e abre numa visão *Console* do Eclipse para interação com o usuário.

[3] <https://github.com/sefms-disi-unitn/Zanshin/wiki/How-to-run-Zanshin-simulations>

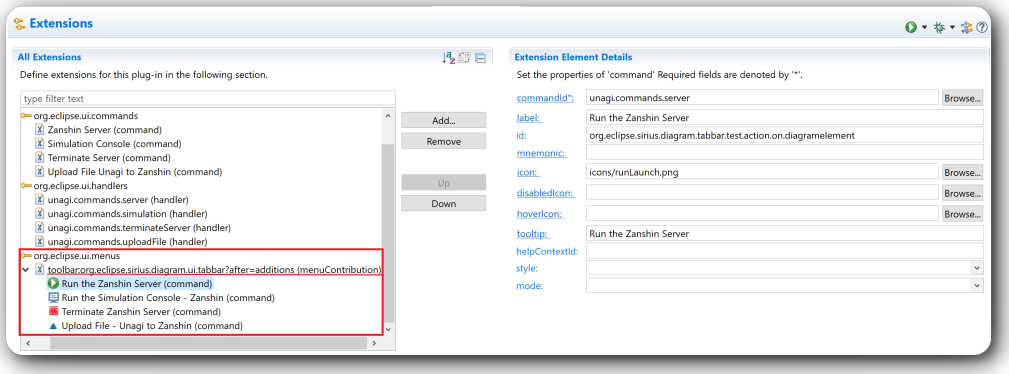


Figura 8 – Extensão de botões adicionados ao pacote *gore.design* do *Unagi*.

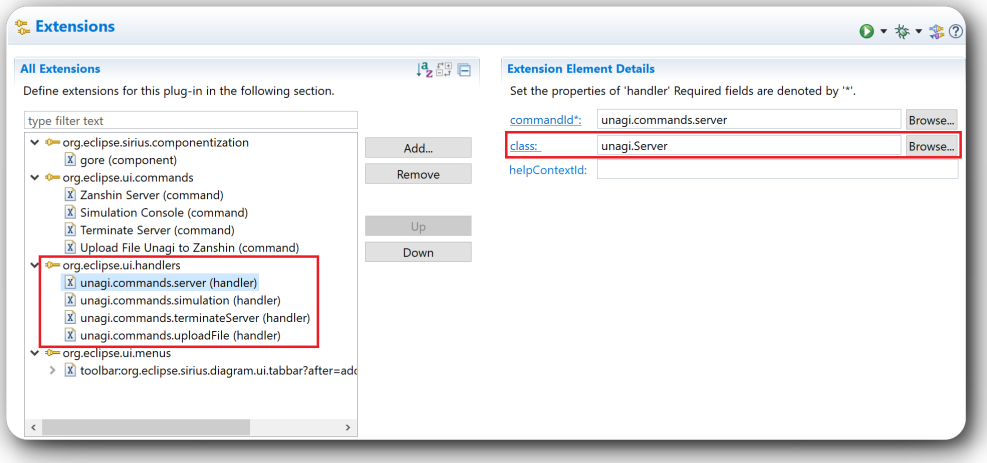


Figura 9 – Extensões do tipo **handlers** com os detalhes dos seus elementos.

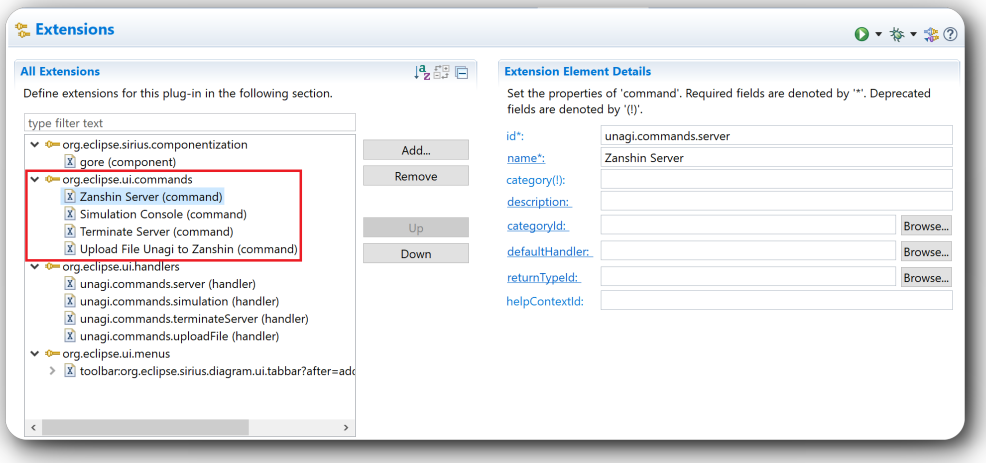


Figura 10 – Extensões do tipo **commands**.

4 Zanshin DSL

O desenvolvimento da DSL foi a segunda contribuição do trabalho, com o propósito de apoiar os usuários do *Zanshin* a realizar simulações nessa ferramenta. Até então, dentre os arquivos necessários para a simulação de um sistema no *Zanshin*, dois desses que apresentam o modelo de requisitos do sistema a ser simulado eram gerados automaticamente pelo *Unagi*, um em formato XML e outro em *Ecore*.

Em uma simulação *Zanshin*, que utiliza modelos de sistemas para testar situações de adaptações, ainda era necessário um processo manual por parte do usuário para o desenvolvimento de código Java implementando os passos necessários para o processo de simulação, como ilustrado na Seção 2.2. Assim, a DSL vem com o propósito de tornar fácil a geração dos arquivos Java necessários para uma simulação no *Zanshin*, já que esses arquivos, para serem desenvolvidos, necessitariam de um conhecimento prévio da linguagem Java e da ferramenta *Zanshin*, em especial de seu *framework* de simulação.

Além disso, sem a DSL o usuário do *Zanshin* deveria desenvolver pelo menos três arquivos em Java para uma única simulação. Com a DSL, o usuário precisa escrever apenas um arquivo de especificação e a mesma se encarrega de gerar todos os outros arquivos necessários automaticamente. A Figura 11 mostra os arquivos que compõem um pacote de simulação no *Zanshin* e que serão gerados automaticamente pelo *Unagi* e pela DSL. O detalhamento desses arquivos será descrito na Seção 4.1.

Figura 11 – Arquivos utilizados pelo *Zanshin* em uma simulação.

A criação da DSL, utilizando ferramentas do Eclipse™, pode ser dividida em duas etapas: Criação de uma Linguagem Xtext (que por si só já é a DSL) e o Conversor de Códigos usando Xtend. O detalhamento do desenvolvimento dessas ferramentas será descrito nas próxima seções: a Seção 4.1 explica como a DSL foi elaborada, usando as tecnologias já discutidas nesse texto, e a Seção 4.2 explica as funcionalidades da DSL. Por

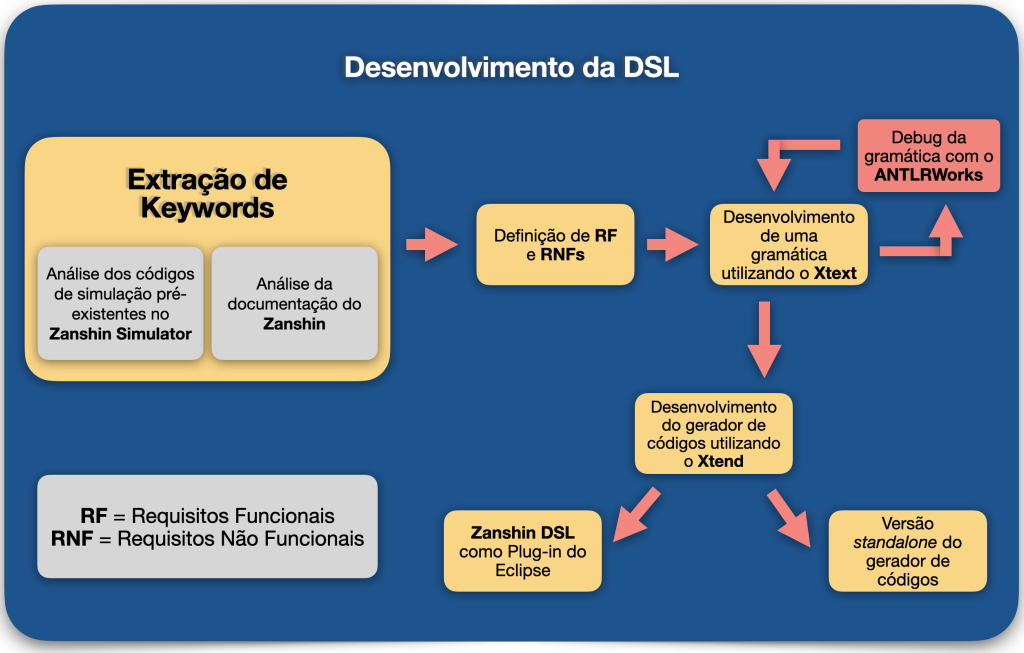


Figura 12 – Fluxograma do processo de criação da Zanshin DSL.

fim, a Seção 4.3 demonstra como o componente responsável pela conversão da gramática da DSL para os arquivos Java nos padrões do *Zanshin* foi construído.

O código fonte do plugin pode ser encontrado em <https://github.com/iagohribeiro/Zanshin-DSL>.

4.1 Criação da DSL

O processo de desenvolvimento da DSL se iniciou pela criação de uma linguagem utilizando o *framework* Xtext. Partindo do principio que *Zanshin* e *Unagi* foram desenvolvidos utilizando ferramentas Eclipse™, decidiu-se que para a construção da DSL iria ser utilizado o Xtext, que também é uma ferramenta do Eclipse™ e que é utilizada para desenvolvimento de DSLs. Além disso, o Xtext utiliza ferramentas como EMF e modelo *Ecore*, assim como o *Zanshin* e *Unagi*, o que viabiliza uma fácil manutenção no futuro pelos interessados nessas ferramentas.

A Figura 12 apresenta um fluxograma com os principais passos da criação da DSL. O passo da criação do gerador de códigos mostrado nessa figura será detalhado no Seção 4.3. Os demais passos são descritos a seguir.

Inicialmente, analisando os códigos já existentes de simulações do *Zanshin*[1] (como o que foi exemplificado na Seção 2.2), foram traçados padrões nesses arquivos para a definição

[1] <https://github.com/nemo-ufes/Zanshin/tree/master/zanshin-simulations/src/it/unitn/disi/zanshin/simulation/cases>

Tabela 2 – Tabela de Keywords.

Keywords	**Definição**
project	Indica o nome do pacote de simulação do *Zanshin*.
Simulation	Define o nome de uma simulação de um pacote de simulação do *Zanshin*.
Part	Indica uma parte de uma simulação.
Failure	Indica a ação de falha para um determinado requisito do sistema.
Success	Indica a ação de sucesso para um determinado requisito do sistema.
Log	Indica a resposta do *Zanshin* para uma parte de simulação.

de *Keywords* a serem utilizadas na gramática da DSL. Além disso, a documentação do *Zanshin*[2] foi usada para o entendimento de como é feita a criação de uma simulação. Da mesma forma, a documentação provê informações extras, que não estavam disponíveis nas simulações de exemplo no passo anterior.

Com essas análises, chegou-se à conclusão que a gramática deveria conter entradas passíveis de serem consumidas para a geração de três principais arquivos de uma simulação *Zanshin*: *Arquivo de Simulação*, *Abstract Simulation* e *Target System*.

Uma simulação (nesse trabalho será considerada como um pacote de simulação) de sistema no *Zanshin* utiliza arquivos de metamodelo, modelo, *Abstract Simulation*, *Target System* e arquivos de simulação (conforme já ilustrado na Figura 11). Os dois primeiros são gerados automaticamente pelo *Unagi* e, portanto, não serão discutidos nesse trabalho.

O arquivo *Abstract Simulation* é responsável por definir qual metamodelo e modelo serão utilizados na simulação, os requisitos do sistema utilizados e, por fim, registra a simulação como um sistema alvo (*target system*). Já o arquivo *Target System* contém os métodos que irão lidar com a resposta de adaptação vindo do *Zanshin*. Para completar o pacote de simulação, tem-se os *Arquivos de Simulação* que, dentro de um mesmo pacote, pode haver mais de um, e consistem em diferentes simulações monitoramento/adaptação (*feedback loop*) que se deseja experimentar para o sistema (modelo) em questão. Um exemplo desse conjunto de arquivos Java construído manualmente foi exemplificado na Seção 2.2.

Após a análise dos arquivos que seriam gerados através da DSL, foi feito o levantamento de todas as entradas que deveriam ser feitas pelo usuário para serem utilizadas nesses arquivos. Assim, com as entradas definidas, seria fácil definir uma gramática para a DSL. A Tabela 2 mostra as *keywords* definidas; na Seção 4.2 será apresentado como a DSL funciona e, consequentemente, como são utilizadas suas *keywords*.

[2] <https://github.com/sefms-disi-unitn/Zanshin/wiki>

Tabela 3 – Tabela de Requisitos Funcionais da DSL.

ID	Descrição
RF-1	• RF-1.1: Um arquivo da DSL pode conter o desenvolvimento de uma ou mais simulações de um pacote de projeto de simulação. • RF-1.2: Um arquivo da DSL pode conter também o desenvolvimento de simulações de um ou mais pacotes de projeto de simulação.
RF-2	Uma simulação pode conter nenhuma ou mais partes de simulação.
RF-3	Uma simulação e uma parte de simulação podem conter uma ou várias entradas de falha e sucesso.
RF-4	Uma simulação e uma parte de simulação podem conter um único log.

Tabela 4 – Tabela de Requisitos Não Funcionais da DSL.

ID	Descrição
RNF-1	A linguagem deve ser de fácil entendimento. Mesmo o usuário não tendo o conhecimento de programação, deve ser capaz de utilizá-la.
RNF-2	A linguagem deve ser concisa (o usuário não precisa escrever muitas linhas de código).
RNF-3	A linguagem deve ser multiplataforma.

Com as *Keywords* definidas, pensou-se em uma estratégia para o desenvolvimento da gramática da linguagem que atendesse todas as necessidades de geração dos arquivos para simulação. Dessa forma, as tabelas 3 e 4 apresentam os Requisitos Funcionais e Não Funcionais, respectivamente, que foram responsáveis pelo processo de desenvolvimento da DSL.

A partir dos requisitos das tabelas 3 e 4, foi possível chegar no resultado de uma gramática e desenvolvê-la utilizando o Xtext, conforme visto na Listagem 4.1. Nessa figura, a primeira linha (Grammar *zanshin.dsl.Dsl*) representa o pacote do projeto da DSL, e o fragmento *zanshin.dsl* o nome declarado para a DSL, que, nesse caso, é chamado de Zanshin DSL.

O primeiro elemento a se atentar em uma definição de gramática é por onde o analisador (*parser*) irá iniciar e o tipo de elemento raiz do modelo da DSL, que é o da AST. Assim, as linhas 5 e 6 da Listagem 4.1 indicam que *Scope* é uma coleção de elementos de escopo e que elas estão armazenadas em um objeto modelo (*Model*) chamado *dsl*.

Ainda na Listagem 4.1, o objeto projeto (*project*), na linha 8, trata-se de um *import* de projetos que a linguagem será capaz de realizar. Isso se dá pelo fato desse objeto utilizar a palavra reservada importedNameSpace pela Xtext que faz com que esse *framework* trate o valor dessa palavra como um *import*. O objeto QualifiedNameWithWildcard, na linha 14, chama o objeto QualifiedName e ambas retornam uma string simples. O símbolo

Listagem 4.1 – Código da gramática da Zanshin DSL.

```
1  grammar zanshin.dsl.Dsl with org.eclipse.xtext.common.Terminals
2
3  generate dsl "http://www.dsl.zanshin/Dsl"
4
5  Model:
6    (dsl+=Scope)*;
7
8  Project:
9    'project' importedNamespace = QualifiedNameWithWildcard;
10
11 QualifiedName:
12     STRING ('.' STRING)*;
13
14 QualifiedNameWithWildcard:
15     QualifiedName '.*'?;
16
17 TestType:
18   Success | Failure;
19
20 Success:
21   simulationType= 'Success' (array ?='[' (length=INT)? ']')? name=STRING;
22
23 Failure:
24   simulationType= 'Failure' (array ?='[' (length=INT)? ']')? name=STRING;
25
26 Log:
27   'Log' message = STRING;
28
29 Scope:
30   project = Project
31   ((simulation += 'Simulation' name += ID (length+=INT)?)?
32     (commands += commandBlock)+)*;
33
34 commandBlock:
35   ('SimulationPart')?
36   (testtype += TestType)*
37   (message += Log)+;
```

"*" na regra do objeto QualifiedName indica que uma string é sucedida por zero ou mais de uma sequência "'.' STRING". Seguindo o mesmo princípio, o símbolo "?" no objeto QualifiedNameWithWildcard indica que a string retornada do objeto QualifiedName é sucedida por zero ou uma sequência de ".*".

Nessa mesma figura, da linha 17 à 24 são tratados os tipos de simulação que o *Zanshin* faz. O objeto TestType indica em sua regra que ele aceita o objeto "Success" ou o objeto "Failure". Esses objetos apresentam uma mesma regra, que determina que um tipo de simulação "Success" ou "Failure" pode ser uma *array* sucedida de uma string (a string é armazenada na palavra reservada pelo Xtext name). A regra também determina que esses objetos podem ser compostos por essas *keywords* sucedidas de uma string.

A linha 26 da Listagem 4.1 apresenta o objeto que é composto pela string Log sucedida de uma string armazenada na variável message. Esse objeto representa as mensagens mostradas no terminal de simulação do *Zanshin* durante uma simulação de sistema.

Por fim, a linha 29 apresenta o objeto Scope que define como será a estrutura da

linguagem. Dentro desse objeto, é chamado o objeto commandBlock definido na linha 34 que apresenta os comandos de ação suportados em um escopo de simulação e que foram explicados anteriormente.

Após a construção da gramática, são gerados os artefatos da linguagem através do MWE2 (*Modeling Workflow Engine 2*), elemento utilizado pelo Xtext para realizar essa geração. Durante a execução do workflow da MWE2, o Xtext gera artefatos relacionados ao Editor UI para a DSL e deriva uma especificação ANTLR, a partir da gramática Xtext definida, com todas as ações para criar uma AST durante a análise (*parsing*). Nesse contexto, o EMF é utilizado para gerar os nós dessa AST.

Assim, para a especificação da gramática desenvolvida, o Xtext automaticamente infere o meta-modelo EMF (Figura 13) para a linguagem (nesse trabalho, é a Zanshin DSL). Para cada regra dessa gramática, uma interface EMF Java e sua respectiva implementação será gerada, conforme mostrado na Figura 14.

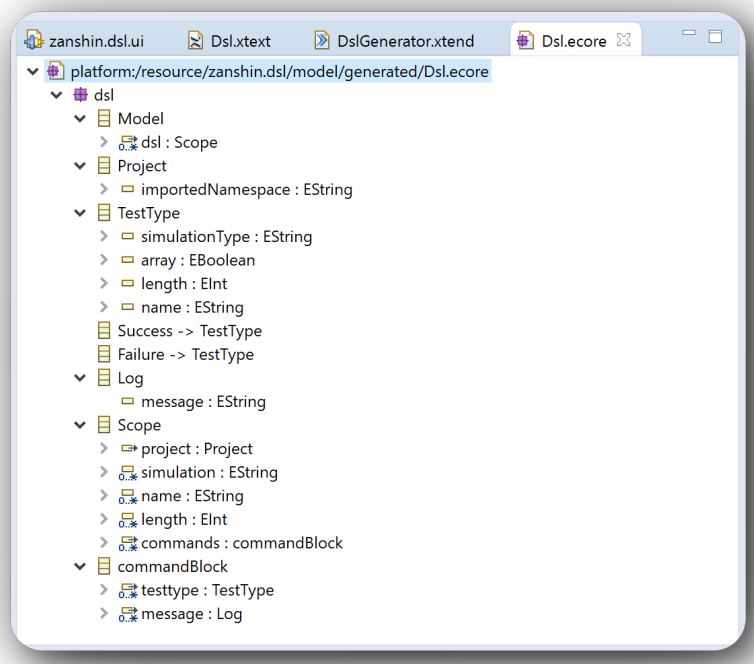


Figura 13 – Meta-modelo da Zanshin DSL.

Apesar da robustez da Xtext, ela não possibilita que o desenvolvedor entenda algumas inconsistências gramaticais de forma fácil. Seus *logs* não apresentam em que parte da gramática ocorreu redundância ou falhas, além de não haver uma ferramenta de simulação da gramática para *debug*.

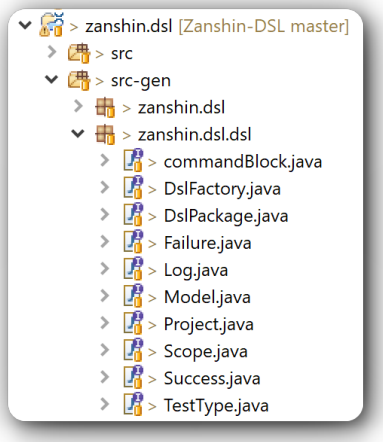


Figura 14 – Interfaces EMF Java criadas pelo Xtext automaticamente a partir da gramática da Zanshin DSL.

Para apoiar o desenvolvedor a encontrar falhas na gramática, o Xtext fornece *logs* no console do Eclipse™, além de diagramas, conforme podem ser vistos nas figuras 15 e 16, respectivamente. Com isso, foi utilizado a ferramenta ANTLRWorks com o intuito de facilitar o *debug* da gramática e eliminar as possíveis inconsistências gramaticais.

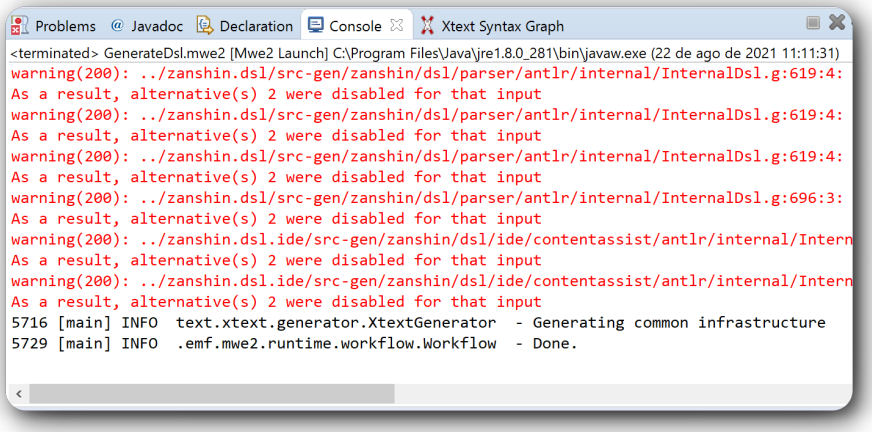


Figura 15 – Console do Eclipse™ com *logs* após a execução do MWE2.

Para a utilização da ferramenta ANTLRWorks, é necessário importar para esse software o arquivo que possui a representação ANTLR da gramática e, com isso, fornecer uma entrada para o software conseguir realizar uma simulação da linguagem a partir da entrada. Após essas configurações, é obtido um resultado gráfico que aponta as inconsistências da linguagem, além de marcar em vermelho qual entidade da gramática está com tal inconsistência, conforme a Figura 17.

Dessa forma, sabendo qual entidade apresentava inconsistências, além do gráfico

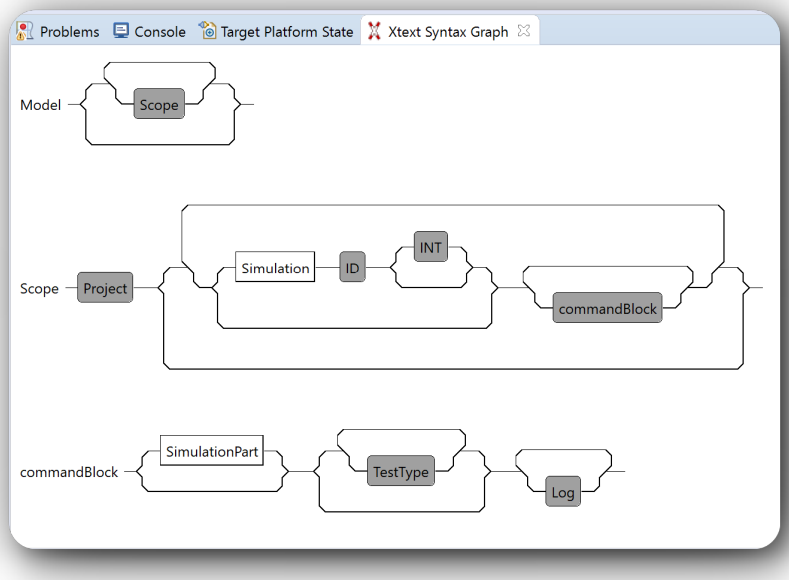


Figura 16 – Exemplos da representação gráfica das entidades da gramática da Zanshin DSL.

fornecido pelo ANTLRWorks guiar o entendimento dessas falhas, foram realizadas correções de forma iterativa até atingir uma linguagem consistente.

4.2 A DSL

O desenvolvimento de uma simulação utilizando a Zanshin DSL é feito em um arquivo de extensão própria *.zdsl*. A Figura 18 mostra um arquivo[3] dessa extensão, com simulações do projeto *Scheduler*, para exemplificar como essa linguagem pode ser utilizada. As simulações na Zanshin DSL iniciam com a *Keyword* Project sucedida de uma string que indica o nome do pacote de simulação. Após a *Keyword* Project, é obrigatório ter pelo menos uma *Keyword* Simulation sucedida de uma string que representa o nome do arquivo de projeto para o pacote de simulação definido pela *Keyword* Project. Após a *Keyword* Simulation, pode conter a *Keyword* SimulationPart que representa uma parte de simulação e, abaixo disso, pode conter a *Keyword* Failure precedida da string que representa o requisito do sistema que irá falhar, a *Keyword* Success precedida da string que representa o requisito do sistema que irá ter sucesso ou a *Keyword* Log que é a resposta do *Zanshin* que aparece no console durante a simulação de um sistema.

Assim, na Figura 18, é possível observar que para o pacote de simulação *scheduler*, serão geradas duas simulações distintas: a *SchedulerAR1FailureSimulation* e a *Schedule-*

[3] <https://github.com/iagohribeiro/Zanshin-DSL/tree/master/codeExamples>

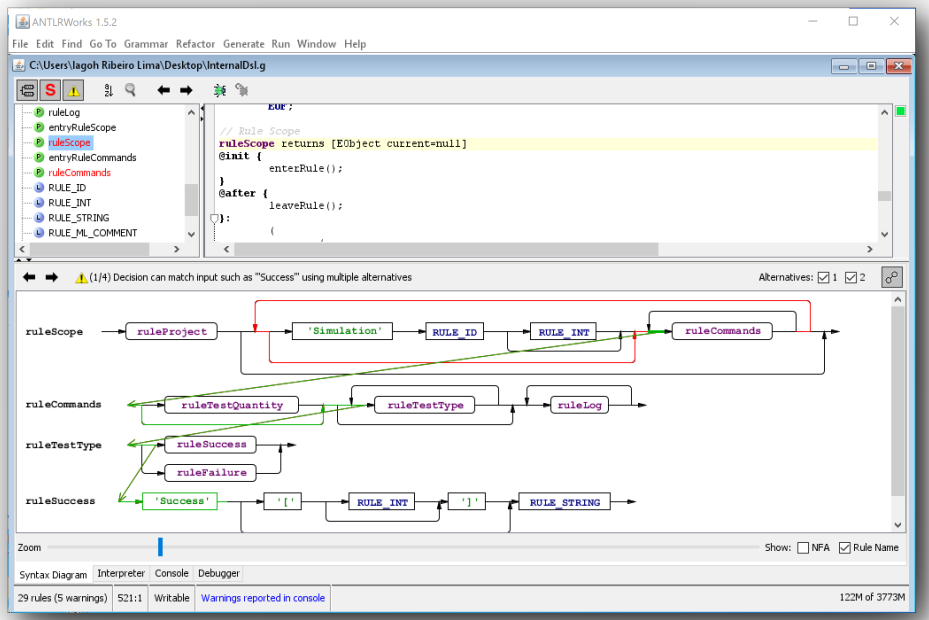


Figura 17 – Simulação da gramática da Zanshin DSL no software ANTLRWorks.

rAR4FailureSimulation. Além disso, o número 3 na linha 36 após a *Keyword* Simulation, representa em quantas partes essa simulação será dividida. Com isso, também é possível observar a *Keyword* SimulationPart que define uma parte de simulação e, embaixo disso, constam as *Keywords* Failure ou Success precedida dos requisitos que irão sofrer a ação que essas *Keywords* representam. Por fim, observa-se também a *Keyword Log* precedida da mensagem que o sistema irá disparar como resposta para aquela parte de simulação.

4.3 O Gerador de Códigos

Um projeto Xtext, *framework* utilizado para a construção da Zanshin DSL, já integra automaticamente o gerador de código na infra-estrutura de construção do Eclipse™. Assim, um projeto Xtext contém um arquivo Xtend que deve ser utilizado para implementar o gerador de código desejado. Essa integração possibilita que esse gerador de código seja chamado automaticamente a cada alteração realizada no documento com a extensão da DSL, não sendo necessária uma chamada por parte do usuário para esse gerador do código.

Para a Zanshin DSL, o gerador de código foi desenvolvido de forma que utilizasse as entradas a partir da gramática da DSL e, no final, produzisse arquivos Java passíveis de serem lidos pelo *Zanshin* (como os exemplificados na Seção 2.2). Basicamente o gerador de código percorre a coleção de elementos *Scope*, citado na Seção 4.1, obtendo os elementos da gramática e substituindo internamente nos moldes dos arquivos que serão gerados no

example.zdsl validation.zdsl

```
24 project "scheduler"
25 Simulation SchedulerAR1FailureSimulation 4
26     SimulationPart
27     Failure "T_CHARACT_MEET"
28     Log "Meeting organizer tries to Characterize meeting but it fails!"
29     SimulationPart
30     Failure "T_CHARACT_MEET"
31     Log "Meeting organizer tries *again* to Characterize meeting but it fails!"
32     SimulationPart
33     Failure "T_CHARACT_MEET"
34     Log "Meeting organizer tries *one more time* to Characterize meeting but it fails!"
35
36 Simulation SchedulerAR4FailureSimulation 3
37     SimulationPart
38     Failure "D_LOCAL_AVAIL"
39     Failure "T_CALL_PARTNER"
40     Failure "T_CALL_HOTEL"
41     Log "No rooms available and both partner and hotels fail!"
42     SimulationPart
43     Failure "D_LOCAL_AVAIL"
44     Failure "T_CALL_PARTNER"
45     Failure "T_CALL_HOTEL"
46     Log "No rooms available and both partner and hotels fail!"
47     SimulationPart
48     Success "T_CALL_PARTNER"
49     Log "This time, calling a partner institution succeeds!"
```

Figura 18 – Exemplo de um código feito em Zanshin DSL.

final dessa iteração.

A Figura 19 apresenta parte do código do Gerador de Códigos. Na linha 225 é possível observar que esse código se trata do método que gera o arquivo *Target System* de uma simulação. Além disso, nessa mesma figura, o texto em azul com sombra representa a parte textual do arquivo *Target System* que não irá se alterar para qualquer simulação gerada utilizando a Zanshin DSL. Porém, nas linhas 234 e 239 é apresentado o *token* projectName obtido da gramática da Zanshin DSL e que será exclusivo de cada simulação.

A Figura 20 apresenta parte do código do arquivo Java *SchedulerSimulatedTargetSystem* gerado pela Gerador de Códigos da Zanshin DSL. As linhas 7 e 12 dessa figura podem ser comparadas com as linhas 234 e 239 da Figura 19, respectivamente. Com isso, é possível observar como é realizado a substituição dos *tokens* apresentados na Figura 19.

Além da forma automática de invocar o gerador de código da Zanshin DSL, esse gerador foi exportado como um arquivo de extensão *jar* para possibilitar o usuário utilizá-lo fora do Eclipse™. Assim como alguns compiladores de outras linguagens (*GNU Compiler Collection* - GCC é um exemplo), que podem ser invocados de um terminal do

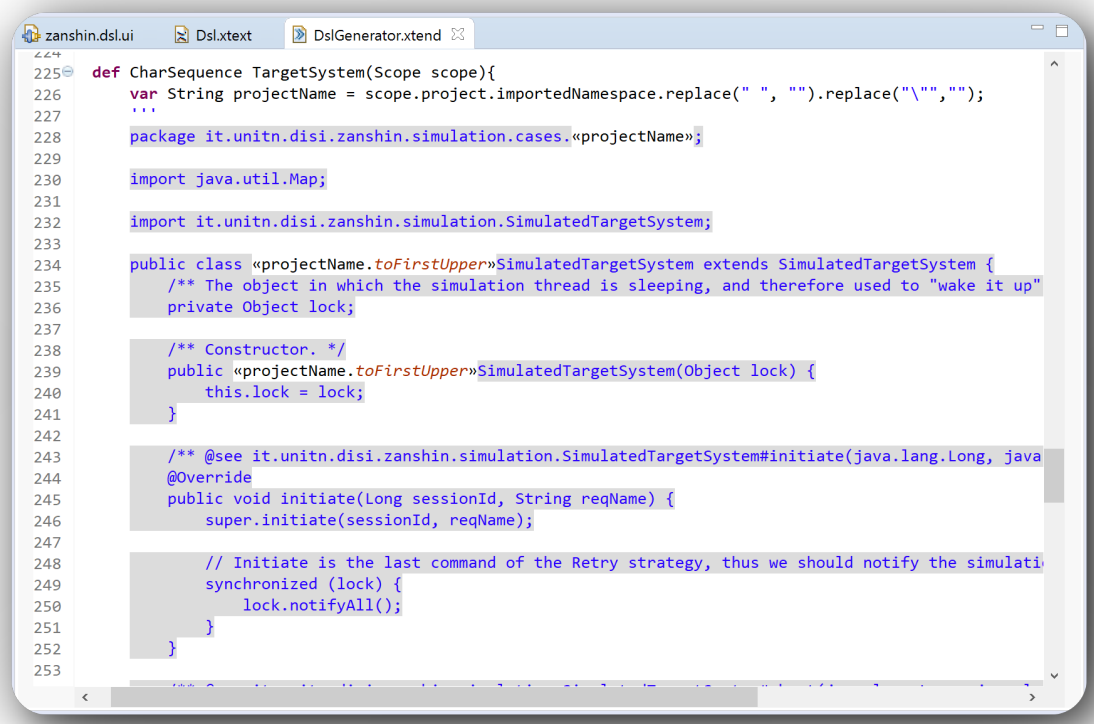
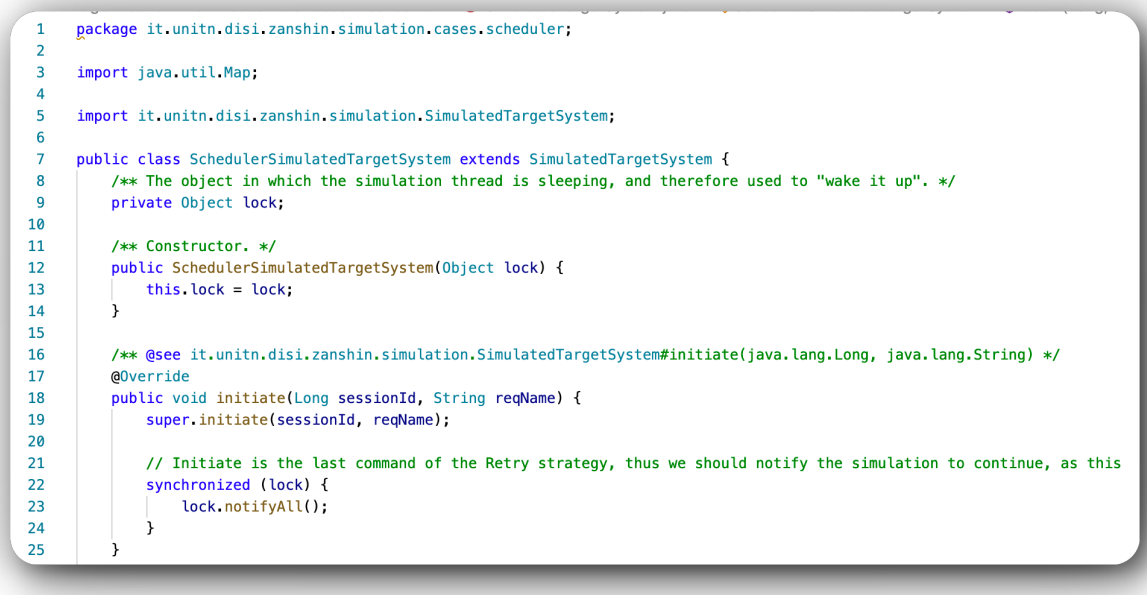


Figura 19 – Parte do código do Gerador de Códigos da Zanshin DSL.

Sistema Operacional do usuário, passando como parâmetro o arquivo suportado por aquele compilador, o gerador da Zanshin DSL pode ser invocado através de qualquer terminal como um produto independente (*standalone*).

O código do gerador de códigos e a versão *standalone* podem ser encontrados em <https://github.com/iagohribeiro/Zanshin-DSL/blob/master/zanshin.dsl/src/zanshin/dsl/generator/DslGenerator.xtend>.



```java
package it.unitn.disi.zanshin.simulation.cases.scheduler;

import java.util.Map;

import it.unitn.disi.zanshin.simulation.SimulatedTargetSystem;

public class SchedulerSimulatedTargetSystem extends SimulatedTargetSystem {
    /** The object in which the simulation thread is sleeping, and therefore used to "wake it up". */
    private Object lock;

    /** Constructor. */
    public SchedulerSimulatedTargetSystem(Object lock) {
        this.lock = lock;
    }

    /** @see it.unitn.disi.zanshin.simulation.SimulatedTargetSystem#initiate(java.lang.Long, java.lang.String) */
    @Override
    public void initiate(Long sessionId, String reqName) {
        super.initiate(sessionId, reqName);

        // Initiate is the last command of the Retry strategy, thus we should notify the simulation to continue, as this
        synchronized (lock) {
            lock.notifyAll();
        }
    }
```

Figura 20 – Parte do código *SchedulerSimulatedTargetSystem* gerado pela Zanshin DSL.

5 Validação

Nesse capítulo são apresentados os métodos utilizados para realizar a validação da Integração entre *Unagi* e *Zanshin*, além da validação da Zanshin DSL e seu gerador de códigos. A Seção **??** explica os passos da validação da Integração do *Unagi* e *Zanshin*, e a Seção **??** explica como foi feita a validação da Zanshin DSL e do gerador de códigos.

5.1 Integração

Para avaliar a integração, conduziu-se o seguinte experimento:

1. Os modelos de requisitos dos exemplos de cases disponíveis no módulo de simulação do *Zanshin* (Zanshin Simulation), ou seja, aqueles construídos manualmente antes da existência do *Unagi* e da DSL, tiveram o envio ao controlador do *Zanshin* desabilitado;
2. Tais modelos foram então reconstruídos de forma gráfica/diagramática dentro do *Unagi*, como pode ser observado no exemplo da Figura **??** para o exemplo do *Meeting Scheduler*;
3. O controlador *Zanshin* foi iniciado pelo *Unagi*, por meio do respectivo botão da *tab-bar*, apresentada no Capítulo 3;
4. Os modelos foram convertidos e enviados ao controlador pelo *Unagi* automaticamente, também por meio do respectivo botão na *tab-bar*;
5. O simulador foi executado através do *Unagi* e foram observados os mesmos resultados (instruções de adaptação) obtidos anteriormente com a execução manual das ferramentas separadas.

Durante a integração, foi explorado o pacote *gore.design* do projeto do *Unagi*. Ao realizar alterações dentro de um projeto no Eclipse™, alguns arquivos podem ser gerados automaticamente, afetando o projeto como um todo. Além disso, o arquivo de configuração do projeto também pode ser alterado, afetando o projeto. Para isso, foi realizado a construção de um modelo de sistema através do *Unagi* para ser utilizado antes e após sua integração com o *Zanshin* para garantir que nada foi alterado após a realização da integração.

A Figura **??** mostra o comparativo dos arquivos de modelo do sistema *Meeting Scheduler* gerados pelo *Unagi* antes e após a integração com o *Zanshin*. Como pode ser observado nessa figura, os arquivos possuem 100% de similaridade, demonstrando que a integração não impactou a geração dos arquivos modelo pelo *Unagi*.

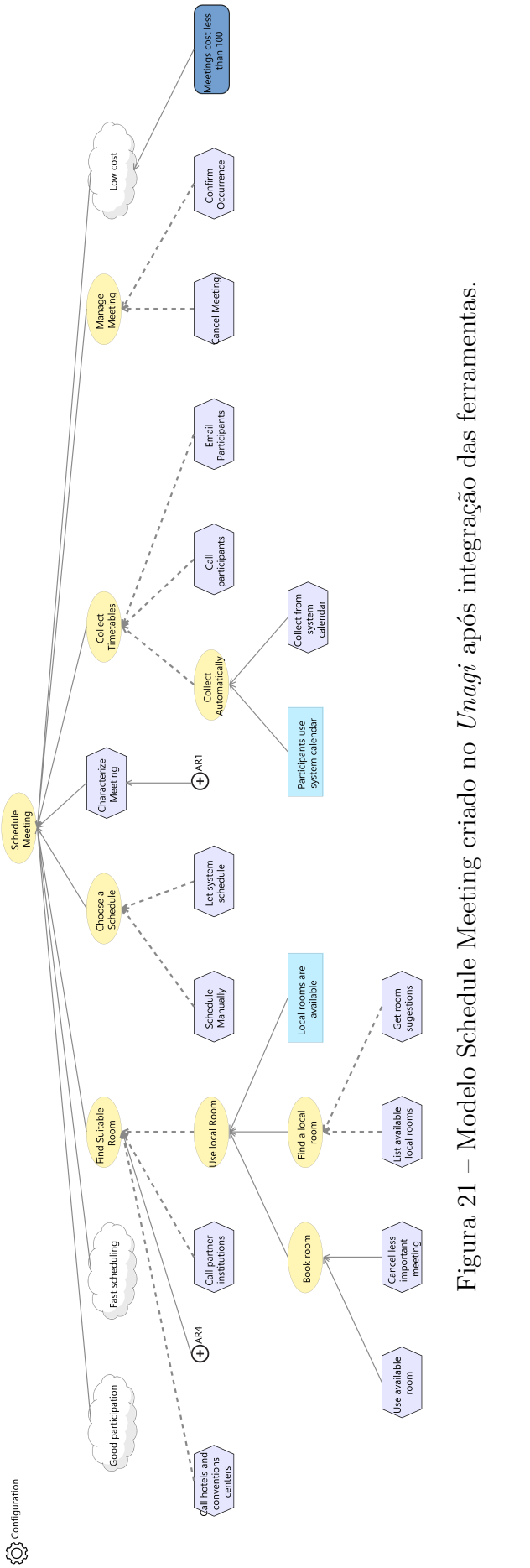


Figura 21 – Modelo Schedule Meeting criado no *Unagi* após integração das ferramentas.

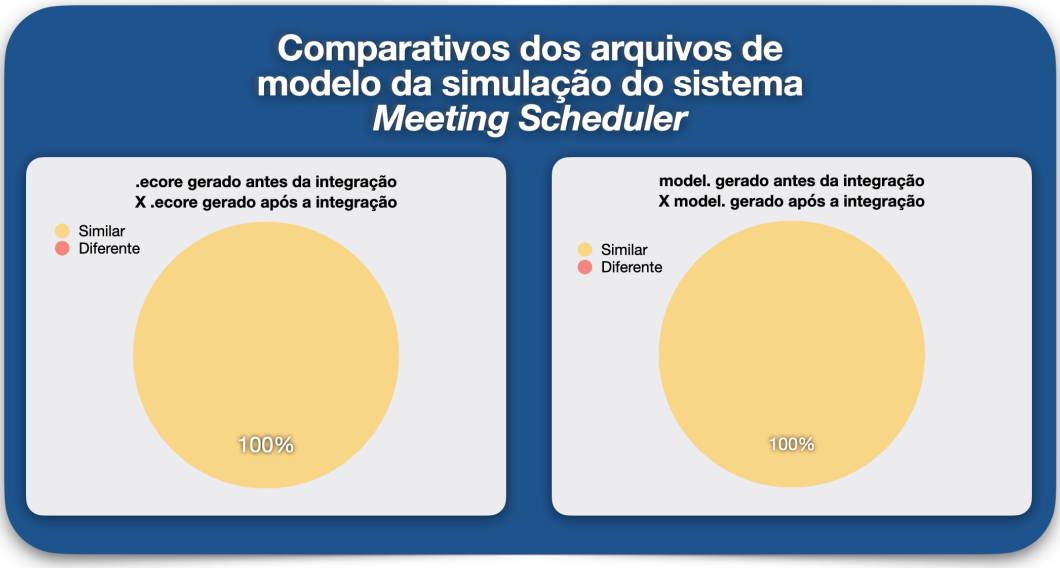


Figura 22 – Comparativos entre os arquivos de modelo do sistema *Meeting Scheduler* gerados antes e após a integração do *Unagi* e *Zanshin*.

5.2 Zanshin DSL

Para a validação da DSL, utilizaram-se os arquivos de simulação disponíveis no repositório do *Zanshin*[1] para comparação. Para cada simulação, foi criado um documento de extensão da DSL (*.zdsl*) e, para cada arquivo, foram adicionadas as entradas que atenderiam ao sistema da simulação. Os arquivos gerados pelo gerador de códigos da DSL foram comparados com os arquivos encontrados no repositório do *Zanshin*.

Na Seção 4.3 foram apresentadas as maneiras que o gerador de código da DSL é invocado. Sendo assim, para a validação desse gerador, duas formas distintas de validação foram utilizadas: o gerador de códigos sendo invocado automaticamente pela DSL dentro do Eclipse™ e o gerador de códigos utilizado como um produto independente (*standalone*) no terminal do Sistema Operacional.

O código da Listagem **??** foi utilizado para gerar os arquivos Java™ da simulação do sistema *Meeting Scheduler* através da Zanshin DSL. Com isso, foi possível comparar esses arquivos gerados com os que o repositório do *Zanshin* disponibiliza para essa simulação. O resultado dessas comparações pode ser visto na Figura **??**.

Essa figura mostra que nenhum dos arquivos possuem 100% de similaridade. Isso se dá pelo fato dos arquivos de simulação no repositório do *Zanshin* apresentarem alguns comentários que foram desconsiderados na implementação da DSL. Além disso, a ordem dos métodos desses aquivos podem ser diferentes dos que foram gerados pela DSL. Apesar dessas diferenças, não há impacto na simulação do sistema utilizando os arquivos gerados pela DSL.

[1] <https://github.com/sefms-disi-unitn/Zanshin/wiki>

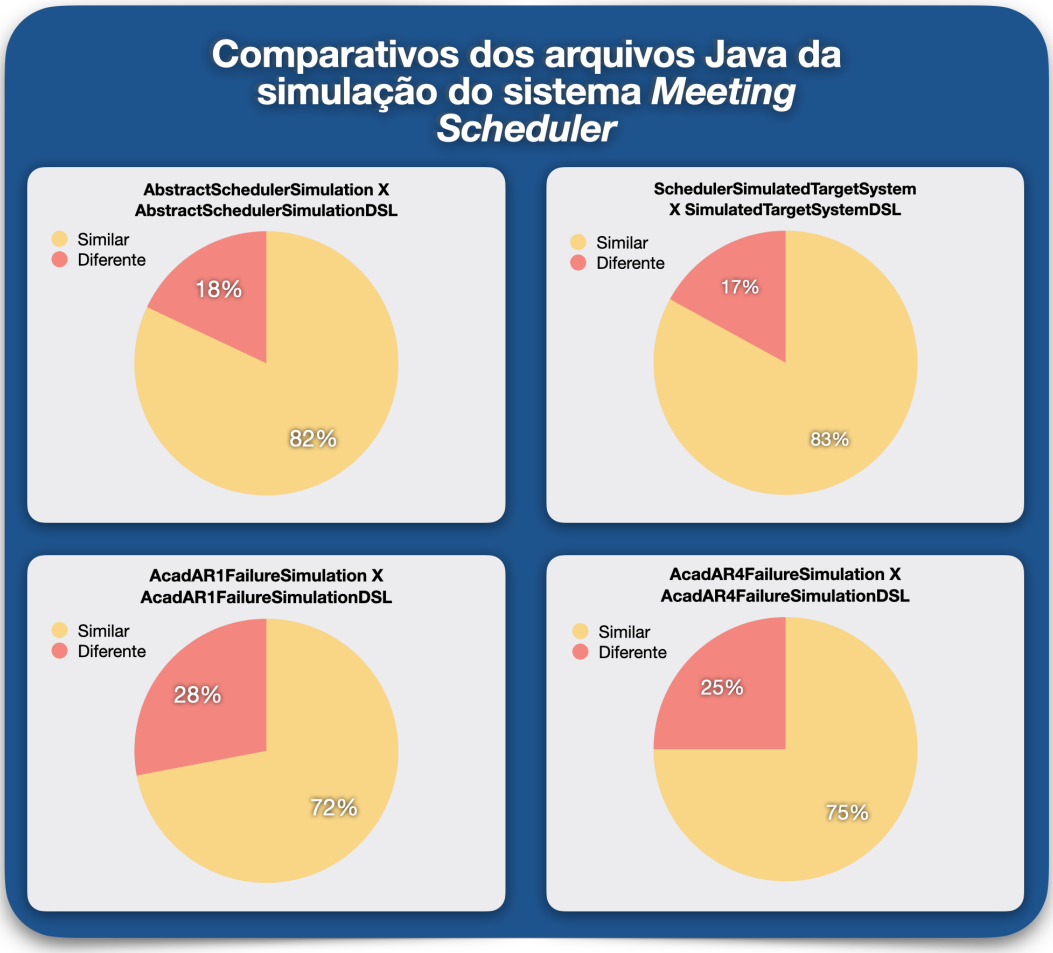


Figura 23 – Comparativos entre os arquivos Java™ da simulação exemplo *Scheduler* encontrada no repositório do *Zanshin* e os arquivos Java™ da mesma simulação geradas pela Zanshin DSL.

Outras simulações encontradas no pacote de simulação do *Zanshin* e seus correspondentes que foram gerados pela Zanshin DSL para a validação da DSL encontram-se nos links abaixo:

- Códigos do pacote de simulação do *Zanshin*: <https://github.com/nemo-ufes/Zanshin/tree/master/zanshin-simulations/src/it/unitn/disi/zanshin/simulation/cases>
- Códigos gerados pela Zanshin DSL: <https://github.com/iagohribeiro/Zanshin-DSL/tree/master/result/>

Listagem 5.1 – Código desenvolvido utilizando a Zanshin DSL para gerar arquivos da simulação *Scheduler*.

```
1  project "scheduler"
2  Simulation SchedulerAR1FailureSimulation 4
3    SimulationPart
4    Failure "T_CHARACT_MEET"
5    Log "Meeting organizer tries to Characterize meeting but it fails!"
6    SimulationPart
7    Failure "T_CHARACT_MEET"
8    Log "Meeting organizer tries *again* to Characterize meeting but it fails!"
9    SimulationPart
10   Failure "T_CHARACT_MEET"
11   Log "Meeting organizer tries *one more time* to Characterize meeting but it
         fails!"
12   SimulationPart
13   Log "The system aborts! Today is not a good day to schedule meetings..."
14 Simulation SchedulerAR4FailureSimulation 3
15   SimulationPart
16   Failure "D_LOCAL_AVAIL"
17   Failure "T_CALL_PARTNER"
18   Failure "T_CALL_HOTEL"
19   Log "No rooms available and both partner and hotels fail!"
20   SimulationPart
21   Failure "D_LOCAL_AVAIL"
22   Failure "T_CALL_PARTNER"
23   Failure "T_CALL_HOTEL"
24   Log "No rooms available and both partner and hotels fail!"
25   SimulationPart
26   Success "T_CALL_PARTNER"
27   Log "This time, calling a partner institution succeeds!"
```

6 Considerações Finais

Neste capítulo são apresentadas as limitações ao decorrer do trabalho, bem como pontos de melhorias a serem abordadas no futuro. A Seção 6.1 aborda a conclusão do trabalho, e então a Seção 6.2 discute as limitações e as perspectivas de trabalhos futuros.

6.1 Conclusão

O *Unagi* (BERNABÉ, 2017) traz a proposta de simplificar a criação e conversão de modelos para o *Zanshin*. A integração do mesmo com o *Zanshin* fornece um ecossistema completo (controlador, simulador, editor e conversor de modelos) e de código aberto, de modo que o desenvolvedor interessado na abordagem *Zanshin* não precise recorrer a softwares distintos para seu uso.

A execução de tarefas como a conversão de modelos dentro do *Unagi* e a inicialização do servidor RMI dentro do *Zanshin* foram facilitadas nessa integração que antes, quando as ferramentas encontravam-se separadas, não eram intuitivas para o usuário inicial. Assim, os usuários não têm o esforço de trabalhar em ambientes distintos, o que poderia prejudicar o tempo de produção e acarretar possíveis falhas.

Além disso, ainda dentro do mesmo ambiente, é possível inicializar a Zanshin DSL para o desenvolvimento dos arquivos necessários para a execução de uma simulação no *Zanshin* após a realização de uma modelagem no *Unagi*. Dessa forma, as pessoas interessadas em utilizar o *Zanshin* contam com uma experiência completa e intuitiva em um mesmo ambiente, dispensando o conhecimento em linguagens de programação convencionais, como o Java™.

6.2 Limitações e Perspectivas Futuras

Em um aspecto geral, considerando a integração entre *Unagi* e *Zanshin*, além da criação da Zanshin DSL, o Eclipse™ fornece um ambiente completo para o desenvolvimento de novas ferramentas. Porém, as atualizações dessa plataforma junto com seus *plugins* resulta, muitas vezes, na quebra de projetos desenvolvidos em versões mais antigas. Com isso, manter a compatibilidade dos projetos para versões novas do Eclipse™ se torna algo moroso e pode comprometer a manutenibilidade desses projetos.

Atualmente, todas as ferramentas desse trabalho rodam na versão Neon do Eclipse™, por limitações dos módulos do *Zanshin*. Espera-se então atualizar esses módulos para que seja possível utilizar essas ferramentas em versões mais novas do Eclipse™. Além dessa

melhoria, a Zanshin DSL pode ser expandida adicionando a ela validadores customizados para que o Eclipse™ avise ao desenvolvedor possíveis falhas no código de simulação que está sendo desenvolvido.

Ainda se tratando da Zanshin DSL, espera-se que ela se torne mais robusta no futuro, facilitando ainda mais sua utilização e aprendizado por parte do usuário. Para isso, será trabalhado no desenvolvimento de uma checagem de tipo, *quickfixes*, um escopo customizado, dentre outros elementos que contribua com esse objetivo. Além disso, pretende-se também evoluir os testes da linguagem e desenvolver testes que contribuam para coletar possíveis inconsistências da gramática.

Por fim, o *Unagi* é uma ferramenta bem intuitiva e fácil de se utilizar, porém, sua interface ainda apresenta alguns problemas de travamento e na ligação entre elementos durante a modelagem de um sistema. Dessa forma, pretende-se trabalhar na melhoria dessa interface, resolvendo essas inconsistências e recriando novos ícones, possibilitando a melhora do entendimento e a utilização da ferramenta pelo usuário.

Referências

ALMEIDA, J. P. A. *Model-driven design of distributed applications*. Tese (Doutorado) — University of Twente, The Netherlands, 2006. Citado na página 16.

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 14.

BERNABÉ, C. H. Evolução da Arquitetura do Zanshin - um framework para análise de requisitos em tempo de execução e desenvolvimento da ferramenta CASE unagi. Technical report, Projeto de Graduação, Departamento de Informática, Universidade Federal do Espírito Santo. 2017. Citado 9 vezes nas páginas 8, 9, 14, 15, 21, 26, 27, 28 e 48.

BERNABÉ, C. H. et al. *Unagi: Uma ferramenta para suporte à Modelagem de Requisitos de Sistemas Adaptativos. In Anais da Sessão de Ferramentas do 8º Congresso Brasileiro de Software: Teoria e Prática*. [S.l.]: (CBSoft 2017), pages 1-8, Fortaleza, CE, Brazil. SBC, 2017. Citado 2 vezes nas páginas 16 e 31.

BETTINI, L. *Implementing Domain-Specific Languages with Xtext and Xtend: Second Edition*. [S.l.]: Packt Publishing, 2016. 397 p. ISBN 9781786464965. Citado 5 vezes nas páginas 15, 16, 28, 29 e 30.

BOVET, J. *ANTLRWorks: The ANTLR GUI Development Environment*. 2021. <https://www.antlr3.org/works/>. Acesso: 10/05/2021. Citado na página 29.

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 na página 15.

DARDENNE, A.; LAMSWEERDE, A. van; FICKAS, S. Goal-directed requirements acquisition. *Science of Computer Programming*, v. 20, n. 1-2, p. 3–50, apr 1993. Citado na página 30.

HORKOFF, J.; YU, Y.; ERIC, S. OpenOME: An Open-source Goal and Agent-Oriented Model Drawing and Analysis Tool. *iStar*, v. 766, p. 154–156, 2011. Citado 2 vezes nas páginas 16 e 30.

KEPHART, J.; CHESS, D. The vision of autonomic computing. *Computer*, v. 36, p. 41 – 50, 02 2003. Citado na página 14.

MORANDINI, M. et al. Tool-supported development with tropos: The conference management system case study. In: SPRINGER. *International Workshop on Agent-Oriented Software Engineering*. [S.l.], 2007. p. 182–196. Citado 2 vezes nas páginas 16 e 30.

MORANDINI, M.; PERINI, A.; MARCHETTO, A. Empirical Evaluation of Tropos4AS Modelling. In: *Proc. of the 5th International i* Workshop (iStar 2011)*. [S.l.]: CEUR, 2011. p. 14–19. Citado na página 30.

MYLOPOULOS, J.; CHUNG, L.; NIXON, B. Representing and Using Nonfunctional

Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering*, v. 18, n. 6, p. 483–497, jun 1992. Citado na página 30.

PASTOR, O. et al. Model-driven development. *Informatik-Spektrum*, v. 31, p. 5, 2008. Citado na página 16.

PERIŠIĆ, B. Model driven software development – state of the art and perspectives. In: . [S.l.: s.n.], 2014. Citado na página 19.

PIMENTEL, J.; CASTRO, J. pistar tool–a pluggable online tool for goal modeling. In: IEEE. *2018 IEEE 26th International Requirements Engineering Conference (RE)*. [S.l.], 2018. p. 498–499. Citado 2 vezes nas páginas 16 e 30.

PIMENTEL, J. H. C. *Systematic design of adaptive systems: control-based framework*. Tese (Doutorado) — Universidade Federal de Pernambuco, Pernambuco, PE, Brasil, 2015. Citado 2 vezes nas páginas 16 e 30.

SABATUCCI, L.; SEIDITA, V.; COSSENTINO, M. The four types of self-adaptive systems: A metamodel. In: . [S.l.: s.n.], 2018. p. 440–450. ISBN 978-3-319-59479-8. Citado na página 14.

SELIC, B. The pragmatics of model-driven development. *IEEE software*, IEEE, v. 20, n. 5, p. 19–25, 2003. Citado 2 vezes nas páginas 14 e 19.

SIRIUS. *Sirius – Provide tab-bar extensions*. 2017. <https://www.eclipse.org/sirius/doc/developer/extensions-provide_tabbar_extensions.html>. Acesso: 20/11/2018. Citado 2 vezes nas páginas 17 e 32.

SOUZA, V. E. S. *Requirements-based software system adaptation*. [S.l.]: Tese (Doutorado) - University of Trento, 2012. Citado 5 vezes nas páginas 8, 14, 15, 16 e 21.

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 2 vezes nas páginas 21 e 22.

SOUZA, V. E. S.; MYLOPOULOS, J. Designing an adaptive computer-aided ambulance dispatch system with Zanshin: an experience report. *Software: Practice and Experience*, Wiley, v. 45, n. 5, p. 689–725, may 2013. ISSN 00380644. Disponível em: <http://doi.wiley.com/10.1002/spe.2245>. Citado 2 vezes nas páginas 14 e 21.

STEINBERG, D. et al. *EMF: eclipse modeling framework*. [S.l.]: Pearson Education, 2008. Citado na página 20.

SUPAKKUL, S.; CHUNG, L. The re-tools: A multi-notational requirements modeling toolkit. In: IEEE. *2012 20th IEEE International Requirements Engineering Conference (RE)*. [S.l.], 2012. p. 333–334. Citado 2 vezes nas páginas 16 e 30.

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 (SEAMS 2013)*. IEEE, 2013. p. 93–102. Disponível em: <http://dl.acm.org/citation.cfm?id=2487353>. Citado na página 21.

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 4 vezes nas páginas 14, 19, 20 e 24.

VOGEL, L. *Eclipse Modeling Framework (EMF) - Tutorial*. 2016. <http://www.vogella.com/tutorials/EclipseEMF/article.html>. Acesso: 12/10/2018. Citado na página 16.

VOGEL, L. *Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions*. 2018. <http://www.vogella.com/tutorials/EclipsePlugin/article.html>. Acesso: 10/11/2018. Citado na página 16.

VOGEL, L. *Eclipse RCP (Rich Client Platform) - Tutorial*. 2018. <http://www.vogella.com/tutorials/EclipseRCP/article.html>. Acesso: 12/12/2018. Citado na página 16.

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 na página 19.

XTEND. *Xtend*. 2021. <https://www.eclipse.org/xtend/>. Acesso: 10/02/2021. Citado na página 29.