Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DEVS2BDD: Um arcabouço para a geração de casos de testes BDD no simulador MS4Me

Pedro A. R. Duarte

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientadora
Prof.a Dr.a Genaina Nunes Rodrigues

Brasília
2022

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DEVS2BDD: Um arcabouço para a geração de casos de testes BDD no simulador MS4Me

Pedro A. R. Duarte

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof.a Dr.a Genaina Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. Dr. Breno Miranda
CIn/UFPE

Prof. Dr. João Luiz Azevedo de Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 22 de setembro de 2022

Dedicatória

Eu dedico este trabalho à minha família que proporcionou um ambiente prospero para os estudos, feliz e saudável durante todas as etapas da minha vida. Dedico também aos meus amigos João Pedro Assis dos Santos e Waliff Cordeiro Bandeira, que foram essenciais para minha graduação. E por fim, dedico a minha namorada Carla Cristina de Lima Brasil.

Agradecimentos

Agradeço à Prof.a Dr.a Genaina Nunes Rodrigues pela orientação durante esse trabalho, pelos ensinamentos na área de engenharia de software e oportunidade de agregar em seu grupo de pesquisa. Agradeço ao grupo de pesquisa LADECIC. Agradeço aos meus companheiros de curso João Pedro Assis dos Santos, Waliff Cordeiro Bandeira e Estevan Alexander de Paula que lutaram junto a mim nessa jornada. Por fim, agradeço, em especial, a minha mãe, Ana Karenina Silva Ramalho Andrade, e a minha família, pelo suporte em todos os aspectos desse sonho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

A simulação e modelagem é uma etapa que traz vários benefícios para o processo de desenvolvimento de softwares, principalmente quando o assunto é *System of Systems*s (SoSs) . Porém, é preciso aceitar que há grandes desafios que dificultam sua democratização. O foco desse trabalho é solucionar o problema da desconexão entre a etapa de *Modeling and Simulation* (M&S) e a etapa de codificação. Para isso, foi desenvolvida uma CLI, nomeada DEVS2BDD, com o objetivo de receber, como entrada, os artefatos do software de simulação e modelagem *MS4Me* e devolver arquivos com cenários de testes BDD do sistema apresentado. Os cenários de testes gerados se mostraram satisfatórios para descrever a missão do SoS. Contudo, na análise individual de cada subsistema, o cenário ficou prolixo e de difícil leitura para o desenvolvedor.

Palavras-chave: BDD, simulação e modelagem de software, DEVS, MS4Me, testes automatizados, SoS

Abstract

Modeling and Simulation (M&S) is a step that brings several benefits to the software development process, especially when it comes to *System of Systems* (SoS). However, it is necessary to accept that there are great challenges that hinder its democratization. The focus of this work is to solve the problem of disconnection between the M&S and the coding process. For this, it was built a CLI named DEVS2BDD, which goal is to receive, as input, the artifacts from MS4ME simulation and modeling software, and to return BDD test scenarios of the system. The generated test scenarios proved to be satisfactory on describing SoS mission. Nevertheless, in the individual analysis for each subsystem the scenario became prolix and difficult to read.

Keywords: BDD, SPSM, DEVS, MS4Me, automateded testing, SoS

Sumário

Lista de Figuras

Lista de Códigos

Lista de Abreviaturas e Siglas

BDD *Behavior-Driven Design.*

BSN *Body Sensor Network.*

CLI *Comamand-line Interface.*

DEVS *Discrete Event System Specification.*

DNL *DEVS Natural Language.*

DSL *Domain-Specific Language.*

M&S *Modeling and Simulation.*

MSA *Microservice Architectures.*

SES *System Entity Structure.*

SoS *System of Systems.*

SoSE *Systems of Systems Engineering.*

SPSM *Software process simulation modeling.*

TDD *Test-Driven Development.*

Capítulo 1

Introdução

Softwares estão presentes em todos os aspectos do nosso cotidiano e, cada vez mais, esperamos que eles resolvam uma maior gama de problemas. Tais sistemas que inicialmente eram simples, por precisar atender novas demandas, acabam se tornando mais complexos e adquirindo novos problemas, tais como: a expressiva quantidade de elementos, variáveis, não linearidade e grande acoplamento [1]. Os citados novos problemas não são resolvidos eficientemente pela tradicional arquitetura monolítica. Uma das possíveis soluções, é a utilização de um novo paradigma de sistemas de sistemas para o design de novos softwares.[1]

Um sistema de sistemas é um sistema composto pela união de outros sistemas independentes que, quando conectados, possuem o objetivo de atender novas funcionalidades[2]. Contudo, com essa forma de compor sistemas, surgem novos desafios técnicos a serem enfrentados, como, por exemplo, o monitoramento, planejamento e *design*. Em decorrência disso, práticas que antes não se apresentavam tão vantajosas, acabam se tornando recomendáveis. Uma dessas novas práticas recomendáveis é a etapa de modelagem e simulação. Dada a complexidade inerente ao combinar sistemas com propriedades emergentes e a natureza abstrata dos sistemas de sistemas, é necessário um software de simulação e modelagem para descrever, especificar e planejar o desenvolvimento deste. [3]

Outro mecanismo que é de bastante importância no desenvolvimento de software são os testes [4]. Na hipótese de *System of Systems* (SoS) que possuem uma maior complexidade natural tendem a mais chances de falhas. Nesse sentido, a etapa de teste é crucial. O teste de software é um processo que tem como o objetivo de validar se certo produto atingiu suas especificações [5] e, geralmente, é uma etapa que demanda um esforço constante da equipe de desenvolvimento. Normalmente, são investidos mais de 50% do tempo de desenvolvimento em testes [6] por isso é necessária uma atenção especial para aumentar a produtividade nessa etapa.

Dessa forma, a etapa de modelagem e simulação e a etapa de testes são fundamentais

para o desenvolvimento e verificação [7] de sistemas de sistemas. Este trabalho tem, portanto, como objetivo estudar a transição entre as etapas citadas para melhorar a produtividade e facilitar a implementação dessa classe de sistemas apoiado pela prática de testes.

1.1 Problema de pesquisa

Na construção de SoS é fundamental diminuir a distância entre o projeto, a verificação e implementação do SoS. A simulação e modelagem, como etapa de verificação de sistemas, é realizada para atender os seguintes propósitos [8]:

- Gerenciamento estratégico.
- Planejamento.
- Controle e gerência operacional.
- Melhorias de processos e adoção de tecnologia.
- Melhorar o entendimento sobre o projeto.
- Treinamento e aprendizado.

A etapa de simulação visa reduzir o risco e ajudar no planejamento do projeto. No entanto, tal etapa ainda apresenta lacuna em permitir integração direta entre o resultado da simulação com o desenvolvimento do software. Consequentemente, isso pode tornar a adoção de ferramentas de simulação como MS4Me[9] menos atrativa.

Outro fator que diminui a integração entre essas duas etapas é que os artefatos proveniente da primeira geralmente não seguem um padrão bem definido. Cada software de simulação possui o seu formato específico, o que dificulta o reuso automatizado do resultado da simulação.

1.2 Objetivos gerais

O objetivo principal desse trabalho é aumentar a integração entre a etapa de simulação e a etapa de desenvolvimento de projetos. A solução proposta neste trabalho é gerar artefatos de teste, a partir da simulação, que serão úteis para o processo de implementação do *softwares*. Os testes foram escolhidos como esses artefatos de ligação entre as etapas (de simulação e desenvolvimento), já que são de grande valor para o estágio inicial de codificação e auxiliam na adoção de boas práticas como o *Behavior-Driven Design* (BDD) [10] e *Test-Driven Development* (TDD) [11].

Neste trabalho, é explorado mais especificamente as saídas do *Software MS4Me* da empresa *MS4 Systems* [9], que será mais detalhado nas seções abaixo. Foi observado que seus artefatos gerados tinham grande potencial para servirem de base para geração de casos de testes. Dessa forma, foi escolhido especialmente gerar casos de comportamento, uma vez que são simulados softwares mais complexos integrando vários sistemas. Dadas as características de *Behavior-Driven Design* apropriada para testes de aceitação e testes de integração, principalmente este último no contexto de SoS, BDD se monstra uma prática muito apropriada a ser adotada no processo de desenvolvimento de SoS.

Portanto, o objetivo deste trabalho consiste em gerar casos de testes de comportamento, tendo como entrada o resultado da etapa de simulação do SoS. A partir dos testes BDDs gerados, espera-se contribuir com o desenvolvimento correto por construção do SoS. E assim, alavancar a utilidade e valor da etapa de simulação, aumentando a produtividade do desenvolvimento de software por meio da adoção de boas práticas de software desde o princípio do desenvolvimento.

1.3 Objetivos específicos

Para a contemplação do objetivo geral que concentra em minimizar o evidente afastamento entre a etapa de simulação e a implementação do SoS, foram escolhidos os seguintes objetivos específicos:

- Estudar e analisar a viabilidade de geração de casos de testes BDD a partir de *softwares* para *Software process simulation modeling* (SPSM).
- Gerar casos de testes de comportamento a partir do projeto exportado pelo *software MS4Me*.
- Desenvolvimento de uma CLI para facilitar o uso da ferramenta.
- Avaliar a qualidade dos testes proveniente da ferramenta.

1.4 Organização do trabalho

O Capítulo 1 introduz o leitor a falta de relação entre o desenvolvimento e a simulação de *software* e, em sequência, disserta sobre uma possível solução para esse problema utilizando as saídas dos *software* de simulação para gerar casos de testes de comportamento.

O Capítulo 2 descreve a fundamentação teórica necessária para o entendimento dos conceitos adotados ao longo desta monografia. Abordando os quatro conhecimentos fundamentais deste trabalho. A explicação inicia-se pela simulação de *software* em si que

é ponto de partida para o nosso produto. Em seguida, é caracterizado o termo *System of Systems* (SoS) e a importância da simulação para esse tipo de sistema. Na sequência, é abordado o conceito de testes de comportamento e por fim é abordado o software de simulação utilizado, o *MS4Me*. É explicado seu funcionamento em detalhes utilizando a simulação de um *System of Systems* como exemplo.

O Capítulo 3 tem a função de explicar a proposta principal deste trabalho. Primeiramente, é demonstrado o fluxo de uso da da ferramenta DEVS2BDD, passando por suas funcionalidades, arquitetura de implementação e o porquê de da escolha da linguagem *Ruby* [12] como linguagem de programação. Logo em seguida é demonstrado o exemplo do sistema de caixa eletrônico que guia a explicação das funcionalidades do produto. Por fim, é explicado, utilizando o exemplo guia, cada comando da CLI em detalhes adentrando na lógica de geração de casos de testes .

O Capítulo 4 demonstra o estudo de caso feito no protótipo *Body Sensor Network* (BSN). Este capítulo explica, brevemente, o conceito da BSN e depois são demonstrados os cenários de testes gerados pelo DEVS2BDD a partir da simulação do sistema. Por fim, os casos de testes são avaliados tendo em vista sua utilidade prática para BSN.

O Capítulo 5 finaliza o trabalho com as conclusões restantes e analisa o resultado do projeto.

Capítulo 2

Fundamentação Teórica

Este capítulo descreve a fundamentação teórica necessária para o melhor entendimento do projeto.

2.1 Simulação de Software

Simular é o ato de reproduzir um processo ou sistema ao longo do tempo. Essa prática é de extrema importância para reduzir riscos de acidentes, custos de operações, realizar treinamentos e outros tipos de benefícios. Cita-se como exemplos da necessidade do processo de simulação as seguintes situações: seria imprudente permitir que um aviador pilote uma aeronave sem antes realizar um treinamento em um software de simulação de voo. Outra área em que a simulação é essencial é a militar, já que para uma reprodução de campo de batalha é necessário um grande investimento e há riscos, motivo pelo qual o departamento de defesa do Estados Unidos investe bilhões de dólares anualmente em modelagem e simulação. [13]

Na área da computação, a simulação e modelagem é uma prática opcional, que pode ser inserida na etapa de *design*, durante o desenvolvimento de software, e é caracterizada por representar o sistema e descrever como ele se comporta ao longo do tempo [14]. Embora não obrigatória, é uma etapa que traz várias vantagens para o desenvolvimento de produtos como: análise de risco, melhor planejamento a longo prazo, ambiente para geração de novas ideias etc. [15] [16]

Dado todos esses benefícios ela poderia ser mais utilizada durante a fabricação de softwares. Porém, é preciso reconhecer que existem vários empecilhos que dificultam sua democratização, como [14] [17] [16]:

- Necessidade de conhecimento específico em simulação e modelagem;
- grande investimento de tempo;

- dificuldade em reusar modelos;
- falta de ferramentas reusável para visualização e análise;
- falta de integração com o ambiente de implementação;

O trabalho apresenta, mais especificamente, os softwares de simulação *Discrete Event System Specification* (DEVS) que serão melhor introduzidos na Seção 2.2. No mercado exite uma gama de softwares de simulações DEVS como aDEVS, PowerDEVS, MS4Me, PyDevs e DEVS-Ruby [18]. O projeto teve seu foco voltado para o estudo do software MS4Me pela pequena barreira de entrada, sua capacidade de tanto definir pequenos fluxos de tarefas até grande sistemas de sistemas [19] e o seu uso em outros projetos do grupo de pesquisa LADECIC (Laboratório de Dependabilidade do CIC).

Em via de regra, todo software de simulação e modelagem DEVS não possui uma integração com o ambiente de implementação, porque seus artefatos de saída são interfaces gráficas da simulação em execução (em formato de vídeo ou foto) ou então arquivos *Domain-Specific Language* (DSL), diferentes para cada software. No caso do MS4Me, software de simulação e modelagem utilizado no projeto, os seus artefatos resultados são exportados em formato de arquivos SES, DNL e java que também não são compatíveis com a etapa de codificação. Contudo, o conteúdo desses arquivos são bem descritivos e caso estivessem em outra formatação, como em formato de cenários de testes, poderiam ser úteis para o desenvolvimento de aplicações. Esse é justamente o foco principal do trabalho: transformar o formato das DSL do MS4Me em um formato universal de BDD.

2.2 DEVS

Discrete Event System Specification (DEVS) *formalism* é um conceito inventado por Zeigler [20] que define um conjunto de regras para modelagem e simulação com eventos discretos. Esse formalismo é usado como base para diversos de softwares de simulação e modelagem. No conceito clássico há dois tipos de modelo os *Atomic* e o *Coupled*[21][22]

2.2.1 Modelo Atómico *(Atomic Model)*

Atomic Model é a menor estrutura do formalismo DEVS, na qual especifica o comportamento da entidade e pode ser definida pela seguinte função:

$$M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{2.1}$$

- X é o conjunto de portas ou valores de entradas

- S representa o conjunto de estados
- Y é o conjunto de portas ou valores de saídas
- δ_{int} é a função de transição interna
- δ_{ext} é a função de transição externa
- λ é a função de saída
- $ta(s) :\rightarrow \Re_0^+ \cup \infty$ é a função de avanço de tempo

2.2.2 Modelo Composto *(Coupled Model)*

Coupled Model é um conjunto de modelos atômicos que descreve a estrutura e a hierarquia da modelagem. Esse modelo pode ser definido pela seguinte função matemática:

$$N =< X, Y, D, EIC, EOC, IC > \tag{2.2}$$

- X é o conjunto de eventos de entrada externa
- Y é o conjunto de eventos de saída
- D é o conjunto de componentes atômicos
- EIC é a relação de acoplamento de entrada externa
- EOC é a relação de acoplamento de saída externa
- IC é a relação de acoplamento interno

Como vantagens de seu uso, pode-se citar os vários tipos de implementações, o fato de ser aberto para extensões, possuir uma definição explicita do comportamento do modelo e uma estrutura modularizada e hierarquizada. [23]

No DEVS os eventos são o foco principal da simulação, eles que determinam o que deve aparecer nas saídas. A descrição do modelo deve determinar como responder os eventos que chegam de fora e os eventos internos mudam o estado do modelo e transmitem os eventos nas portas de saída. [24]

2.3 Sistemas de Sistemas

Sistemas de sistemas é um termo muito utilizado e tem um significado amplo, cada autor tem uma forma de definir essa classe de sistemas. A mais descritiva delas: SoS é a integração em larga escala de sistemas heterogêneos e operáveis de forma independente, mas estão contactados para um objetivo comum. [25] [26]

O interesse nessa classe de sistemas cresceu muito nos últimos anos porque características como robustez, alta performance, confiabilidade e a junção de vários sistemas se tornaram requisitos de softwares modernos. [25]

Outro fator que aumenta a procura por essa classe de sistemas é a arquitetura baseada em microsserviços, que hoje é referência para suportar as constantes mudanças nas aplicações modernas e uma maior escalabilidade. Nessa arquitetura se desenvolve aplicações unindo um conjunto de pequenos serviços, cada um rodando em seu único processo e comunicando entre si por leves mecanismos, geralmente HTTP. Esses serviços são independentes e existe uma pequena central os ligando [27]. Por consequência de utilizar o *Microservice Architectures* (MSA) é possível alcançar as propriedades de SoS [28]. Por isso, indiretamente, ao utilizar essa arquitetura os arquitetos se deparam com os conceitos de SoS.

Embora tenha muitas definições, os SoSs possuem as seguintes 5 características que podem ser utilizadas para distinguir um sistema comum e um SoS [29]:

- Independência operacional: Cada sistema do SoS deve funcionar independente com seus próprios objetivos e não possuir interferência na funcionalidade do sistema vizinho;
- independência gerencial: Cada sistema é responsável por sua própria operação e não deve ser perturbado pelo SoS;
- desenvolvimento evolucionário: Ao longo de seu desenvolvimento o SoS pode acoplar mais serviços ou então não utilizar sistemas que não são mais necessários;
- comportamento emergente: Todos os sistemas em conjunto precisam realizar um ou mais objetivos em comum que não pode ser realizado por somente um deles. E podem surgir comportamos emergentes em tem de execução;
- distribuídos: Podem ser distribuídos geograficamente ou não;

Em 2012, no INCOSE, foi realizada um pesquisa sobre os pontos de dores ao lidar com um SoS. Um dos 7 pontos de dores apontados na área de *Systems of Systems Engineering* (SoSE) é a parte de validação e testes [30]. Cada característica dessa classe de sistema dificulta a etapa de verificação e validação do desenvolvimento de softwares.

Como consequência da independência operacional muitas das funções de um serviço não são utilizadas para completar a missão do SoS, então é necessário ter uma seleção de casos de testes dos serviços, estabelecer critérios de aceitação e definir o que será testado [31].

A característica de independência gerencial agrava ainda mais a dificuldade da implementação de testes, já que, por consequência dessa característica, é preciso atentar-se

com o estados dos serviços, a confiabilidade, a interoperabilidade e disponibilidade para a execução e fabricação dos testes. [31].

2.4 Testes de Comportamento

O desenvolvimento orientado a comportamento é um processo de desenvolvimento de software que foi idealizado por Dan North, com o objetivo de melhorar o *Test-Driven Development* (TDD) [10]. É um processo que visa criar cenários de execução de uma funcionalidade em linguagem com um vocabulário comum, que servirão de guia para o desenvolvimento do software.

Esses cenários podem ser transformados em casos de testes automatizados e ao mesmo tempo servir de documentação, sendo assim tem-se uma documentação viva para o nosso software. Essa última vantagem também resolve outro problema no fluxo de desenvolvimento que é a defasagem da documentação e a implementação. Como a documentação, geralmente, é algo à parte do código, ela não precisa estar correta para o software funcionar. Porém, com o *Behavior-Driven Design* (BDD), ela necessita sempre ser atualizada para execução dos testes obterem êxito.

Para o nosso produto, foi escolhida a linguagem Gherkin [32] como formato de saída, pois ela é a mais famosa entre as linguagens de BDD. Uma das melhores características dessa linguagem é que pessoas técnicas ou não podem escrever e/ou entender os cenários de testes, transformando-a em uma documentação para o seu projeto e ajudando a melhorar a comunicação entre a equipe de desenvolvimento e a equipe de produto.

Além do mais, a linguagem Gherkin é suportada pelo eco sistema do Cucumber [33], que é o mais famoso quando nos referimos ao BDD. O Cucumber possui suporte para várias plataforma/linguagens de programação, integração com o Selenium e demais ferramentas. Possui exportação de relatórios para diversos formatos e suporta vários idiomas.

Sua sintaxe em português contém as seguintes palavras chaves essenciais para um caso de teste de comportamento: "Dado", "Quando", "Então", é possível observar abaixo um exemplo de cenário de teste hipotético.

```
# features/meal.feature

Funcionalidade: Alimentação # Nome da funcionalidade
  Cenário: Comer 5 das 12 maças
    Dado que existem 12 maças # Estado inicial
    Quando eu comer 5 maças # Ação
    Então eu deveria ter 7 maças # Resultado ou evidência
```

Código 1: Exemplo da sintaxe Gherkin em ruby

Para definições dos passos o Cucumber permite a utilização de uma grande diversidade de linguagens de programação. No exemplo abaixo é utilizado o *ruby*:

```
# features/step_definitions/meal_steps.rb

Dado("que existem {integer} maças") do |qte_macas|
  @qte_macas = qte_macas
end

Quando("eu comer {integer} maças") do |qte_macas|
  @qte_macas -= qte_macas
end

Então("eu deveria ter {integer} maças") do |qte_macas|
  expect(@qte_macas).to eq(qte_macas)
end
```

Código 2: Implementação das definições do Gherkin em ruby

2.5 Software de Simulação (MS4)

O software MS4 Modeling Environment [9] foi escolhido como primeiro *software* a ser suportado pelo DEVS2BDD. É uma ferramenta implementada em java [34] usando como base o editor de texto Eclipse [35] e que tem sido utilizada na modelagem de sistemas complexos, principalmente, SoSs. A ferramenta citada é interessante porque possibilita realizar uma simulação de eventos discretos e utilizar as cadeias de markov [36]. Um dos maiores exemplos de software que foram modelados com essa ferramenta é o sistema de saúde dos Estados Unidos [37]

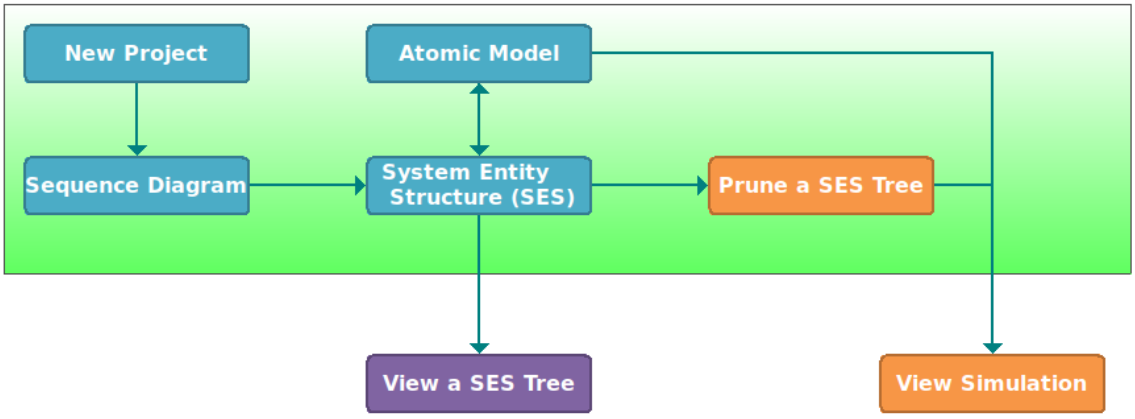


Figura 2.1: Fluxo de Trabalho do MS4Me

Para iniciar a simulação é necessário criar, por meio de uma interface gráfica, as entidades do sistema e preencher as trocas de eventos realizadas entre os mesmos. Essa etapa é realizada através do ícone *Sequence Diagram* do fluxo de trabalho 2.1.

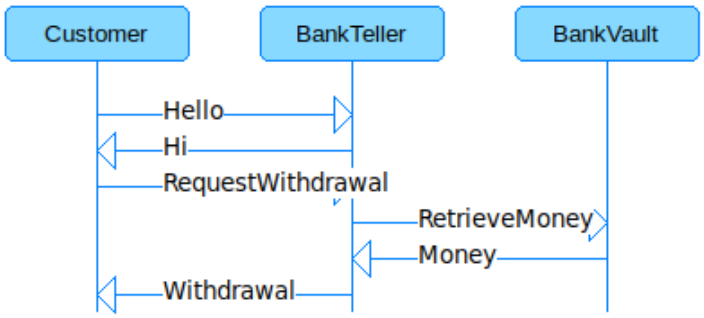


Figura 2.2: Diagrama de Sequência

O resultado desse processo é um arquivo do tipo *System Entity Structure* (SES), que possui uma estrutura bem padronizada. Nas primeiras linhas, ele lista os componentes que constituem o sistema utilizando a palavra chave *is made of*, exemplo:

```
From the BankTellerExamplesys perspective, BankTellerExample is made of  Customer, BankTeller, and
    BankVault!
```

Código 3: Exemplo de primeira linha do arquivo com a extensão .ses

Logo em seguida, são enumerados, em sequência de ocorrência, a troca de eventos entre os componentes do sistema, utilizando as palavras chaves *sends* e *to* exemplo:

```
From the BankTellerExamplesys perspective, Customer sends Hello to BankTeller!
```

Código 4: Troca de eventos do arquivo .ses

Esse modelo bem padronizado facilita o trabalho de extração de informação e tratamento para o desenvolvimento dos casos de teste.

Depois dessa etapa é possível gerar as *models* que são as definições do comportamento de cada entidade, com base nesse arquivo *ses*. Como saída desse processo são gerados os arquivos *dnl* e *java* de cada entidade, que descrevem o comportamento das mesmas, listando detalhadamente a sua execução de cada uma, individualmente, durante a simulação do sistema. É possível alterar a codificação das *models* para conseguir realizar simulações mais completas e complexas, adicionando condições, variáveis e códigos java customizados para os eventos internos e externos.

O arquivo com o tipo *.dnl* é mais complexo que o *.ses*. Ele é focado em descrever em mais detalhes como aquela entidade se comporta durante a simulação. Nesse tipo de arquivo, primeiro é descrito as entradas e saídas daquele componente e, em seguida, os detalhes das transições de estados e trocas de eventos.

Por fim, é possível realizar a simulação de fato. Para essa última etapa, primeiro é necessário unir as informações do arquivo *.ses* com a *models* por meio do botão *"Prune a SES Tree"* e depois executar a simulação de fato. Na execução é demonstrado passo a passo a trocas de eventos entre os sistemas da aplicação, por meio de uma interface gráfica, e nela podemos visualizar as mensagens entrando e saindo dos componentes. Ao final da simulação, são geradas estatísticas com os dados de quantas iterações, transições e o tempo de execução da mesma.

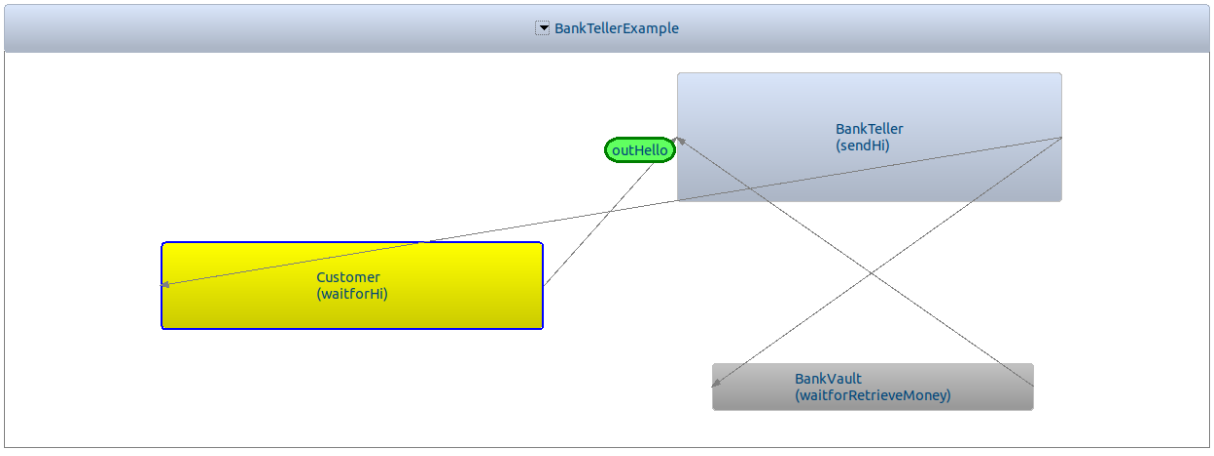


Figura 2.3: Simulação no MS4Me

Embora seja utilizado o *software MS4Me* de base para o projeto, os mesmos conhecimentos adquiridos para as montagem dos casos de testes podem ser reaproveitados para

integração de outros *softwares* de simulação e modelagem. O DEVS2BDD já foi fabricado pensando nessa integração futura. Para esse propósito, é necessário somente criar um novo comando e uma nova estratégia para adaptar-se a sintaxe da nova DSL do novo software de simulação, como mencionado na Seção 3.1.1.

Capítulo 3

DEVS2BDD

Neste capítulo será abordado sobre o produto desenvolvido no projeto, no qual foi nomeado como DEVS2BDD e é explicado em detalhes seu funcionamento e organização.

Figura 3.1: Visão geral do processo de desenvolvimento com a CLI

Na Figura 3.1, ilustra-se o processo da proposta deste trabalho e como ela se encaixa no processo de desenvolvimento de software, conforme foi vislumbrado. Primeiramente é realizada a simulação e modelagem no *MS4ME*. Os resultados da simulação são então

processados pela CLI DEVS2BDD. Essa, por sua vez, retorna os cenários de testes de comportamento no formato de cenários BDD. Dessa forma, a equipe de desenvolvimento de software pode não apenas utilizar esses cenários como base de sua implementação orientada a teses de comportamento como pode validar e verificar cenários importantes de execução advindos da simulação do SoS na ferramenta DEVS.

A CLI foi desenvolvida utilizando a biblioteca *dry-cli* [38] que utiliza uma classe para codificar cada comando. Por meio do DEVS2BDD é possível utilizar três comandos para geração de casos de testes: *ses-generator*, *dnl-generator*, *dnl-bulk-generator* que, respectivamente, possuem a função de gerar cenários de BDD a partir de um arquivo do tipo *System Entity Structure* (SES), gerar a partir de um arquivo do tipo *DEVS Natural Language* (DNL) e o último gera, automaticamente, um arquivo de teste para cada *model (.dnl)* do projeto exportado.

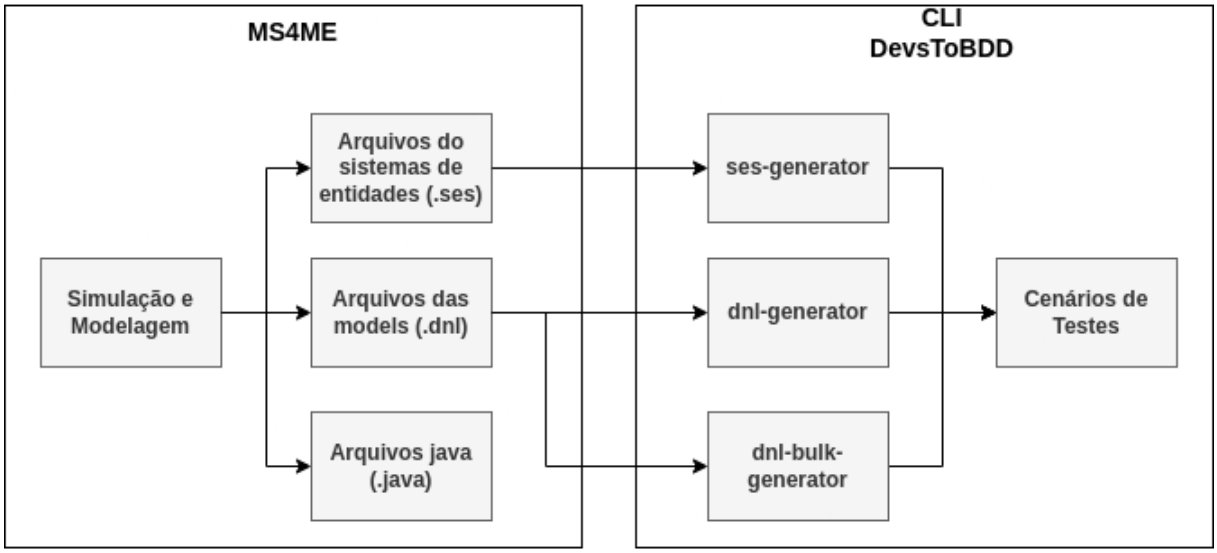


Figura 3.2: Diagrama da relação dos comandos com os arquivos do ms4

No Diagrama 3.2, pode-se visualizar melhor qual entrada cada comando utiliza. Vale observar que a saída de arquivo .java, do *software* de simulação, ainda não foi utilizada neste trabalho para a geração dos casos de testes. Esses arquivos são gerados, pelo *MS4ME*, em conjunto com os arquivos *.dnl* e possuem a mesma informação com diferentes níveis de abstração. No entanto, os arquivos .dnl já provêem os fluxos com os respectivos passos necessários para a geração das *features* BDD. No entanto, reconhece-se que em uma proposta futura, explorar os arquivos *.java* podem enriquecer os passos dos cenários nas *features* BDD.

Na CLI também é possível utilizar três comandos extras para auxiliar o uso da mesma. O comando *version* para verificação da versão da ferramenta, o comando *help* que lista

as opções de comandos da CLI e o comando *steps-generator* que gera as definições dos passos com base em um arquivo de *features*.

```
Commands:
  main.rb dnl-bulk-generator PROJECT_FILE_PATH           # Generate all bdd features from dnl folder inside ms4 project
  main.rb dnl-generator DNL_FILE_PATH [OUTPUT_FILE_NAME] # Generate bdd feature file from ms4 .dnl file
  main.rb ses-generator SES_FILE_PATH [OUTPUT_FILE_NAME] # Generate bdd feature file from ms4 .ses file
  main.rb steps-generator FILE_PATH [OUTPUT_FILE_PATH]   # Generate step definitions from a feature file
  main.rb version                                        # Print version
```

Código 5: Saída do comando *help* da CLI

3.1 Arquitetura da Implementação

A codificação do projeto se encontra no repositório PedroAugustoRamalhoDuarte/devs-to-bdd [39] no github e a CLI foi desenvolvida na linguagem *ruby*. Foi escolhido *ruby* como linguagem de programação para a ferramenta, pela facilidade que a linguagem proporciona e principalmente pela grande integração entre as ferramentas BDD, como o Cucumber.

3.1.1 Modularização

Em relação ao código ele foi modularizado nas seguintes partes:

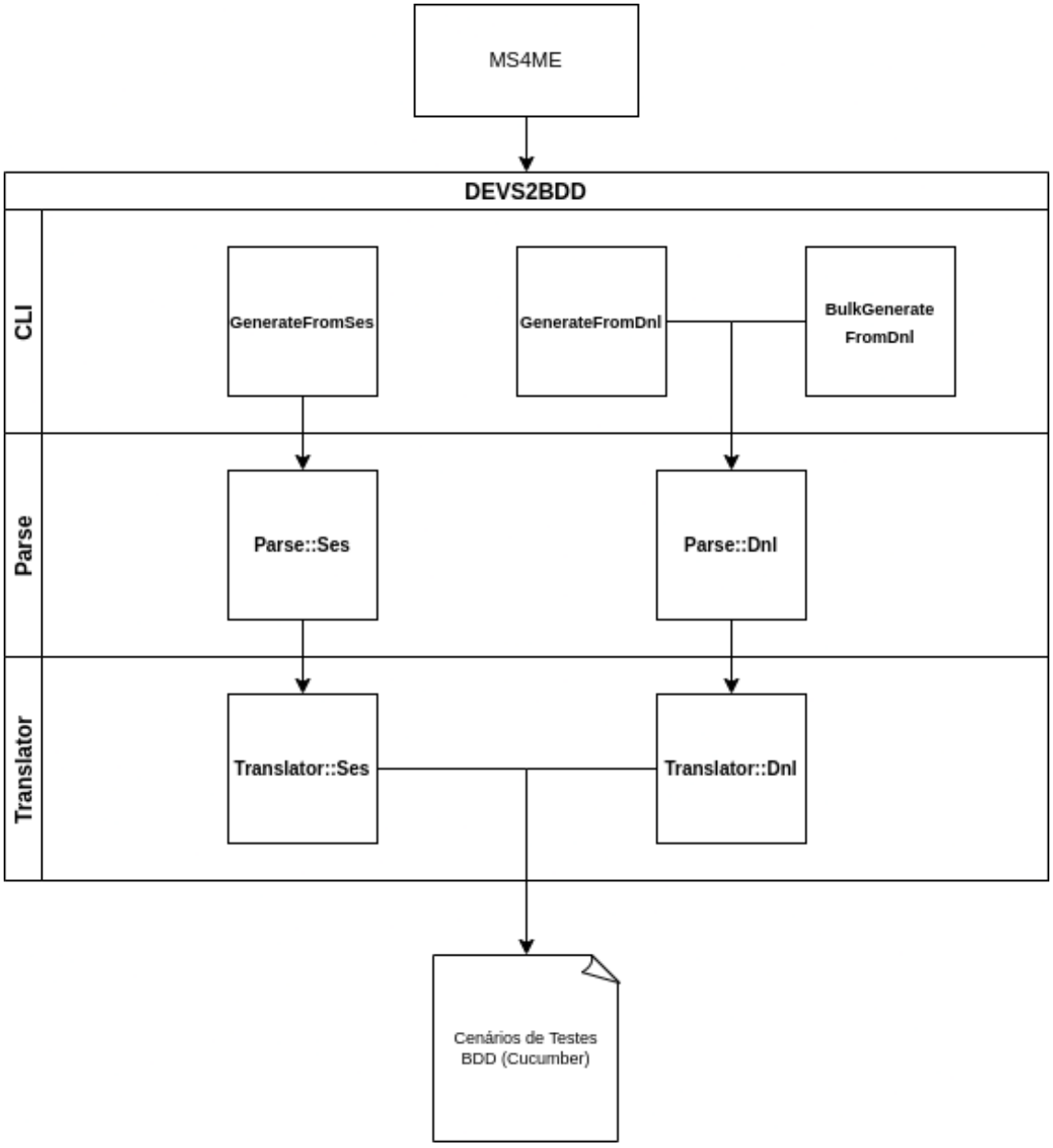


Figura 3.3: Modularização de código

- A CLI presente no arquivo *main.rb* que lida com a interface de terminal.
- O *Parse*: que tem a função de ler os arquivos fornecidos pelo MS4 e colocar em uma estrutura de eventos de casos de teste.
- O *Translator* que é responsável em transformar a estrutura de eventos fornecida pelo *Parse* em arquivos do *Cucumber* com a extensão *.feature*

Para a codificação da geração dos casos de teste de cada tipo de arquivo do MS4Me utiliza-se o *design pattern*, bastante popular, denominado *Strategy*. Esse é um padrão de projeto comportamental que permite a troca da implementação de uma funcionalidade

em tempo de execução, colocando cada estratégia em classes separadas e utilizando um contexto como interface [40].

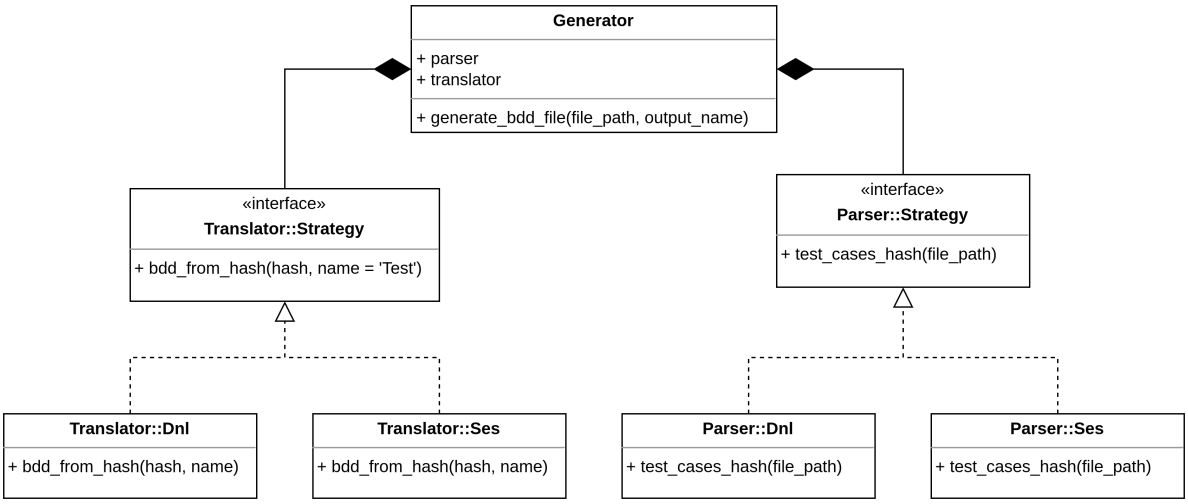


Figura 3.4: Diagrama UML das classes que compõem o Strategy

No projeto há duas estratégias, uma para o *parse* e outra para tradução. Cada uma das estratégias foi implementada duas vezes, uma vez para o tipo de arquivo SES e outra para o DNL, como é possível observar na Figura 3.4:

```
class Parse::Strategy
  # Returns test cases hash from file_path
  # @abstract
  #
  # @return [Hash]
  def test_cases_hash(file_path)
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
end
```

Código 6: Parse::Strategy

```ruby
# Strategy abstract class to translate files
class Translator::Strategy
  # Creates feature files from hash
  # @abstract
  #
  # @return [String] Output path
  def bdd_from_hash(_test_cases_hash, _feature_name = 'Test')
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
end
```

Código 7: Translator::Strategy

Utilizando esse padrão é necessário uma classe contexto, que possui o objetivo de ser a interface entre os clientes, que no caso são os comandos da CLI, e as estratégias de implementação. No projeto o contexto é a classe *Generator*:

```ruby
class Generator
  attr_writer :parser, :translator

  def initialize(parser, translator)
    @parser = parser
    @translator = translator
  end

  # Creates feature files from file path using strategy above
  # @param [String] file_path
  # @param [String] output_name
  #
  # @return [String] Output path
  def generate_bdd_file(file_path, output_name)
    test_case_hash = @parser.test_cases_hash(file_path)
    @translator.bdd_from_hash(test_case_hash, output_name)
  end
end
```

Código 8: Classe Generator

É utilizado esse padrão para organização de código para facilitar o suporte de novas DSL de novas ferramentas de simulação. Já que para tal ação é necessário somente definir um comando da CLI e as estratégias para o novo tipo de arquivo.

3.1.2 Bibliotecas utilizadas

O projeto utiliza somente três bibliotecas do *ruby* para sua execução: a *dry-cli* que já foi mencionada anteriormente, na qual auxilia na confecção da CLI, a gema *zeitwerk* [41] que tem a função de auxiliar no carregamento do código. A última é a gema *cucumber* [42] que auxilia na fabricação das definições dos passos das *feature*.

Em termos de bibliotecas de desenvolvimento é utilizado o *rspec* [43] para os testes, *rubocop* [44] com suas extensões de *performance* e do *rspec* para padronização do código, o *simplecov* [45] para analisar a cobertura de testes da aplicação e o *yard* [46] pra documentação.

3.2 Exemplo Guia: BankTellerExample

Para a explicação do código é empregue o exemplo padrão fornecido pela *MS4 Systems* de um sistema de Caixa Eletrônico nomeado de *BankTellerExample* e que pode ser encontrado dentro da pasta *examples* do projeto [47]. O sistema tem como base 3 entidades principais, na qual cada uma pode ser considerada um sistema, o Cliente *(Customer)*, o Banqueiro *(BankTeller)* e o Caixa eletrônico *(BankVault)*.

3.2.1 Simulação

Para iniciar a simulação o cliente fala "Oi" para o banqueiro que em seguida responde "Olá"; esse é nosso primeiro cenário de comportamento. Logo depois, o cliente solicita um saque de dinheiro para o banqueiro que repassa a informação para o caixa eletrônico. Esse, por sua vez, devolve o dinheiro requisitado para o banqueiro e por fim o banqueiro repassa o dinheiro para o cliente, encerrando a simulação e o segundo caso de teste.

Esse exemplo se mostrou interessante para guiar a explicação da ferramenta DEVS2BDD, porque é fácil perceber os cenários de testes que deveriam ser gerados e o sistema se comporta como um grafo fechado, no qual um subsistema envia uma mensagem e espera uma resposta. Essa segunda característica é fundamental para facilitar a geração dos cenários de testes BDD pela ferramenta, pois esse requisito é utilizado em uma de suas funcionalidades, conforme descrito na Seção 3.2.3.

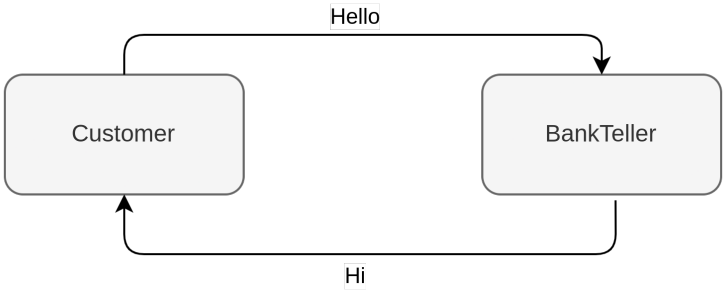


Figura 3.5: Diagrama de envio cíclico de eventos

3.2.2 Cenários esperados

Foram definidos os seguintes cenários de teste de comportamento utilizando a linguagem Gherkin:

```
Feature: BankTellerExample
  Scenario: Initial comunication
    When Customer sends Hello to BankTeller
    Then BankTeller sends Hi to Customer
  Scenario: Request withdrawal
    When Customer sends RequestWithdrawal to BankTeller
    And BankTeller sends RetrieveMoney to BankVault
    And BankVault sends Money to BankTeller
    Then BankTeller sends Withdrawal to Customer
```

Código 9: Casos de testes esperados com os dois cenários

Ou é possível separar um pouco mais os casos do segundo cenário:

```
Feature: BankTellerExample
  Scenario: Initial comunication
    When Customer sends Hello to BankTeller
    Then BankTeller sends Hi to Customer
  Scenario: Request withdrawal
    When Customer sends RequestWithdrawal to BankTeller
    Then BankTeller sends RetrieveMoney to BankVault
  Scenario: BankTeller request money to bank vault
    When BankTeller sends RetrieveMoney to BankVault
    Then BankVault sends Money to BankTeller
  Scenario: Bank vault returns money
    When BankVault sends Money to BankTeller
    Then BankTeller sends Withdrawal to Customer
```

Código 10: Casos de testes esperados com quatro cenários

A seguir, será possível observar que o *software* de geração de casos de teste apresenta a saída mais semelhante ao primeiro exemplo. E agora, será demonstrado o funcionamento do DEVS2BDD, utilizando esse exemplo, nos 2 possíveis geradores de casos de testes.

3.2.3 SeS Generator

O SeS Generator é um gerador que recebe de entrada os arquivos do tipo *System Entity Structure* (SES) do *MS4Me* e tem o objetivo retornar um arquivo *cucumber* com os casos de testes de comportamento de forma mais generalista do sistema de sistemas apresentado.

Esse comando aceita dois parâmetros, o primeiro é o caminho para o arquivo com a extensão *.ses* e o segundo, opcional, é o nome do arquivo *cucumber* de saída.

Tendo em vista a lógica de programação, primeiro é executado um *parse* do arquivo de entrada que transforma as linhas em um lista de conjuntos de eventos. Cada evento contém o remetente, destinatário e o nome do evento enviado. Cada conjunto de eventos representa um cenário de BDD, no qual é formado em um ciclo de envios de mensagens, ou seja, um conjunto de eventos é finalizado quando o primeiro remetente é o último destinatário.

A segunda etapa do processo é feita pela classe *Translator::Ses*, na qual recebe a lista de conjuntos de eventos, da etapa anterior e cria um arquivo *cucumber* com os cenários de testes. Para converter o evento em uma linha de BDD é utilizada a seguinte lógica: Quando é o primeiro evento do conjunto utilizamos a palavra chave *Then*, quando é o último utilizamos *When* e para os demais a palavra chave *And*. Para facilitar o enten-

dimento do fluxo completo do comando SeS Generator, é possível visualizar o diagrama abaixo.

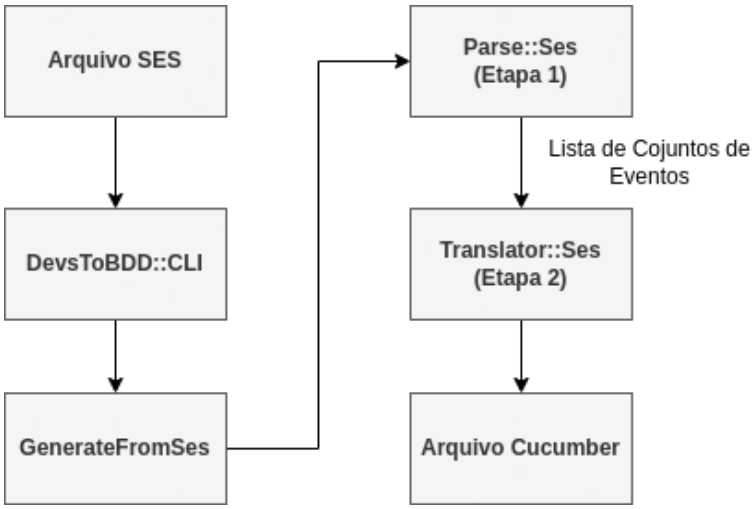


Figura 3.6: Fluxo de responsabilidade no arquivo SES

Para demonstrar o funcionamento desse comando, é possível executar a seguinte linha no terminal na pasta raiz do projeto:

```
ruby ./lib/main.rb ses-generator './examples/BankTellerExample/ses/BankTellerExample.ses' BankTellerExample
```

```
From the BankTellerExamplesys perspective, BankTellerExample is made of Customer,
    BankTeller, and BankVault!
From the BankTellerExamplesys perspective, Customer sends Hello to BankTeller!
From the BankTellerExamplesys perspective, BankTeller sends Hi to Customer!
From the BankTellerExamplesys perspective, Customer sends RequestWithdrawal to
    BankTeller!
From the BankTellerExamplesys perspective, BankTeller sends RetrieveMoney to BankVault
    !
From the BankTellerExamplesys perspective, BankVault sends Money to BankTeller!
From the BankTellerExamplesys perspective, BankTeller sends Withdrawal to Customer!
```

Código 11: Entrada BankTellerExample.ses

```
Feature: BankTellerExample
  Scenario: 0
    When Customer sends Hello to BankTeller
    Then BankTeller sends Hi to Customer
  Scenario: 1
    When Customer sends RequestWithdrawal to BankTeller
    And BankTeller sends RetrieveMoney to BankVault
    And BankVault sends Money to BankTeller
    Then BankTeller sends Withdrawal to Customer
```

Código 12: Saída BankTellerExample.feature

Pode-se observar que os cenários 0 e 1 do Código 12 se mostram bem interessantes para descrever o problema, porém não foram nomeados de forma eficiente, como é possível fazer nos próximos comandos, porquê não temos as informações necessárias nos arquivos do tipo *System Entity Structure* (SES). Outra observação importante, é que foram obtidos os mesmos resultados esperados do Código 9, com exceção do nome do cenário, conseguindo assim, ajudar o desenvolvedor a aplicar o BDD em seu projeto.

3.2.4 Dnl Generator

O *dnl generator* tem uma visão mais específica da *model* que queremos analisar e não da troca de eventos do sistema por completo como era no caso do *ses generator*.

Para executar esse comando é necessário novamente de dois parâmetros, o primeiro deles é o caminho para o arquivo *.dnl* e o segundo é o nome do arquivo de saída, que é opcional.

O fluxo de informação nesse comando é idêntico ao fluxo do SES, Figura 3.6, porém a implementação interna do *parse* e do *translator* é diferente.

Primeiro é realizado o *parse* do arquivo lendo linha por linha e é realizada uma análise de palavras chaves. Sempre é iniciado um novo cenário de teste quando aparecem as palavras *to start* e *hold* na mesma linha ou então *passivate*, já que essas são as palavras chaves para o inicio da comunicação entre os sistemas. Para finalizar o cenário de teste, procura-se o início do próximo cenário ou quando a leitura do arquivo é finalizada, assim são obtidos fluxos satisfatório de testes.

Depois dessa primeira etapa de *parse* do arquivo DNL e a separação das linhas em conjuntos (cenários de testes), é necessário passar o resultado dessa primeira etapa pelo *translator*. O *Translator:Dnl* é o responsável pela transformação das linhas em código *.dnl*

para o formato de passos do *cucumber*, para isso é utilizada a seguinte regra de tradução descrita na linguagem *ruby*, conforme segue:

```ruby
# Convert a dnl line to a cucumber instruction
#
# @return [String]
def convert_dnl_line(line)
  if line.include? 'output'
    "Then sends #{line.split[-1]}"
  elsif line.include? 'from'
    "And go to #{line.split[-1]}"
  elsif line.include?('to start') && line.include?('hold')
    "When holding #{line.split(',')[1][6..]}"
  elsif line.include? 'hold'
    "And #{line}"
  elsif line.include? 'passivate'
    # pass
  elsif !line.empty?
    line[0].upcase + line[1..]
  end
end
```

Código 13: Processo de tradução do dnl para cucumber em ruby

Para a fabricação dos cenários BDD são empregues as seguintes palavras chaves do *Gherkin* : *Feature, Scenario, When, Then e And.* E a tradução é feita da seguinte maneira para cada uma delas:

- *Feature*: É utilizada na primeira linha do arquivo *cucumber*, em conjunto com nome da *model* ou a palavra passada de parâmetro pela CLI.
- *Scenario*: É utilizada para nomear o cenário e contém o nome do primeiro evento que inicia a troca de eventos.
- *When*: É utilizado no aparecimento das palavras chaves *to start* e *hold* na mesma linha.
- *Then*: É utilizado quando a linha possui a palavra chave *output*.
- *And*: É utilizado na sequência de expressões *When* e *Then*. Depois do *Then*, quando a linha possui a palavra chave *from* e para o *When* quando a linha possui a palavra chave *hold*, mas não possui *to start*.

Para exemplificar o uso o gerador, é utilizado novamente o sistema do *BankTellerExample*. Passando de parâmetro o caminho do arquivo da *model* do *BankTeller* são obtidos os seguintes resultados;

```
accepts input on Money !
generates output on Withdrawal !

to start,passivate in waitforHello !
when in waitforHello and receive Hello go to sendHi!
hold in sendHi for time 1!
after sendHi output Hi!
from sendHi go to waitforRequestWithdrawal!
passivate in waitforRequestWithdrawal !
when in waitforRequestWithdrawal and receive RequestWithdrawal go to sendRetrieveMoney
    !
hold in sendRetrieveMoney for time 1!
after sendRetrieveMoney output RetrieveMoney!
from sendRetrieveMoney go to waitforMoney!
passivate in waitforMoney !
when in waitforMoney and receive Money go to sendWithdrawal!
hold in sendWithdrawal for time 1!
after sendWithdrawal output Withdrawal!
from sendWithdrawal go to passive!
passivate in passive!
```

Código 14: Entrada BankTeller.dnl

```
ruby ./lib/main.rb dnl-generator './examples/BankTellerExample/dnl/BankTeller.dnl'
```

Código 15: Comando bash utilizando a CLI para o dnl-generator

```
Feature: BankTeller
  Scenario: waitforHello
    When in waitforHello and receive Hello go to sendHi
    And hold in sendHi for time 1
    Then sends Hi
    And go to waitforRequestWithdrawal
  Scenario: waitforRequestWithdrawal
    When in waitforRequestWithdrawal and receive RequestWithdrawal go to sendRetrieve
    And hold in sendRetrieveMoney for time 1
    Then sends RetrieveMoney
    And go to waitforMoney
  Scenario: waitforMoney
    When in waitforMoney and receive Money go to sendWithdrawal
    And hold in sendWithdrawal for time 1
    Then sends Withdrawal
    And go to passive
```

Código 16: Saída BankTeller.dnl

Pode-se observar que foi possível obter fluxos interessantes para os cenários de teste, porém a leitura não fica tão agradável como é de esperar de cenários de testes BDD, já que o arquivo analisado é bem descritivo e possui frases que fazem mais sentido no ambiente de simulação do que no ambiente de implementação de fato. No caso desse gerador, como temos mais informações, conseguimos obter os nomes dos cenários por meio dos nomes dos eventos de espera.

Foi utilizada a *model* do *BankTeller*, mas seria possível a mesma análise com os arquivos do *Customer* ou *BankVault* sem perda de generalidade.

Ao longo do arquivo do tipo DNL, podem ocorrer aparições de códigos java como citados no Capítulo 2.5, esses trechos são excluídos do *parse*, pois não ajudam na confecção de casos de testes.

3.2.5 Dnl Bulk Generator

Esse comando tem a mesma função do *dnl generator*, porém automaticamente, por meio da pasta do projeto *MS4 Me*, localiza todos os arquivos *.dnl* e executa a lógica de negócio do *dnl generator*, gerando um arquivo de saída *cucumber* para cada um dos arquivos *.dnl*. Exemplo de utilização:

```
ruby ./lib/main.rb dnl-bulk-generator './examples/BankTellerExample'
```

Código 17: Comando bash com o exemplo da utilização do dnl-bulk-generator

Ao utilizar esse comando no caso do exemplo dado são gerados 3 arquivos: *BankTeller.feature*, *Customer.feature* e *BankVault.feature*, com casos de testes mais individualizados para cada sub-sistema simulado.

3.2.6 Step Definition Generator

Além dos comandos acima para geração dos casos de testes, o DEVS2BDD também fornece um comando para geração de definições de passos, em *ruby*, para a um determinado arquivo *feature*. Esse comando possui dois argumentos, o primeiro é o caminho para o arquivo *.feature* e o segundo é o caminho do arquivo de saída.

Para codificação desse comando é utilizada a gema *cucumber* desenvolvida pela sua própria organização. Essa gema implementa uma CLI para a execução de cenários de testes no formato Gherkin.

Quando é fornecido para essa CLI um arquivo *.feature* sem a implementação dos seus passos, ela lança um error no terminal informando a falta desses passos e escreve o trecho de código para implementação dos mesmos.

Aproveitando dessa funcionalidade da CLI é utilizado o comando *cucumber* com os argumento *–publish-quiet* (para não exibir no terminal as informações de publicação dos casos de testes) e *-d* (afim de não executar os casos de testes):

```
cucumber input --publish-quiet -d
```

Código 18: Comando bash usado para geração das definições dos passos

Depois de utilizar esse comando é extraída a implementação dos passos utilizando um *regex*. Em seguida, os passos extraidos, são exportados em um arquivo *ruby*. Para demonstração doo funcionamento do comando *steps-generator* do DEVS2BDD, será utilizado o Código 12.

```
ruby ./lib/main.rb steps-generator ./output/BankTellerExample.feature
```

Código 19: Utilização do comando steps-generator

```
When('Customer sends Hello to BankTeller') do
  pending # Write code here that turns the phrase above into concrete actions
end

Then('BankTeller sends Hi to Customer') do
  pending # Write code here that turns the phrase above into concrete actions
end

When('Customer sends RequestWithdrawal to BankTeller') do
  pending # Write code here that turns the phrase above into concrete actions
end

When('BankTeller sends RetrieveMoney to BankVault') do
  pending # Write code here that turns the phrase above into concrete actions
end

When('BankVault sends Money to BankTeller') do
  pending # Write code here that turns the phrase above into concrete actions
end

Then('BankTeller sends Withdrawal to Customer') do
  pending # Write code here that turns the phrase above into concrete actions
end
```

Código 20: Definições de passos para o exemplo do BankTeller

É possível executar esse comando após todos os outros comandos de geração de casos de testes automaticamente adicionando o argumento `--steps=true`. Sendo assim é possível gerar os cenários BDDs em conjunto com suas definições.

No próximo capítulo, é realizada uma prova de conceito da nossa metodologia no protótipo da Body Sensor Network [48] desenvolvido pelo grupo LADECIC (Laboratório de Dependabilidade do Departamento de Ciência da Computação).

Capítulo 4

Estudo de Caso BSN

4.1 Introdução

Para validação da ferramenta desenvolvida no trabalho, foi escolhido realizar os testes de comportamento com base no protótipo da *Body Sensor Network* (BSN) [48]. A BSN é protótipo de um sistema que tem como objetivo monitorar e analisar constantemente a saúde de um paciente. A BSN acomoda diversos sensores para captura dos sinais vitais do paciente e envia para uma central de processamento, afim de ser analisado. O sistema foi feito para ser configurável e auto adaptativo o que aumenta sua utilidade e resiliência.

Para o estudo de caso foi separado um subsistema da BSN, já que, dada a complexidade, o espectro geral do sistema de sistema abordado seria de difícil análise. Esse subsistema foca em analisar a parte da comunicação do *Central Hub* e do sensor de temperatura, um subsistema que ainda pode ser considerado um sistema de sistemas, e com base nesse espectro, foi desenvolvida uma modelagem no software *MS4ME*.

Para esse subsistema, existem os seguintes 4 sistemas com seus respectivos objetivos:

- O *CentralHub* com o objetivo de ser o centralizador do protótipo, analisando os dados previventes do sensor e comunicando com o serviço de alerta caso necessário
- O *Collector* que tem a função de servir como ponte entre o *CentralHub* e o Sensor
- O Sensor responsável por aferir os sinais vitais do usuário, que no nosso caso é a temperatura do paciente
- E o *AlertSystem* que tem o objetivo de alertar os serviços de emergência caso aconteça alguma anormalidade

Na primeira parte da simulação o *CentralHub* solicita para o *Collector* as informações de temperatura, e logo em seguida o *Collector* solicita para o sensor de temperatura seus

dados que envia para o *CentralHub*, primeiramente resultados saudáveis de temperatura, sendo assim a *CentralHub* não precisa se comunicar com o serviço de Alerta. No segundo momento, é realizada a mesma troca de mensagens iniciais, porém é enviado um sinal de temperatura não saudável e o *CentralHub* envia uma mensagem ao serviço de alerta. Então esperamos que os casos de testes gerados narrem essas duas etapas da simulação.

O *Collector* na simulação não possui grande importância já que ele só repassa a informação das entidades que são conectadas por ele, porém foi decidido mantê-lo por sua existência na codificação da BSN já implementada.

Para validação da ferramenta, será seguido o fluxo da Figura 3.1;

4.2 Simulação

O primeiro passo para validar a ferramenta DEVS2BDD é realizar a simulação do nosso subsistema da BSN. A simulação foi realizada tendo como base as especificações da BSN.

Como foi explicado na seção de introdução do *MS4ME*, Seção 2.5, a primeira etapa do processo de simulação e modelagem é descrever a troca de eventos realizadas entre os sistemas, via interface gráfica, na parte de diagrama de sequência. Baseado no fluxo de eventos da BSN foi desenvolvido o seguinte diagrama:

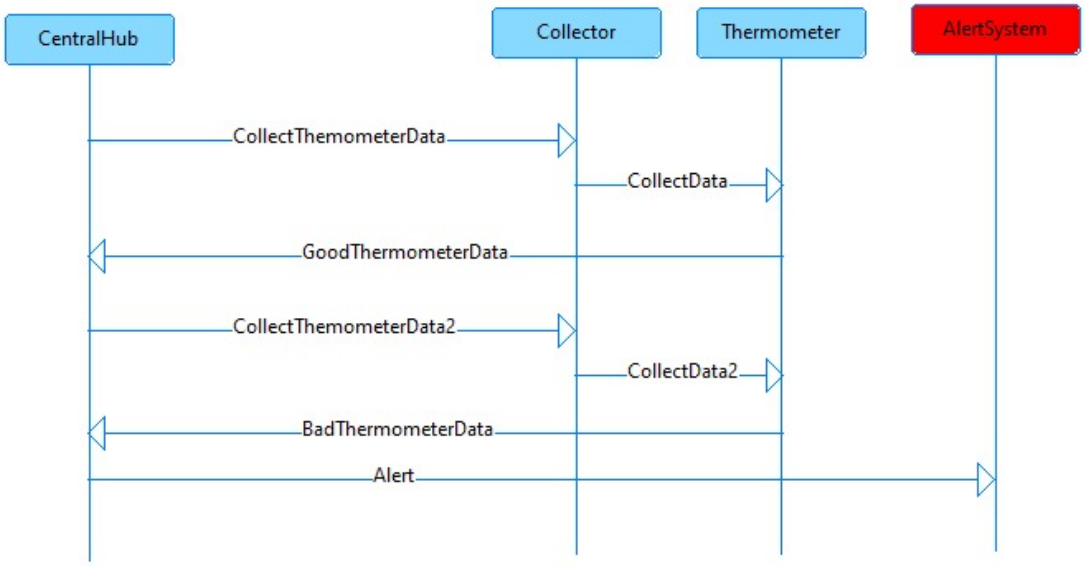


Figura 4.1: Diagrama de sequência da BSN

Depois de finalizado o diagrama de sequência, é possível gerar, a partir dele, o seguinte arquivo do tipo SES:

```
From the BSNsys perspective, BSN is made of CentralHub, Collector, Thermometer, and
    AlertSystem!
From the BSNsys perspective, CentralHub sends CollectThermometerData to Collector!
From the BSNsys perspective, Collector sends CollectData to Thermometer!
From the BSNsys perspective, Thermometer sends GoodThermometerData to CentralHub!
From the BSNsys perspective, CentralHub sends CollectThermometerData2 to Collector!
From the BSNsys perspective, Collector sends CollectData2 to Thermometer!
From the BSNsys perspective, Thermometer sends BadThermometerData to CentralHub!
From the BSNsys perspective, CentralHub sends Alert to AlertSystem!
```

Código 21: BSN.ses

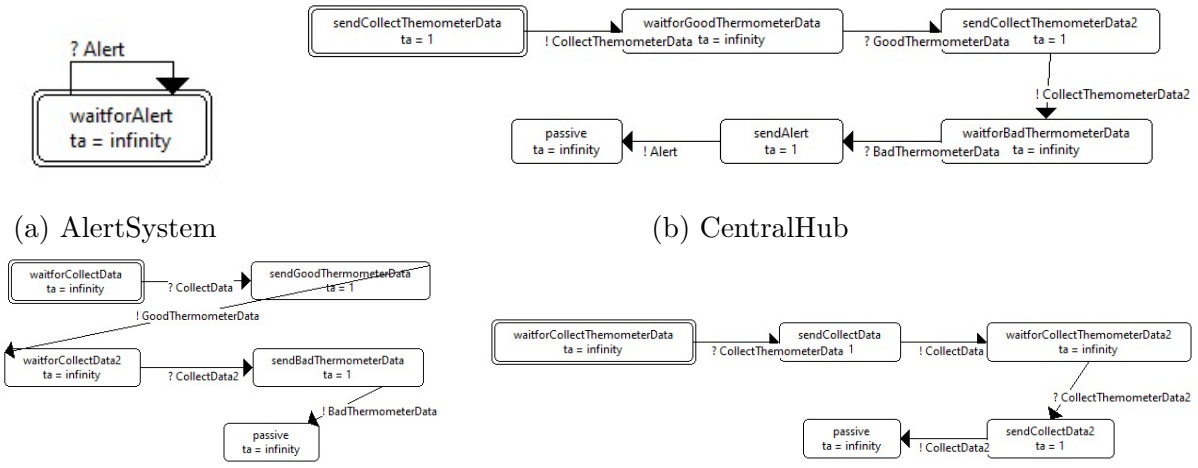
Novamente com base no diagrama de sequência é possível gerar automaticamente os arquivos referentes as *models* do sistema. Para cada entidade foram gerados 2 arquivos, um arquivo do tipo DNL e um arquivo do tipo java. O arquivo da *model AlertSystem* é utilizado, a seguir, para exemplificação do formato do arquivo DNL, sem perda de generalidade.

```
accepts input on Alert !

to start, passivate in waitforAlert !
when in waitforAlert and receive Alert go to waitforAlert!
external event for waitforAlert with Alert
<%//Add your own code
Serializable variable = messageList.get(0).getData();
%>!
```

Código 22: Arquivo AlertSystem.dnl

Por meio do MS4Me, também é possível gerar um grafo com o diagrama de estados de cada entidade, conforme apresentado na Figura 4.2.



(a) AlertSystem (b) CentralHub

(c) Thermometer (d) Collector

Figura 4.2: Diagrama de estados de todas as entidades

Por fim, podemos juntar as informações dos arquivos DNL e SES e executar a simulação via interface gráfica:

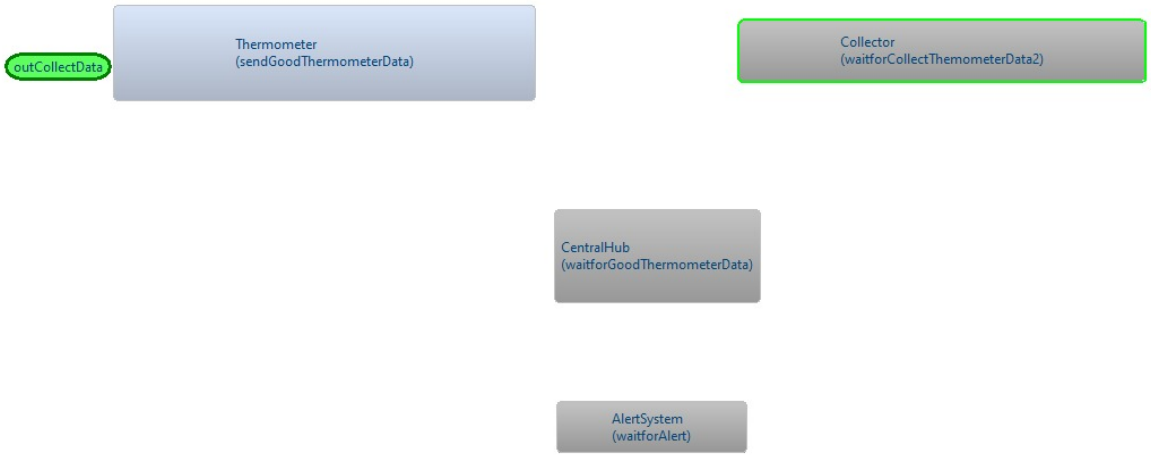


Figura 4.3: Simulação da BSN

4.3 Resultado esperado

Antes de vislumbrar os resultados obtidos pelos DEVS2BDD, é necessário abstrair os resultados esperados dos cenários de testes BDD, tendo como base os arquivos gerados na seção anterior.

Tendo como base o Código 21, é possível extrair dois cenários de testes interessantes. O primeiro deles é um caso em que o paciente está com uma temperatura dentro da normalidade, portanto não se deve alertar os serviços de emergência. No segundo caso,

o paciente está com temperatura fora da normalidade então é necessário comunicar o sistema de alerta.

Desse modo, quando o arquivo de entrada *(BSN.ses)*, Código 21, for executado com o comando do *ses-generator* 3.2.3, são esperados cenários de testes semelhantes a:

```
Feature: BSN
  # Cenário no qual o paciente está saudável
  Scenario: Paciente is healthy
    Given Pacient is healthy
    When CentralHub sends CollectThermometerData to Collector
    And Collector sends CollectData to Thermometer
    Then Thermometer sends GoodThermometerData to CentralHub
  # Cenário no qual o paciente não está saudável
  Scenario: Pacient is not healthy
    Given Pacient is not healthy
    When CentralHub sends CollectThermometerData2 to Collector
    And Collector sends CollectData2 to Thermometer
    And Thermometer sends BadThermometerData to CentralHub
    Then CentralHub sends Alert to AlertSystem
```

Código 23: Cenários esperados de testes do subsistema da BSN

4.4 Geração de Casos de Testes

4.4.1 SES Generator

Depois de criada a simulação, foi utilizado o software criado no trabalho para gerar os casos de testes. Ao executar o comando *ses-generator* com o arquivo (BSN.ses), Código 21, de entrada, foi obtido o seguinte resultado:

```
Feature: BSN
  Scenario: 0
    When CentralHub sends CollectThermometerData to Collector
    And Collector sends CollectData to Thermometer
    Then Thermometer sends GoodThermometerData to CentralHub
  Scenario: 1
    When CentralHub sends CollectThermometerData2 to Collector
    And Collector sends CollectData2 to Thermometer
    And Thermometer sends BadThermometerData to CentralHub
    Then CentralHub sends Alert to AlertSystem
```

Código 24: Saída BSN.feature

Como era de se esperar, o *ses-generator* não conseguiu detectar o nome dos cenários. Outro problema é que a propriedade *Given*, importante para o cenário de teste BDD ficar mais legível, não foi detectada, justamente porque não há nenhum indício na existência da mesma no arquivo SES. Mesmo com esses dois problemas, o resultado gerado é satisfatório e auxilia na implementação do BDD e poupa tempo do desenvolvedor.

4.4.2 DNL Generator

Paralelamente ao gerador de cima, o DEVS2BDD também fornece a opção de gerar cenários baseados nos arquivos do tipo DNL. Para fins de facilidade, foi utilizado o comando *dnl-bulk-generator*, com o parâmetro da pasta do projeto da BSN exportado e o resultado, com base nos 4 arquivos DNL da simulação, foram os seguintes arquivos cucumber.

```
Feature: CentralHub
  Scenario: hold in sendCollectThermometerData
    When holding in sendCollectThermometerData for time 1
    Then sends CollectThermometerData
    And go to waitforGoodThermometerData
  Scenario: waitforGoodThermometerData
    When in waitforGoodThermometerData and receive GoodThermometerData go to sendCollectThermometerData2
    And hold in sendCollectThermometerData2 for time 1
    Then sends CollectThermometerData2
    And go to waitforBadThermometerData
  Scenario: waitforBadThermometerData
    When in waitforBadThermometerData and receive BadThermometerData go to sendAlert
    And hold in sendAlert for time 1
    Then sends Alert
    And go to passive
```

Código 25: CentralHub.feature

O CentralHub é o agente ativo da simulação ele que inicia o processo de troca de eventos do sistema. Pode-se observar 3 cenários de teste: O primeiro no qual ele inicia requisitando a informação de temperatura para o Collector e é esperado que ele espere por essa resposta do sensor de temperatura. O Segundo que ele inicia esperando o sinal de temperatura, analisa que está dentro da normalidade e segue seu funcionamento. Por fim, ele recebe uma temperatura anormal e envia uma mensagem de alerta.

```
Feature: Collector
  Scenario: waitforCollectThermometerData
    When in waitforCollectThermometerData and receive CollectThermometerData go to sendCollectData
    And hold in sendCollectData for time 1
    Then sends CollectData
    And go to waitforCollectThermometerData2
  Scenario: waitforCollectThermometerData2
    When in waitforCollectThermometerData2 and receive CollectThermometerData2 go to sendCollectData2
    And hold in sendCollectData2 for time 1
    Then sends CollectData2
    And go to passive
```

Código 26: Collector.feature

```
Feature: Thermometer
  Scenario: waitforCollectData
    When in waitforCollectData and receive CollectData go to sendGoodThermometerData
    And hold in sendGoodThermometerData for time 1
    Then sends GoodThermometerData
    And go to waitforCollectData2
  Scenario: waitforCollectData2
    When in waitforCollectData2 and receive CollectData2 go to sendBadThermometerData
    And hold in sendBadThermometerData for time 1
    Then sends BadThermometerData
    And go to passive
```

Código 27: Thermometer.feature

```
Feature: AlertSystem
  Scenario: waitforAlert
    When in waitforAlert and receive Alert go to waitforAlert
```

Código 28: AlertSystem.feature

Os cenários de testes provenientes aos arquivos DNL se mostraram ser úteis para testar cada entidade individualmente. Vale notar, que no exemplo de BSN não foi possível evidenciar a detecção de propriedades emergentes por parte do MS4. No entanto, isso não invalida a proposta deste trabalho, uma vez que a riqueza da análise em si está a cargo do grau de detalhes dos artefatos de entrada do MS4.

Capítulo 5

Conclusão

O presente trabalho abordou o conceito de simulação e modelagem com foco no problema da desconexão entre a etapa de *Modeling and Simulation* (M&S) e a etapa de implementação. Inicialmente, foi demostrada a importância da etapa de M&S para o planejamento e design de SoSs e a importância da etapa de testes para a fabricação de SoSs.

Para solucionar esse problema foi proposta a criação de uma ferramenta que possibilita a geração de cenários de testes BDD, tendo como base os artefatos gerados por softwares de simulação e modelagem.

Diante disso foi desenvolvido o software DEVS2BDD, na linguagem *ruby* [12], que faz justamente o que foi proposto para o projeto. Recebe os diversos tipos de saídas do software do MS4Me como entrada e devolve para o desenvolvedor casos de testes BDD.

Os cenários de testes gerados, a partir dos arquivos SES, se mostraram satisfatórios para descrever a missão do SoS. Contudo, o conteúdo dos casos de testes BDD gerados a partir dos arquivos DNL ficaram prolixos e com uma semântica de ambiente de simulação. Portanto, não foi possível obter bons cenários de testes para os componentes da aplicação;

5.1 Trabalhos Futuros

Vislumbra como futuras melhorias o aumento do espectro de cobertura de softwares de simulação que são suportados pela ferramenta, alterando a estratégia de fabricação de casos de testes BDD para cada uma delas. Posteriormente, o DEVS2BDD também pode englobar outras funcionalidades interessantes para facilitar e melhorar a geração de casos de testes, sendo elas:

- **Interface gráfica**: A opção de interface pela CLI é voltada para programadores. Porém, como as ferramentas de simulação e modelagem não são somente para desenvolvedores, poderia ser desenvolvida uma versão da ferramenta com uma interface gráfica para auxiliar na sua democratização.

- **Parser de comentários nas DSLs de simulações**: Uma funcionalidade que seria interessante para agregar na legibilidade dos testes é fornecer a opção, de por meio de comentários nas DSLs dos software de simulação, enriquecer o cenários BDD com propriedades do contexto em que aqueles eventos estão sendo trocados.
- ***Plugins* para os *softwares* de simulação e modelagem**: Alguns *softwares* de simulação e modelagem aceitam *plugins*. Sendo assim, o DEVS2BDD poderia ser integrado diretamente no ambiente de simulação.
- Enriquecer os passos dos cenários a partir das funcionalidades implementadas nas classes .java.
- Realizar estudos de caso mais complexos para evidenciar que as propriedades emergentes podem ser expressas nos cenários BDD gerados.
- Aplicar a ferramenta em um ambiente de desenvolvimento fim-a-fim de modo que seja possível validar a integração dos cenários BDDs gerados com a equipe de desenvolvimento do Sistema de Sistema.

Além das melhorias citadas para a ferramenta DEVS2BDD, outra opção que se demonstra interessante para trabalhos futuros é inverter o fluxo de criação. Essa opção se daria pela criação da simulação com base nos cenários de BDDs, ao invés de criar cenários BDD a partir da simulação e modelagem, tal como é apresentado no trabalho. Essa alternativa se torna atraente porque os BDDs podem ser utilizados como especificação do sistema na etapa de coleta de requisitos, seguindo, assim, um fluxo melhor no desenvolvimento do *software*.

Referências

[1] Xiao, Boping, Yalan Wang, Lin Ma e Zhisheng Cao: *Reasearch on the history and perspective of system of systems*. Em *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, páginas 1262–1266, Guiyang, China, junho 2011. IEEE, ISBN 978-1-61284-667-5. http://ieeexplore.ieee.org/document/5979463/, acesso em 2022-08-30. 1

[2] Inocêncio, Thiago J., Gustavo R. Gonzales e Flávio E. A. Horita: *PASoS: Processo para definição da arquitetura de Sistemas-de-Sistemas*. Em *Anais do Brazilian Workshop on Large-scale Critical Systems (BWare)*, páginas 25–28. SBC, setembro 2019. https://sol.sbc.org.br/index.php/bware/article/view/7505, acesso em 2022-08-30, tex.copyright: Copyright (c). 1

[3] Maier, Mark W.: *The Role of Modeling and Simulation in System of Systems Development*. Em Rainey, Larry B. e Andreas Tolk (editores): *Modeling and Simulation Support for System of Systems Engineering Applications*, páginas 11–41. Wiley, 1ª edição, dezembro 2014, ISBN 978-1-118-46031-3 978-1-118-50175-7. https://onlinelibrary.wiley.com/doi/10.1002/9781118501757.ch2, acesso em 2022-08-29. 1

[4] *Importance of Software Testing in Software Development Life Cycle - ProQuest*. https://www.proquest.com/openview/a938358c4025fb1fc1551f31dbf81eaa/1?pq-origsite=gscholar&cbl=55228, acesso em 2022-08-30. 1

[5] Neto, Arilo e Dias Neto: *Introdução a Teste de Software*. agosto 2022. 1

[6] Pan, Jiantao: *Software testing*. Dependable Embedded Systems, 5:2006, 1999. 1

[7] *System Verification - SEBoK*. https://www.sebokwiki.org/wiki/System_Verification, acesso em 2022-09-07. 2

[8] Kellner, Marc I, Raymond J Madachy e David M Raffo: *Software process simulation modeling: Why? What? How?* Journal of Systems and Software, 46(2):91–105, 1999, ISSN 0164-1212. https://www.sciencedirect.com/science/article/pii/S0164121299000035. 2

[9] Systems, MS4: *MS4Me*. http://www.ms4systems.com/pages/ms4me.php, acesso em 2022-03-30. 2, 3, 10

[10] North, Dan e others: *Introducing bdd*. Better Software, 12, 2006. 2, 9

[11] Beck, Kent: *Test-driven development: by example*. Addison-Wesley Professional, 2003. 2

[12] *Ruby: A PROGRAMMER'S BEST FRIEND*. `https://www.ruby-lang.org/en/`, acesso em 2022-09-26. 4, 37

[13] Sadagic, Amela e Floy Yates: *Large Scale Adoption of Training Simulations: Are We There Yet?* dezembro 2015. 5

[14] Kuhl, M. E., N. M. Steiger, F. B. Armstrong, J. A. Joines e John S. Carson Ii: *Proceedings of the 2005 Winter Simulation Conference*. 5

[15] Banks, Jerry: *Introduction to simulation*. Em *2000 winter simulation conference proceedings (cat. No. 00CH37165)*, volume 1, páginas 9–16, 2000. tex.organization: IEEE. 5

[16] Grikštaitė, Jūratė: *Business process modelling and simulation: advantages and disadvantages*. Global Academic Society Journal: Social Science Insight, 1(3):4–14, 2008. 5

[17] Taylor, Simon JE, Azam Khan, Katherine L Morse, Andreas Tolk, Levent Yilmaz, Justyna Zander e Pieter J Mosterman: *Grand challenges for modeling and simulation: simulation everywhere—from cyberinfrastructure to clouds to citizens*. simulation, 91(7):648–665, 2015. Publisher: SAGE Publications Sage UK: London, England. 5

[18] Franceschini, Romain, Paul Antoine Bisgambiglia, Luc Touraille, Paul Bisgambiglia e David Hill: *A survey of modelling and simulation software frameworks using Discrete Event System Specification*. Em *2014 imperial college computing student workshop*, 2014. tex.organization: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 6

[19] Zeigler, Bernard P e Hessam S Sarjoughian: *DEVS integrated development environments*. Em *Guide to modeling and simulation of systems of systems*, páginas 11–26. Springer, 2013. 6

[20] Zeigler, Bernard P: *Multifacetted modelling and discrete event simulation*. Academic Press Professional, Inc., 1984. 6

[21] Vangheluwe, Hans: *The discrete event system specification (DEVS) Formalism*. tech. rep, 2001. 6

[22] Moreno, Alejandro, José L Risco-Martín, Eva Besada, Saurabh Mittal e Joaquín Aranda: *DEVS/SOA: Towards DEVS interoperability in distributed M&S*. Em *2009 13th IEEE/ACM international symposium on distributed simulation and real time applications*, páginas 144–153, 2009. tex.organization: IEEE. 6

[23] Bouanan, Youssef, Gregory Zacharewicz e Bruno Vallespir: *DEVS modelling and simulation of human social interaction and influence*. Engineering Applications of Artificial Intelligence, 50:83–92, 2016. Publisher: Elsevier. 7

[24] BRAZ, RENATO CORREA: *A CONSTRUÇÃO DE UM SIMULADOR DE PROCESSOS CONCORRENTES UTILIZANDO DEVS COMO ESTRUTURA FORMAL DE MODELAGEM*. 7

[25] Jamshidi, MO: *System of systems engineering-new challenges for the 21st century*. IEEE Aerospace and Electronic Systems Magazine, 23(5):4–19, 2008. Publisher: IEEE. 7, 8

[26] Mahmoud, Magdi S, Mohamed Saif Ur Rahman e Fouad M AL-Sunni: *Review of microgrid architectures–a system of systems perspective*. IET Renewable Power Generation, 9(8):1064–1078, 2015. Publisher: Wiley Online Library. 7

[27] James Lewis e Martin Fowler: *Microservices*. `https://martinfowler.com/articles/microservices.html`, acesso em 2022-09-13. 8

[28] Cuesta, Carlos E, Elena Navarro e Uwe Zdun: *Synergies of system-of-systems and microservices architectures*. Em *Proceedings of the international colloquium on software-intensive systems-of-systems at 10th european conference on software architecture*, páginas 1–7, 2016. 8

[29] Maier, Mark W: *Architecting principles for systems-of-systems*. Systems Engineering: The Journal of the International Council on Systems Engineering, 1(4):267–284, 1998. Publisher: Wiley Online Library. 8

[30] Dahmann, Judith: *1.4. 3 system of systems pain points*. Em *INCOSE international symposium*, volume 24, páginas 108–121, 2014. Number: 1 tex.organization: Wiley Online Library. 8

[31] Oliveira Neves, Vânia de, Antonia Bertolino, Gugliemo De Angelis e Lina Garcés: *Do we need new strategies for testing systems-of-systems?* Em *Proceedings of the 6th international workshop on software engineering for systems-of-systems*, páginas 29–32, 2018. 8, 9

[32] *Gherkin Syntax - Cucumber Documentation*. `https://cucumber.io/docs/gherkin/`, acesso em 2022-09-13. 9

[33] *Cucumber*. `https://cucumber.io/`, acesso em 2022-09-24. 9

[34] Arnold, Ken, James Gosling e David Holmes: *The java programming language*. Addison Wesley Professional, 2005. 10

[35] Guindon, Christopher: *Eclipse IDE | The Eclipse Foundation*. `https://eclipseide.org/`, acesso em 2022-09-13. 10

[36] Cowles, Mary Kathryn e Bradley P Carlin: *Markov chain Monte Carlo convergence diagnostics: a comparative review*. Journal of the American Statistical Association, 91(434):883–904, 1996. Publisher: Taylor & Francis. 10

[37] Zeigler, Bernard P.: *1 - How Can Modeling and Simulation Help Engineering of System of Systems?* Em Traoré, Mamadou Kaba (editor): *Computational Frameworks*, páginas 1–46. Elsevier, janeiro 2017, ISBN 978-1-78548-256-4. https://www.sciencedirect.com/science/article/pii/B9781785482564500016, acesso em 2022-08-30. 10

[38] dry-rb: *dry-cli: General purpose Command Line Interface (CLI) framework for Ruby.* https://github.com/dry-rb/dry-cli, acesso em 2022-09-24. 15

[39] Duarte, Pedro Augusto Ramalho: *Devs to BDD*, setembro 2022. https://github.com/PedroAugustoRamalhoDuarte/devs-to-bdd, acesso em 2022-09-12, original-date: 2022-02-04T13:11:28Z. 16

[40] *Strategy.* https://refactoring.guru/pt-br/design-patterns/strategy, acesso em 2022-09-10. 18

[41] fxn: *zeitwerk: Efficient and thread-safe code loader for Ruby Interface (CLI) framework for Ruby.* https://github.com/fxn/zeitwerk, acesso em 2022-09-26. 20

[42] *Cucumber*, setembro 2022. https://github.com/cucumber/cucumber-ruby, acesso em 2022-09-18, original-date: 2008-04-17T18:19:13Z. 20

[43] *rspec: Behaviour Driven Development for Ruby. Making TDD Productive and Fun.* http://rspec.info/, acesso em 2022-09-26. 20

[44] *rubocop: A Ruby static code analyzer and formatter, based on the community Ruby style guide.* https://github.com/rubocop/rubocop, acesso em 2022-09-26. 20

[45] *simplecov: Code coverage for Ruby with a powerful configuration library and automatic merging of coverage across test suites.* https://github.com/simplecov-ruby/simplecov, acesso em 2022-09-26. 20

[46] Segal, Loren: *YARD: Yay! A Ruby Documentation Tool*, setembro 2022. https://github.com/lsegal/yard, acesso em 2022-09-10, original-date: 2008-02-26T00:01:52Z. 20

[47] *BankTellerExample.* https://github.com/PedroAugustoRamalhoDuarte/devs-to-bdd/tree/main/examples/BankTellerExample, acesso em 2022-09-26. 20

[48] Gil, Eric Bernd, Ricardo Diniz Caldas, Arthur Rodrigues, Gabriel Levi Gomes da Silva, Genaína Nunes Rodrigues e Patrizio Pelliccione: *Body sensor network: A self-adaptive system exemplar in the healthcare domain.* Em *SEAMS@ICSE*, páginas 224–230. IEEE, 2021. 29, 30