INSTITUTO DE MATEMÁTICA E ESTATÍSTICA - IME
UNIVERSIDADE DE SÃO PAULO - USP

TRABALHO DE FORMATURA SUPERVISIONADO

Simulador de Eventos modelados em Statecharts

Uma simulação de temperaturas atmosféricas

Processo nº 2012/23767-2, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

António Augusto Tavares Martins Miranda

supervisionado pela
Profa. Dra. Ana Cristina Vieira de Melo

São Paulo, Janeiro de 2016

A persistência é o menor caminho do êxito.

Charles Chaplin

Resumo

Devido a importância e a necessidade de validação de softwares críticos de forma robusta e com elevado grau de realismo em balões estratosféricos, foi proposto pelos pesquisadores do Instituto Nacional de Pesquisas Espaciais (INPE), a implementação de um simulador de eventos espaciais modelados em *Statecharts* para gerar casos de testes. Este simulador aplicaria estratégias de passeios sobre a máquina de estados finitos, gerado a partir da modelagem em *Statecharts* do evento espacial, para gerar as mais variadas simulações.

Portanto, para este trabalho foi desenvolvido em JAVA um simulador de eventos modelados em *Statecharts*, e ao longo do mesmo serão apresentados e analisados as implementações dos módulos (*parser*, máquina de estados finitos e simulação) do simulador.

Para demonstrar o funcionamento e a escalabilidade do simulador de eventos modelados em *Statecharts* desenvolveu-se um simulador específico para temperaturas atmosféricas, que também será analisado desde a modelagem da máquina de estados da temperatura atmosférica até a implementação.

Palavras-chave: *Statecharts* , máquina de estados finitos, simulador, estratégias de passeio, temperatura.

Abstract

Due to the importance and necessity of stratospheric balloons critical software validation in a robust way and with a high degree of realism, it was proposed by some researchers of the national institute of space researches of Brazil (INPE), the implementation of a spatial events simulator modeled in Statecharts, to generate test cases. This simulator would apply walking strategies over the finite state machine, built from the Statechart model of the spatial event, to generate the most varied simulations.

Therefore, for this work it was developed, in JAVA, a simulator of events modeled in Statecharts, and along the same work the implementations of the simulator's modules (parser, finite state machine and simulation) will be presented and analyzed.

To demonstrate how the simulator of events modeled in Statecharts works and its scalability, a specific simulator of atmospheric temperatures was developed, which will also be analyzed from the modeling of the state machine of atmospheric temperature to the implementation.

Key-words: Statecharts, finite state machine, simulator, walking strategies, temperature.

Sumário

Parte I

Parte Objetiva

Capítulo 1

Introdução

No trabalho de conclusão de curso elaborado pela Camila Achutti, sob a orientação da Profa. Dra. Ana Cristina Viera de Melo, para a obtenção do título de Bacharel em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP), foi feito um estudo e aplicação de métodos de verificação e validação de softwares em um sistema crítico real da área espacial, para explorar o tema da qualidade de softwares de missão crítica [1].

Para a concretização desse estudo, precisava-se de um software de missão crítica real que apresentasse todos os desafios de trabalhar com um software crítico, e além disso possuísse uma especificação completa para que as etapas de implementação, validação e verificação pudessem se concretizar de forma realista e consistente[1].

Nesse contexto apareceu o Software Piloto Embarcado no *Payload Data Handling Computer* (SWPDC), um software embarcado real usado para o gerenciamento de dados espaciais, que foi desenvolvido no contexto Qualidade de Software Embarcado em aplicações Espaciais (QSEE) do INPE [1].

Com o intuito de acompanhar os novos rumos da pesquisa espacial e desenvolvimento de softwares de missão crítica, decidiu-se por desenvolver uma versão do SWPDC na linguagem de programação Java[1]. Por se tratar de um software extenso e complexo, Camila Achutti implementou no seu trabalho de conclusão de curso a componente de dados do SWPDC (sua componente central).

Para auxiliar na validação do SWPDC e tornar a mesma mais consistente e realista, foi implementado um protótipo que simula o *Payload Data Handling Computer* (PDC)[1]. Esse protótipo, na sua essência, simula eventos espaciais e os provê ao SWPDC[1]. Por motivos de simplicidade e por se tratar de um protótipo, a temperatura foi o único evento espacial implementado no simulador de PDC.

O objetivo desejado com a simulação de eventos espaciais, era a de produção de amostras de eventos com um grau de realismo aceitável e que pudessem ser usadas como casos de testes. Mas como as temperaturas geradas pelo simulador de PDC eram calculadas de forma aleatória, fugindo assim do grau de realismo desejado, foi proposto pelos pesquisadores do INPE e com apoio da Profa. Dra. Ana Cristina Vieira de Melo, a implementação de um simulador de temperaturas baseado em uma modelagem, na forma de *Statecharts*, do comportamento de temperaturas na atmosfera. Esse novo simulador receberia uma Máquina de Estados Finitos (MEF), gerada a partir de uma especificação *Statechart* modelando um determinado comportamento da temperatura na atmosfera, e computaria as

amostras de temperaturas a partir de uma varredura algorítmica pelos seus vários estados.

Em suma, este trabalho apresenta a implementação de um Simulador de Eventos modelados em *Statecharts* (SES), para geração automática de casos de teste a partir de um modelo formal do comportamento de um evento. Também, fazendo parte deste trabalho e para demonstrar a escalabilidade do simulador, desenvolveu-se um módulo de simulação de temperaturas atmosféricas.

1.1 Motivação

Nos tempos de hoje, devido ao grande aumento do uso de sistemas computacionais, a atividade de teste de software é indispensável para garantir a qualidade e a confiabilidade de qualquer tipo de software. É indispensável submeter o software a casos de testes realistas, isto é, coerentes com o ambiente de operação do mesmo.

O uso de simuladores para a geração dos casos de testes mais realistas é muito comum na atividade de teste de softwares críticos, já que muitas vezes é inviável testar o software diretamente no seu ambiente de operação. No caso de softwares embarcados críticos, muitas vezes se não for garantido um elevado grau de robustez dos mesmos, corre-se um grande risco de perda permanente de equipamentos valiosos, causar danos ambientais e também de perda de vidas.

Portanto, é de extrema importância que softwares, principalmente os de missão crítica, sejam testados com o máximo de fidelidade possível em relação aos seus ambientes de operação, prevenindo assim eventuais falhas.

1.2 Objetivo

Além da produção de amostras de eventos com um grau de realismo aceitável, ao ponto de serem usadas como casos de testes em softwares críticos, o objetivo deste trabalho é também propor uma possível abordagem para se realizar e automatizar testes de software baseados em modelos.

1.3 Organização do trabalho

A parte objetiva deste trabalho foi dividida em 6 capítulos, e nos mesmos são apresentados os seguintes assuntos:

- Capítulo 1 faz a introdução do trabalho, apresentando a contextualização, motivação, objetivo e organização dos capítulos do mesmo;
- Capítulo 2 apresenta alguns fundamentos para uma melhor compreensão do trabalho e da metodologia usada;
- Capítulo 3 apresenta a arquitetura, implementação e eficiência do *parser* usado pelo SES nas leituras dos arquivos *eXtensible Markup Language* (XML) das MEF;
- Capítulo 4 apresenta arquitetura e implementação da representação interna da MEF dos eventos;

- Capítulo 5 estuda cada um dos componentes de simulação do SES e apresenta um exemplo de implementação de um simulador de temperaturas atmosféricas;
- Capítulo 6 apresenta as contribuições e conclusões obtidas a partir deste trabalho, mas também as oportunidades de trabalhos futuros.

Capítulo 2

Fundamentos

2.1 Ferramentas

Nesta seção serão apresentadas, sem entrar em muitos detalhes, duas das principais ferramentas usadas para idealizar todo este trabalho, os *Statecharts* e o software Geração Automática de Casos de Teste Baseada em Statecharts (GTSC).

2.1.1 Statecharts

O *Statechart*, originalmente criado por David Harel em 1987 e hoje parte do *Unified Modeling Language* (UML), é uma notação para especificar o comportamento de sistemas representáveis como MEF [4], ainda podemos dizer que um *Statechart* é uma linguagem de modelagem visual focada na modelagem de sistemas de tempo discreto [2] .

Agora de um ponto de vista mais técnico e segundo o próprio Harel [3], *Statecharts* é uma "extensão dos diagramas de estados convencionais com essencialmente três elementos, lidando, respectivamente, com as noções de hierarquia, concorrência e comunicação; estes, transformam a linguagem de diagramas de estados numa linguagem de descrição econômica e altamente estruturada; *Statecharts* são, portanto, compactos e expressivos – pequenos diagramas que podem expressar comportamentos complexos – bem como a composicionais e modulares" [1].

Statecharts vs MEF

Algumas diferenças fulcrais entre *Statecharts* e MEF, apresentadas nos slides *Test Generation: Statecharts* do Aditya P. Mathur [4]:

- **Configuração:**
  - MEF: apenas um estado pode estar ativo num dado instante de tempo;
  - *Statechart*: múltiplos estados podem estar ativos num dado instante de tempo.

[1] Tradução livre de "extended conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive - small diagrams can express complex behavior - as well compositional and modular."

Enquanto que uma MEF move de um estado para outro, um *Statechart* move de uma configuração (de estados) para outra.

- **Hierarquia:**
  - MEF: um estado é uma entidade atômica;
  - *Statechart*: um estado pode ser decomposto em subestados.
- **Memória interna:**
  - MEF: não tem memória;
  - *Statechart*: os estados têm memória e um conjunto de ações associados a esse estado podem modificá-la. Uma mudança na memória interna pode levar a uma transição.

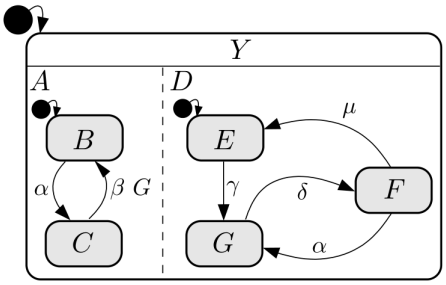

Figura 2.1: Modelagem da concorrência em *Statecharts* [3, 2]. O estado Y é formado pelos componentes concorrentes A e D.

2.1.2 GTSC

O software Geração Automática de Casos de Teste Baseada em *Statecharts* (GTSC), permite, tal como o nome indica, a geração automática de casos de teste baseando numa modelagem comportamental em *Statecharts* e em Máquinas de Estados Finitos (MEF) [5].

O GTSC possui uma série de ferramentas que permitem ao projetista de teste ter um ambiente de modelagem comportamental de software em *Statecharts* e em MEF para gerar casos de teste [5].

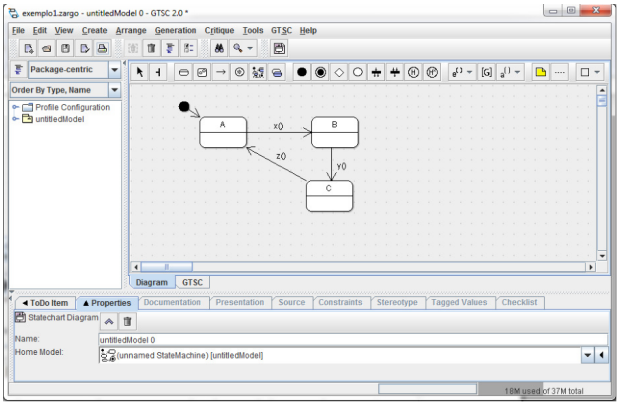

Figura 2.2: Ambiente de modelagem no GTSC[5].

2.2 Metodologia

Gerar arquivos XML de MEF de eventos modelados em *Statecharts*, através do GTSC. Tais arquivos XML serão alimentados ao SES, que por sua vez, fazendo uso de um *parser* adequado, extrairá as informações necessárias para a construção em memória dessas MEF.

Por fim, usando estratégias de passeios sobre a MEF e técnicas de geração de erros, serão construídas as amostras de eventos que serão usadas como casos de testes.

Devido a limitações técnicas do GTSC, que não está pronto para criar a MEF plana a partir de qualquer modelagem em *Statechart*, é recomendável, sempre que a modelagem em *Statechart* for um tanto complexa, criar a MEF do evento diretamente no arquivo XML. Trata-se de um trabalho um tanto braçal, mas foi a única solução para o problema encontrada a curto prazo.

A MEF usada pelo simulador no capítulo 5, capítulo que apresenta a implementação de um simulador de temperaturas atmosféricas, foi criado diretamente no arquivo XML da MEF.

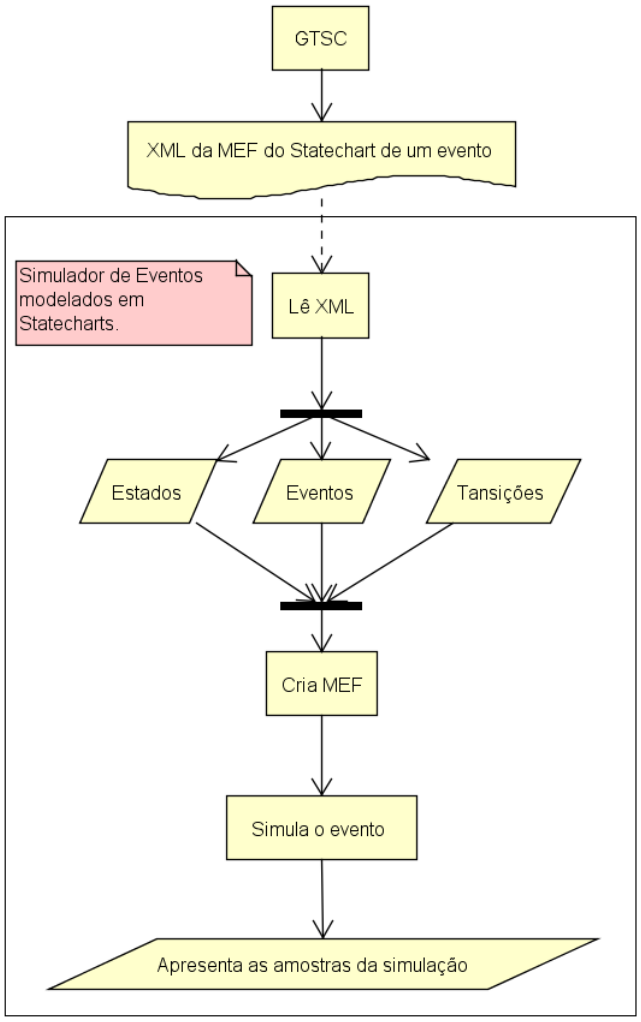

Figura 2.3: Fluxograma resumindo a metodologia do trabalho.

2.3 Algumas questões fundamentais sobre o SES

Não só para esclarecer possíveis dúvidas, mas também para justificar algumas decisões tomadas ao longo do desenvolvimento do SES, eis algumas questões e respostas sobre a implementação e o funcionamento do mesmo.

O que serão os eventos?
A princípio, pode ser considerado como um evento, qualquer propriedade modelável em uma MEF (diretamente ou indiretamente, isto é, por meio de outros tipos de modelagem, como neste caso *Statecharts*).

Porquê *Statecharts*?
Dada a sua natureza compacta, expressiva, composicional e modular, o que o torna numa das ferramentas de modelagem preferencial por parte de engenheiros (por exemplo, engenheiros aeronáuticos) na modelagem de sistemas reativos e complexos[3], e também por ser muito usada, pelos pesquisadores do INPE, na modelagem de softwares embarcados[5], *Statecharts* foi escolhido como o instrumento de modelagem preferencial para este trabalho.

Como funcionará o simulador?
O SES não trabalhará diretamente com a modelagem em *Statechart* para fazer as simulações, mas sim a sua MEF equivalente. Será nessa MEF que o SES aplicará diferentes estratégias de passeio (ou varredura), que não só permitirão gerar diferentes resultados nas simulações, como explorar sob várias perspectivas a MEF. Mais para a frente, essas estratégias de passeio serão discutidas com mais detalhes.

Porquê o simulador usa a MEF em vez do *Statechart*?
Essa escolha foi feita simplesmente por motivos de simplicidade de programação. Programaticamente falando, é muito mais simples a interpretação e uso das informações descritas na forma de uma MEF do que como um *Statechart*. A implementação da concorrência por parte do *Statechart* torna a sua interpretação muito mais complexa.

Porquê o GTSC?
Como o GTSC, além de oferecer um ambiente de modelagem em *Statecharts*, também oferece um meio de conversão de um modelo *Statechart* para a sua MEF equivalente, ele foi escolhido para fazer o papel de "interface" para a geração de *Statecharts* e de gerador de MEF para o SES.

Quão escalável é o SES?
O SES é escalável a vários tipos de simulações, ou seja, um usuário poderá implementar um simulador para um determinado evento, com as estratégias de passeio (ou varredura), e acoplá-lo ao SES. Mais para a frente, será exemplificado como adicionar um novo simulador ao SES.

Capítulo 3

Parser

Como descrito no fluxograma da figura 2.3, apresentado no capítulo anterior, o SES usa o XML da MEF gerada pelo GTSC para poder construir na memória uma réplica dessa mesma MEF, que por sua vez será usada como base para a realização das simulações.

Portanto, para permitir que o SES consiga extrair informações do XML da MEF, foi implementado um *parser* de arquivos XML chamado de *FiniteStateMachineParser*.

O *FiniteStateMachineParser* fornece os meios necessários para obter e implementar a extração de informações como os estados, eventos e transições da MEF descritos no arquivo XML.

3.1 Tecnologias

Por ser simples, intuitivo, e pela eficiência do *parser* não ser prioridade nesta primeira versão do SES, o *Document Object Model* (DOM), importada do vasto ecossistema do Java [6, 7], foi a tecnologia usada como base para a implementação do *FiniteStateMachineParser*.

O DOM varre um arquivo XML e carrega-o em memória em forma de árvore, com o objetivo de permitir uma melhor manipulação e facilitar o deslocamento entre os vários elementos do arquivo XML [7].

3.2 Estrutura e implementação

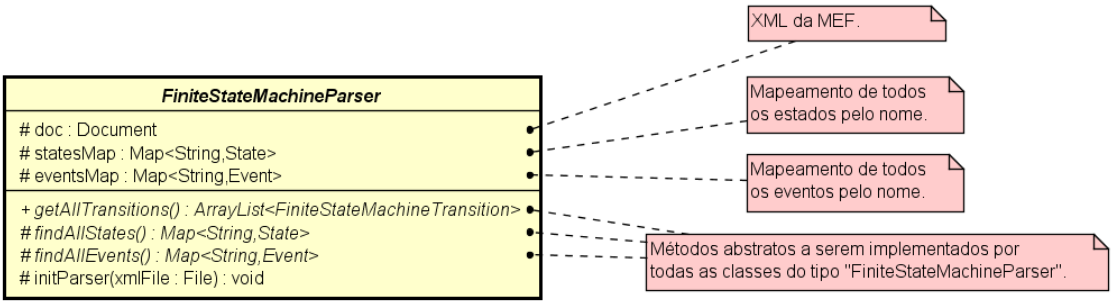

Figura 3.1: Classe *FiniteStateMachineParser*.

A variável *doc* do tipo *Document*, ilustrada na figura 3.1, armazena o XML da MEF, portanto, todo o "parseamento" será feito sobre ela. A mesma é inicializada a partir de um arquivo XML lido pelo método *initParser*.

Para garantir uma maior escalabilidade no "parseamento" de arquivos XML das MEF, ou seja, permitir a leitura de arquivos XML em formatos diferentes de MEF, o *FiniteStateMachineParser* foi implementado como uma classe abstrata, com os seguintes métodos abstratos:

- *getAllTransitions*;
- *findAllStates*;
- *findAllEvents*.

Os métodos *findAllStates* e *findAllEvents* implementarão respectivamente, tendo em conta a estrutura XML da MEF, a extração das informações referentes aos estados e eventos, retornando estruturas de mapeamentos de um nome para um estado e de um nome para um evento.

Por serem métodos essenciais ao funcionamento do *parser* e na criação da lista de transições da MEF, os mesmos têm visibilidade do tipo *protected*. Isso é uma medida de segurança para garantir que apenas classes que herdam da classe *FiniteStateMachine*, ou classes que pertencem ao mesmo pacote da classe *FiniteStateMachine*, vejam e manipulem tais métodos.

O método *getAllTransitions*, implementa, dependendo da estrutura XML da MEF, a forma como serão extraídas as informações que permitem construir as transições da MEF. A mesma retorna uma lista com todas as transições da MEF.

O *getAllTransitions* deve fazer uso das seguintes variáveis auxiliares de mapeamento:

- *statesMap* – guarda e faz o mapeamento dos nomes dos estados para os próprios estados;
- *eventsMap* – guarda e faz o mapeamento dos nomes dos eventos para os próprios eventos.

Além de servirem como meio de armazenamento e consulta de estados e eventos, têm um papel importante no desempenho eficiente do *parser*.

Durante o processo de "parseamento" das transições da MEF, elas permitem a determinação em tempo constante do evento e estados envolvidos numa transição. Por conseguinte, permitem que a lista de transições seja criada de forma mais eficiente.

Por exemplo, se em vez de usar variáveis do tipo *Map*, forem usadas variáveis do tipo listas simples (ou mesmo *arrays*), a determinação dos estados e evento envolvidos numa transição levariam tempo linear. Supondo que o número de estados e o número de eventos sejam muito grandes, teríamos uma queda significativa na eficiência da criação da lista de transições, e por conseguinte, uma queda na eficiência do próprio *parser*.

3.3 Escalabilidade

Para escalar o *parser* do SES, permitindo que o mesmo suporte arquivos XML com diferentes formatos de MEF, basta estender a classe abstrata *FiniteStateMachine* e implementar os métodos *getAllTransitions*, *findAllStates* e *findAllEvents*.

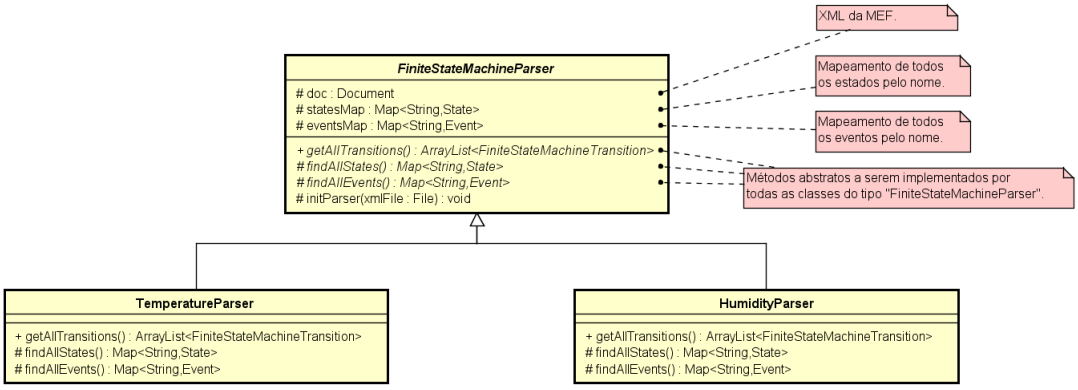

Figura 3.2: Escalabilidade do *parser*.

No apêndice A é apresentado a implementação da classe *TemperatureParser* da figura acima.

3.4 Relação com a MEF

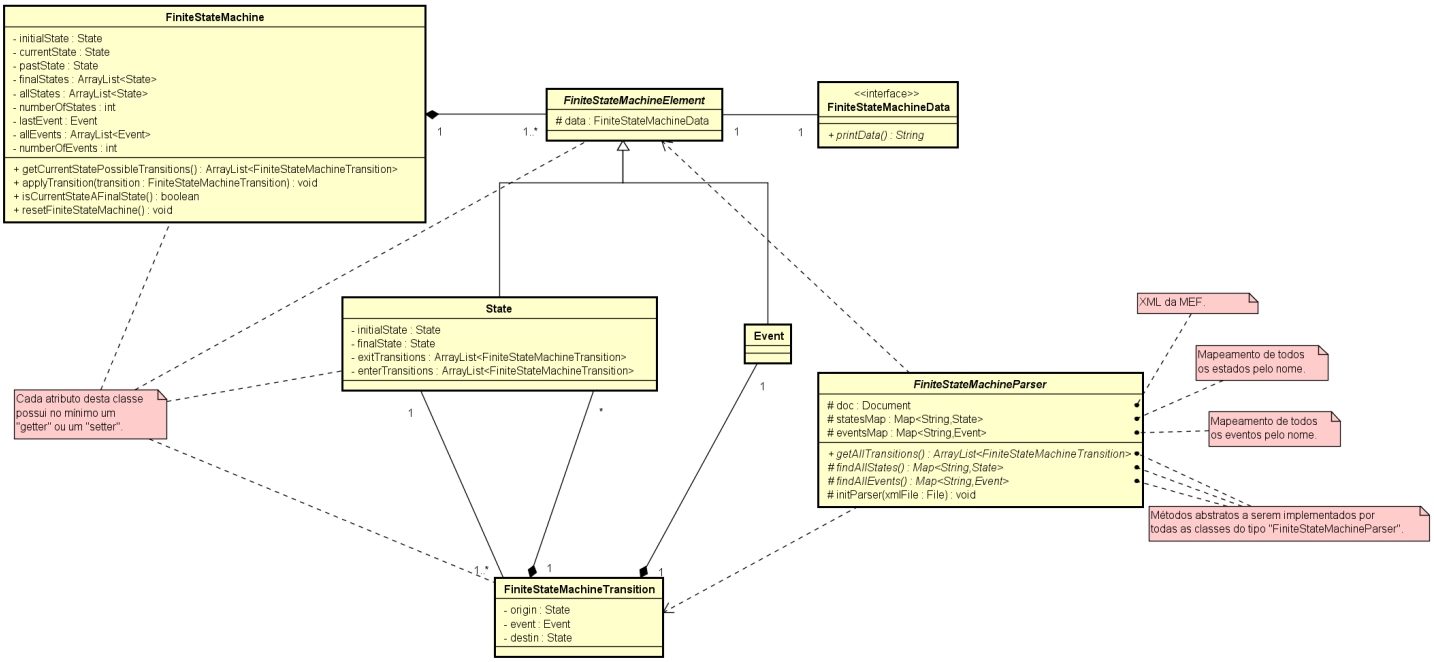

Figura 3.3: Relação entre o *parser* e a MEF do SES.

Tal e qual está ilustrado na figura 3.3, o *parser* tem uma relação de dependência para com a MEF. O *parser* necessita do estado, evento e transição para poder gerar as respectivas listas de estados, lista de eventos e listas de transições da MEF.

3.5 Discussão

Infelizmente, *parsers* baseados no DOM são lentos e consomem muita memória quando carregam arquivos XML com muitos dados [7]. Como o tamanho da MEF é proporcional ao grau de complexidade da MEF e ao grau de realismo que o projetista de testes deseja na sua modelagem, a MEF resultante pode ser enorme, portanto, a substituição do mesmo por uma tecnologia mais eficiente é fortemente recomendável.

Uma possível alternativa ao DOM é o *Simple API for XML* (SAX). O SAX é mais rápido, usa menos memória, e utiliza funções *callback* para informar aos clientes sobre a estrutura do XML [8].

Fora isso, a implementação, funcionamento e escalabilidade do *parser* ficaram bastante satisfatórios.

Capítulo 4

Máquina de Estados Finitos

A representação interna da MEF definida no XML é dado pela classe *FiniteStateMachine*. A classe *FiniteStateMachine* é uma composição de *FiniteStateMachineElement*, ou seja, estados e eventos, o que implica também uma composição de transições. Uma transição é dada pela classe *FiniteStateTransition*, que também é uma composição de estados e eventos.

Como a MEF será a base de qualquer simulação realizada, a mesma tem que ser implementada de forma eficiente e ter um uso intuitivo.

4.1 Estrutura e implementação

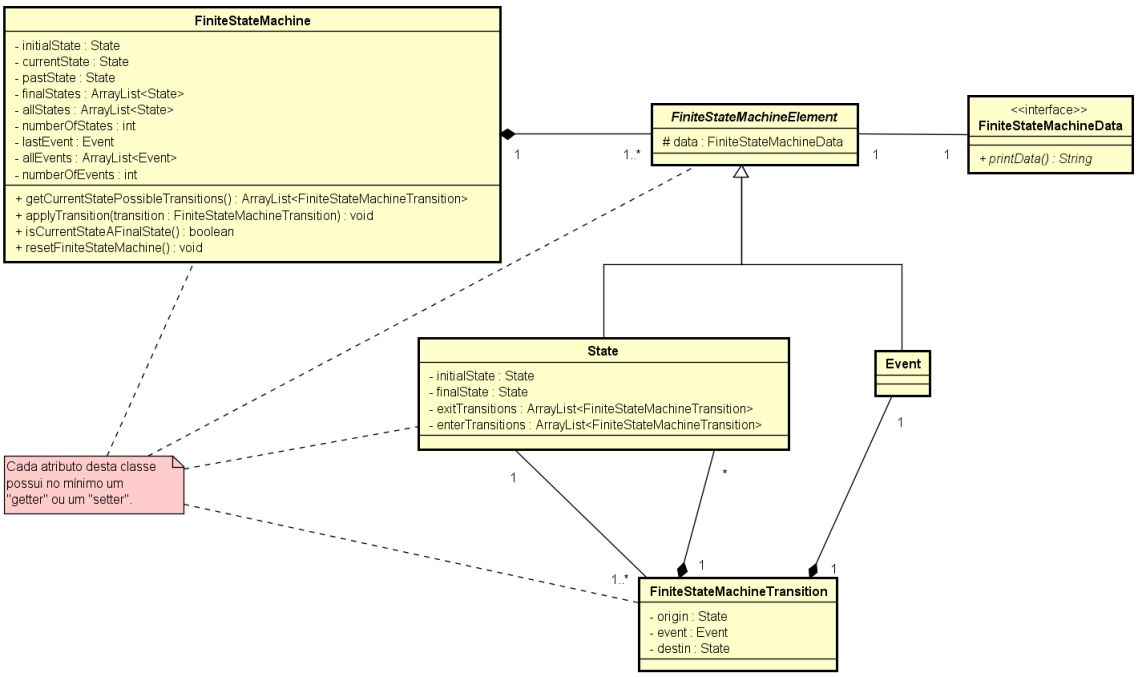

Figura 4.1: Diagrama de classes da MEF do SES.

Todos eventos e estados, estendem da classe abstrata *FiniteStateMachineElement*, que por sua vez representa todos os elementos atômicos que compõe uma MEF, ou seja, os estados e os eventos. Ela

ainda contém a variável *data* do tipo *FiniteStateMachineData*, que serve para armazenar o tipo de dado associado a um estado ou evento.

O *FiniteStateMachineData* implementa os tipos de dados a serem usados pelos elementos da MEF. Qualquer *FiniteStateMachineElement*, pode assumir um valor qualquer, desde que esses valores implementem a interface *FiniteStateMachineData*.

Um estado, é constituído pelos seguintes atributos:

- *initialState* – booleano que apenas assume valor verdadeiro quando um estado é o estado inicial da MEF;
- *finalState* – booleano que assume valor verdadeiro quando um estado é um estado final da MEF;
- *exitTransitions* – lista com todas as transições de saída a partir do estado em questão;
- *enterTransitions* – lista com todas as transições cujo o destino é o estado em questão.

Os atributos *initialState* e *finalState* permitem determinar se um estado é inicial ou final de forma bem rápida e eficiente, enquanto que os atributos *exitTransitions* e *enterTransitions* permitem computar de forma rápida, eficiente e prática, respectivamente, todos os eventos que levam a um determinado estado e todos os estados que possuem transições para um determinado estado.

A classe evento não possui nenhum atributo e nem métodos, ou seja, serve apenas para designar um evento.

Uma transição é caracterizada pelos seguintes elementos de uma MEF: estado de origem, evento de transição e estado de destino.

4.2 MEF

O construtor da classe *FinteStateMachine* recebe como parâmetros as listas de estados, eventos e transições, e computa informações tais como, estados inicial e finais e estabelece as conexões entre os vários estados da MEF (transições).

Para computar os estados inicial e finais, é usado o método *setInitialAndFinalStates*, que recebe uma lista com todos os estados da MEF e faz uma busca procurando os estados inicial e finais.

Para criar as transições entre os estados da MEF temos o método *connectTransitions*. Percorrendo a lista com todas as transições da MEF, dada uma transição qualquer, o *connectTransitions* cria as conexões entre os estados da MEF, pela simples adição da transição na lista dos *exitTansitions* do estado de origem da transição e da adição da mesma transição na lista dos *enterTransitions* do estado de destino da transição. Ao final da lista de transições da MEF, o *connectTransitions* terá estabelecido todas as conexões entre os estados da MEF.

Pensado na simulação de eventos, e com o intuito de facilitar a mesma, foram implementados os seguintes métodos na MEF:

- *getCurrentStatePossibleTransitions*;
- *applyTransition*;
- *isCurrentStateAFinalState*;
- *resetFiniteStateMachine*.

O *getCurrentStatePossibleTransitions*, simplesmente retorna a lista de *exitTransitions* do estado corrente. Este método será (muito) útil aos algoritmos que implementam as estratégias de passeio sobre a MEF.

O *applyTransition*, dada uma transição válida para o estado corrente, aplica essa transição a partir do estado corrente, ou seja, anda um passo na direção definida pela transição. Este método foi concebido para ser usado logo após a escolha do novo passo a ser dado na MEF.

Caso seja necessário parar a simulação de uma amostra num determinado estado final (por definição do usuário), é possível usar o *isCurrentStateAFinlaState* para validar se o estado é final, e para reiniciar a simulação de uma amostra a partir do estado inicial da MEF, faz-se o uso do método *resetFiniteStateMachine*.

4.3 Escalabilidade

O único elemento da MEF escalável é seu tipo de dados. Os tipos de dados da MEF são usados nos dados dos estados e nos dados dos eventos, e o tipo de dados usado nos eventos pode ser igual ou diferente ao tipo de dados usado nos estados.

Para criar e adicionar um novo tipo de dados à MEF, basta criar uma classe que implemente a interface *FiniteStateMachineData* e implemente o método *printData*.

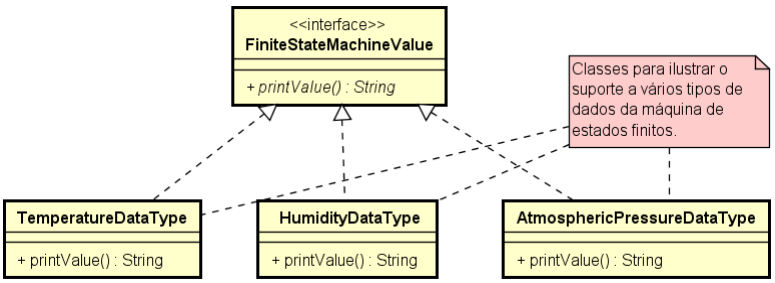

Figura 4.2: Adicionando novos tipos de dados a MEF.

4.4 Comentários

A capacidade da MEF aceitar vários tipos de dados, tanto para os estados, como para os eventos de transições, é uma funcionalidade muito importante para o SES, visto que o mesmo tem que suportar MEF para os mais variados tipos de eventos.

A estrutura da MEF construída, atendeu de forma satisfatória as necessidades do SES.

Capítulo 5

Simulador de Eventos modelados em Statecharts

O objetivo deste capítulo é apresentar, sob um ponto de vista técnico, a arquitetura, implementação e funcionamento do SES. Pretende-se concretizar esse objetivo abordando os seguintes temas:

- Arquitetura e escalabilidade do SES;
- Configurações do SES;
- Implementação do SES;
- Discussão e implementação das estratégias de passeios sobre a MEF;
- Discussão e implementação dos métodos de geração de erros nas simulações;
- Formato da MEF de um evento;
- Implementação e adição de um simulador de temperaturas ao SES.

5.1 Arquitetura e escalabilidade

Uma das qualidades mais desejadas no SES, é a capacidade de suportar vários tipos de simulações de eventos. Com isso em mente, a classe *Simulator*, responsável pela implementação das simulações, foi projetada como uma classe abstrata.

Como uma classe abstrata, é possível generalizar todos os tipos de simuladores de eventos, e por conseguinte, implementar as necessidades e funcionalidades comuns entre eles. Por exemplo, as funções responsáveis pela preparação de uma simulação e o ciclo principal de uma simulação.

A diferença fundamental entre os vários tipos de simuladores está na forma como eles implementam o passeio sobre as várias transições da MEF. Como esse passeio dependerá muito do tipo da simulação, o mesmo foi projetado de forma abstrata.

Por fim, é possível implementar e adicionar, de forma bem prática, vários tipos de simulações ao SES, visto que só teremos que estender a classe abstrata *Simulator* do SES e implementar o método abstrato que realiza o passeio.

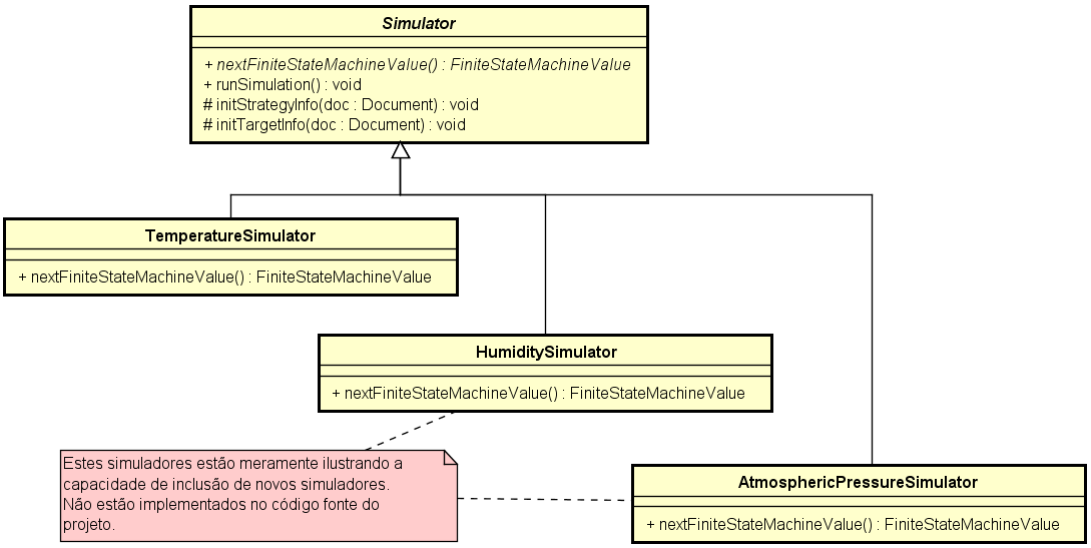

Figura 5.1: Adicionando vários tipos de simuladores.

5.2 Configurações

Para permitir a configuração de uma simulação, foram criados arquivos de configurações no formato XML. Existem arquivos de configuração na raiz do pacote da classe *Simulator* e nas raízes dos pacotes de implementação de cada simulador específico.

O XML de configuração localizado na raiz da classe *Simulator*, além de servir para definir que tipo de simulação será executado, também serve para agrupar configurações comuns a todos os simuladores, como por exemplo, o número e tamanho das amostras, e estados de parada.

Arquivo de configuração:

```
<config>
  <simulator name="TemperatureSim"/>
  <sample size="13" num="5"/>
<!--  Opcional -->
<!--  <stop> -->
<!--    <final name="-56.46"/> -->
<!--    <final name="[-10.00,0.00]"/> -->
<!--  </stop> -->
</config>
```

O bloco *simulator* é usado para definir um simulador, dado pelo atributo *name*. O bloco *sample*, através dos seus atributos *size* e *num*, permitem definir respectivamente, o tamanho e número de amostras de um evento.

Caso um usuário do SES deseje que o tamanho das amostras de eventos esteja relacionado com a presença num determinado estado (final ou não final) da MEF, em vez de ser limitado pelo atributo *num* do elemento *sample*, usa-se o bloco *stop*. Esse bloco permite definir os estados de parada de uma amostra, exemplo:

```
<stop>
  <final name="-56.46"/>
  <final name="[-10.00,0.00]"/>
</stop>
```

O sub-bloco *final* define um estado de parada, identificado pelo atributo *name*.

Os arquivos de configuração que se encontram nas raízes dos pacotes dos simuladores, são para definir configurações específicas do próprio simulador e do seu tipo de simulação. Na seção de configurações do simulador de temperaturas atmosféricas, veremos um exemplo de um arquivo de configuração específico.

5.3 Implementação

As classes responsáveis pela realização de uma simulação são: as classes *Simulation* e *Simulator*.

A classe *Simulation* implementa o *main* da aplicação e carrega, a partir do arquivo XML de configurações que se encontra no mesmo pacote das classes *Simulation* e *Simulator*, informações importantes para a inicialização e configuração da instância da classe *Simulator* que irá executar a simulação.

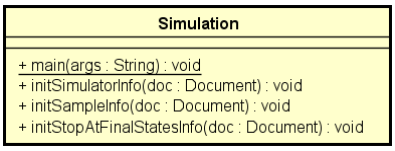

Figura 5.2: *Simulation*

Além do método *main*, responsável pela execução de todo o programa, os principais métodos da classe *Simulation* são:

- *initSimulatorInfo* – recebe um documento XML de configurações e extrai a informação sobre que simulador deverá ser usado na simulação;
- *initSampleInfo* – recebe um documento XML de configurações e extrai informações sobre o número de amostras a serem geradas na simulação e o tamanho das mesmas;
- *initStopAtFinalStatesInfo* – recebe um documento XML de configurações e extrai, caso exista, a lista dos estados de parada de uma amostra.

A classe *Simulator*, mencionada anteriormente, é a classe abstrata ao qual todos os simuladores devem estender, ou seja, é a classe que generaliza todos os tipos de simuladores.

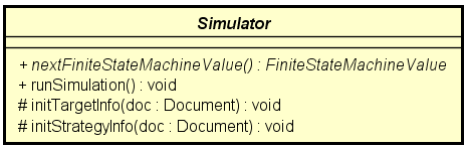

Figura 5.3: *Simulator*

Ela faz uso dos métodos *initTargetInfo* e *initStrategyInfo* para carregar respectivamente, a partir do documento XML de configurações, o arquivo que contém a MEF do evento a ser modelado e a estratégia a ser empregada ao longo do seu varrimento.

Depois do método abstrato *nextFSMData*, que retorna o dado associado a uma mudança de estados na MEF (visto com mais detalhes no exemplo de um simulador de temperaturas atmosféricas), o método *runSimulation* é o principal método classe *Simulator*.

O método *runSimulation* implementa o ciclo principal de uma simulação e cria corretamente as amostras de um evento, em outras palavras, ele coordena todo o processo da simulação.

Implementação do método *runSimulation*:

```
public void runSimulation() {
  int atualSize;
  FiniteStateMachineData currentFSMData;
  for (int samplesMade = 0; samplesMade < numOfSamples; samplesMade++) {
    atualSize = 1;
    System.out.println("Sample " + (samplesMade + 1) + ":");
    System.out.print(" " + initialValue);
    while (true) {
      currentFSMData = nextFSMData();
      System.out.print(" " + currentFSMData.printData());
      if (stopAtFinalStates) {
        if (finiteStateMachine.isCurrentStateAFinalState()) {
          if (stopFinalStatesNames.contains(
              finiteStateMachine.getCurrentState().getData().printData())) {
            break;
          }
        }
      } else if (atualSize == (sizeOfSample - 1)) {
        break;
      } else {
        atualSize++;
      }
    }
    System.out.println();
    finiteStateMachine.resetFiniteStateMachine();
  }
}
```

5.4 Estratégias de passeio

Os valores computados para uma amostra de um evento dependem do caminho percorrido na MEF do evento, ou seja, dependendo da forma como é percorrida a MEF, teremos amostras com valores diferentes.

Do ponto de vista de customização de uma simulação, isso é muito vantajoso porque permite-nos definir estratégias de passeio mais apropriadas aos tipos de simulação e às funcionalidades que se pretendem validar. Essas estratégias de passeio não passam de algoritmos que definem como uma MEF será percorrida.

Programaticamente falando, no projeto, uma estratégia de passeio é uma interface, chamada de *WalkStrategy*, com o método abstrato *path*, que implementa a lógica de uma determinada estratégia.

O método *path* deve receber o estado corrente da MEF, e a partir dele computar a transição que vai levar ao próximo estado da MEF e consequentemente da simulação.

Para criar uma estratégia de passeio e adicioná-la ao projeto, basta implementar uma nova classe que implemente a interface *WalkStrategy*.

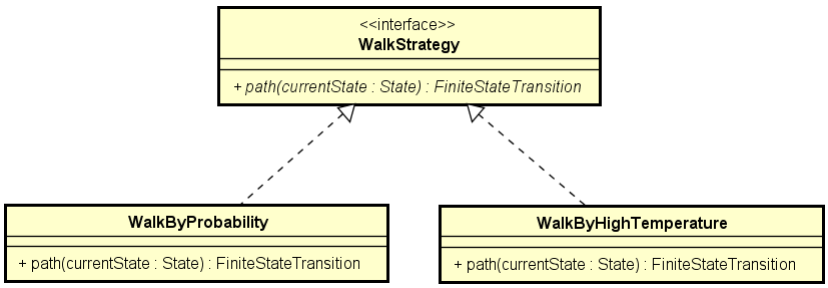

Figura 5.4: Escalabilidade das estratégias de passeio.

Algumas estratégias de passeio são aplicáveis a qualquer MEF e outras são bem específicas ao tipo de simulação. Designaremos de estratégias gerais de passeios às estratégias que independem da MEF e do simulador, e de estratégias particulares de passeios às estratégias que dependem da MEF e do seu simulador.

Exemplos de possíveis estratégias que independem da MEF e do tipo de simulação da mesma:

- todos os caminhos da MEF;
- escolha de transições da direita para a esquerda (quando o estado corrente possuir mais de uma transição);
- transição por probabilidade (definir uma probabilidade de ocorrência para cada transição e escolher uma transição de acordo com essa probabilidade).

No projeto, por motivos de simplicidade, foi implementado como estratégia de passeio geral a transição por probabilidade. Para implementar essa estratégia, tendo em conta que foi assumindo a existência

de uma probabilidade de ocorrência de um valor errado numa amostra (discutido com mais detalhes na seção de geração de erros), a probabilidade restante foi dividida igualmente entre as transições de saída do estado corrente, em outras palavras, cada transição tem um intervalo (probabilístico) de ocorrência. Para decidir qual transição efetuar, é computado uma probabilidade aleatória, onde o valor dessa probabilidade é um número sorteado dentro do intervalo de zero (inclusivo) a um (exclusivo), e depois é determinado em que intervalo probabilístico essa probabilidade se enquadra. Por fim se escolhe a transição associada a esse intervalo probabilístico.

Implementação da classe *WalkByProbability*:

```
public class WalkByProbability implements WalkStrategy {
  private double probabilityOfError;
  private FiniteStateMachineTransition lastTransition;
  public WalkByProbability(double probabilityOfError) {
    this.probabilityOfError = probabilityOfError;
  }
  private int getProbabilityInterval(double validTransitionsProbability, double
     prob, int sz) {
    double fDelimiter, lDelimiter;
    int i = 1;
    fDelimiter = 0;
    lDelimiter = validTransitionsProbability;
    while (!(fDelimiter <= prob && prob < lDelimiter)) {
      fDelimiter = lDelimiter;
      lDelimiter = (++i)*validTransitionsProbability;
    }
    return i-1;
  }
  @Override
  public FiniteStateMachineTransition path(State currentState) {
    int index, numExitTransitions;
    double validTransitionsProbability, randomValue;
    if (currentState.getExitTransitions() != null) {
      if (currentState.getExitTransitions().size() > 0) {
        numExitTransitions = currentState.getExitTransitions().size();
        validTransitionsProbability = (1.0 - probabilityOfError) /
           numExitTransitions;
        randomValue = Math.random();
        if (randomValue >= 0 && randomValue < (1.0 - probabilityOfError)) {
          index = getProbabilityInterval(validTransitionsProbability,
             randomValue, currentState.getExitTransitions().size());
          lastTransition = currentState.getExitTransitions().get(index);
        } else {
          return null;
        }
      }
    }
    return lastTransition;
  }
}
```

Para estratégias particulares de passeios, recomenda-se a sua implementação dentro do pacote do simulador específico, não só por uma questão de organização mas para facilitar o atendimento das

necessidades específicas do mesmo.

No exemplo de um simulador de temperaturas atmosféricas falaremos e exemplificaremos uma estratégia de passeio específica a implementação do simulador de temperaturas.

5.5 Geração de erros

A validação de qualquer software tem que ser feita tanto para valores de *input* esperados (corretos), como para valores de *input* inesperados (incorretos), isto é, o comportamento do software tem que ser validado para todos os tipos de entradas, inclusive as erradas. Daí surge a necessidade do simulador ter a capacidade de gerar e incluir erros nas amostras de eventos.

A geração de erros nas amostras de um evento, pode ser realizado de várias formas, mas as propostas para a implementação de erros vistas e analisadas ao longo desse projeto foram:

- inclusão de eventos errados diretamente na modelagem da MEF do evento;
- mutação da MEF;
- inclusão de dados perturbados nas amostras dos eventos.

A inclusão de eventos errados diretamente na modelagem da MEF é uma solução não programática, prática e consideravelmente simples (dependendo da modelagem da MEF), mas infelizmente pode ser bastante limitada. Para conseguir um número razoável de valores errados e diversificados, a modelagem teria que incluir um número considerável de valores errados, o que poderia aumentar consideravelmente o tamanho da MEF e por conseguinte a complexidade da modelagem.

A geração de erros a partir da mutação da MEF, inspirado nos testes por mutação, consiste em alterar o comportamento da MEF, em tempo de execução, de forma que esta produza dados errados ao longo da simulação. Não se trata de uma solução muito complexa em termos de implementação mas, sem dúvida, que é uma solução custosa em termos de memória, visto que além de preservar em memória a MEF alterada com erros, seria necessário preservar a MEF original, para o caso da necessidade de um eventual restauro.

A inclusão de dados errados na amostra por meio de perturbação de dados, é uma solução prática, e por não necessitar do armazenamento de mais de uma MEF na memória, é bem mais econômica em termos de gasto de memória. Basicamente, consiste em gerar dados que estão fora do comportamento ideal do evento, ou seja, trata-se de uma função que computa apenas dados errados para o evento que está sendo simulado. Também, dependendo de como essa função for implementada, ela pode garantir uma geração diversificada de valores errados.

Neste projeto, por razões de simplicidade e custo/benefício em relação aos métodos da modelagem de eventos errados diretamente na MEF e mutação da MEF, foi implementado o método da perturbação de dados. Esse método de geração de erros será abordado com mais detalhes ao longo da seção do simulador de temperaturas atmosféricas.

5.6 Estrutura do XML da MEF

O *PerformCharts Markup Language* (PcML) é uma técnica utilizada para descrever um modelo *Statecharts* com suas respectivas características, estados e transições [9, 10].

A ferramenta GTSC a partir do PcML gera um arquivo XML que representa uma MEF plana [5], ou seja, a MEF equivalente do *Statechart*.

Quando é feita a transformação de um *Statechart* em PcML para a MEF plana, o arquivo XML gerado (pelo GTSC) descreve o nome dos estados e seu tipo no bloco <STATES >, o nome dos eventos e os seus valores no bloco <EVENTS >e no bloco <TRANSITION >são descritos o nome do estado atual, o nome do estado destino, o valor da transição de entrada e de saída do estado correspondente [9].

No apêndice B encontra-se a MEF usada no exemplo do simulador de temperaturas atmosféricas que será apresentado na próxima seção.

5.7 Simulador de temperaturas atmosféricas

Nesta seção demonstraremos a escalabilidade do SES, implementado e adicionando um simulador de temperaturas atmosféricas ao mesmo. Também será apresentado em detalhe a modelagem da MEF da temperatura.

A razão pela qual foi escolhido um simulador de temperaturas, além de se tratar de um evento de "fácil" simulação, deve-se pelo fato da primeira versão do simulador de eventos espaciais, usado para testar o SWPDC, simular apenas temperaturas.

5.7.1 Construção da MEF de temperaturas

O primeiro problema encontrado na simulação de eventos baseado em *Statecharts* foi como modelar o evento (neste caso a temperatura).

Para modelar o evento temperatura, foi usado uma fórmula que descreve o comportamento da temperatura na atmosfera terrestre (*T(h)*), da troposfera até a estratosfera, em função da altura (*h*) [11]:

$$T(h) = \begin{cases} -131.21 + 0.00299h, & \text{se} \quad h > 25000 \\ -56.46, & \text{se} \quad 11000 < h < 25000 \\ 15.04 - 0.00649h, & \text{se} \quad h < 11000 \end{cases}$$

As temperaturas calculadas por essa fórmula são dadas em graus Célsius e o parâmetro altura é em metros. Os intervalos de alturas presentes na fórmula representam as seguintes camadas da atmosfera terrestre:

- h <11000 – (aproximadamente) troposfera;
- 11000 <h <25000 – (aproximadamente) tropopausa;

- h >25000 – (aproximadamente) estratosfera.

Como a função descrita acima é contínua (para valores de temperaturas entre a troposfera até a estratosfera) e com infinitos resultados, é impossível traduzi-la diretamente para um *Statechart*, ou consequente, MEF. Logo, é necessário limitar e discretizar a imagem da função de modelagem da temperatura.

Descrevendo um movimento ascendente de um balão estratosférico, e assumindo que o mesmo parte do nível do mar e atinge a altura máxima de aproximadamente 43883 (valor segundo a fórmula, onde a temperatura atinge aproximadamente zero graus centígrados na estratosfera), foi possível modelar a seguinte MEF da temperatura:

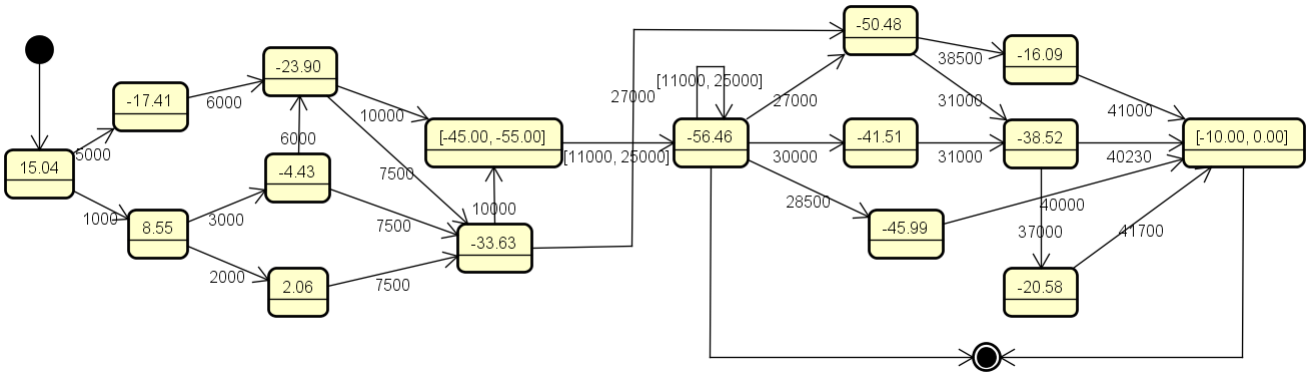

Figura 5.5: MEF do comportamento da temperatura ao longo de determinados movimentos ascendentes na atmosfera.

Onde a altura representa o valor da transição e o estado representa a temperatura. Como podemos constatar pela imagem, uma temperatura pode ser representada por apenas um único valor ou por um intervalo de valores.

A representação de uma temperatura sob a forma de um intervalo, serve simplesmente para salientar a natureza contínua da temperatura.

Essa MEF de temperatura é apenas a união de algumas temperaturas verificadas em determinadas alturas do movimento ascendente do balão estratosférico.

5.7.2 Configurações

Tal como vimos na seção de configurações do capítulo anterior, cada simulador específico pode ter um arquivo de configurações. Nesta seção serão apresentadas as possíveis configurações que podemos submeter ao simulador de temperaturas atmosféricas.

O arquivo de configuração específico de um simulador segue a mesma estrutura do arquivo de configuração discutido no capítulo anterior. A diferença está apenas nos blocos que cada um contém.

O simulador de temperaturas atmosféricas possui os seguintes blocos de configurações:

- *target* – o nome indicado pelo atributo *name* identifica o arquivo XML da MEF da temperatura;
- *temperature* – através dos atributos *ini*, *max*, *min* e *abs* é possível definir respectivamente a temperatura inicial da simulação, temperaturas mínimas e máximas possíveis e a representação da temperatura na MEF como um estado ou evento de transição (note que esse campo tem que estar obrigatoriamente de acordo com a modelagem da MEF);
- *temperature_within_range* – o atributo *pick* representa o algoritmo a ser aplicado na discretização dos intervalos de temperaturas (visto com mais detalhes na seção do tipo de dados temperatura);
- *error* – o atributo *prob*, permite definir a probabilidade de ocorrência de erros numa amostra de temperaturas;
- *strategy* – o atributo *name*, permite escolher que estratégia a ser usada para percorrer a MEF da temperatura.

Dos blocos de configurações mencionados acima, todos devem ser configurados com cautela, visto que eles afetam diretamente o funcionamento do simulador, mas no bloco *temperature*, mais concretamente o atributo *abs*, requer uma especial atenção porque ele depende da modelagem da MEF.

Se a MEF modela a temperatura como um estado então o *abs* deve ser um estado (*state*), mas se ela modela a temperatura como um evento de transição, este deve ser um evento (*event*).

A configuração errada desse campo gerará comportamentos inesperados por parte do simulador.

Arquivo XML de configuração do simulador de temperaturas atmosféricas:

```
<config>
  <target name="EarthAtmosphereFSM.xml"/>

  <!--
  Max and Min temperatures accepted by the temperature sensor.
  abs is the attribute that indicates what type data structure
  represents a temperature.
  ini is the temperature in the first state of the FSM.
  Obs:
    abs possibilities:
      - event
      - state
  -->
  <temperature ini="15.04" max="16" min="-65" abs="state" />

  <!--
  This optional tag is an auxiliary element to help the simulator decide how to
     pick a temperature
  within a range. If the tester choose not to define this tag, it will be up to
     him to program a logic
  to choose this temperature, unless he doesn't want to or is not needed.
```

```
  pick possibilities:
    - random_ascending
    - constant_ascending
  -->
  <temperature_within_range pick="random_ascending" />

  <!--
  prob indicates the probability of generate a temperature not
  accepted by the temperature sensor.
  -->
  <error prob="0.1"/>

  <!--
  All strategies:
    - WalkByProbability
    - WalkByHighTemperature
  -->
  <strategy name="WalkByHighTemperature"/>
</config>
```

Como podemos observar pelo arquivo de configuração do simulador de temperaturas, o *abs* é um estado, o que está coerente com a modelagem do MEF da temperatura (seção anterior).

5.7.3 Implementação

A classe que implementa o simulador de temperaturas chama-se *TemperatureSim*. Essa classe recebe como parâmetros o número de amostras, tamanho de uma amostra e uma lista com os estados de parada.

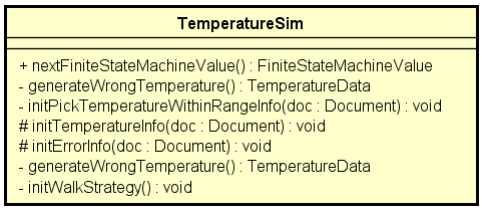

Figura 5.6: Classe *TemperatureSim*.

Pelo diagrama de classes acima, podemos constatar que essa classe implementa o método abstrato *nextFSMData*. Em outras palavras, a mesma estende da classe *Simulator*, o que era de se esperar, visto que *TemperatureSim* é um simulador da SES. Como visto no capítulo anterior, o método *nextFSMData* retorna o valor de um evento, que neste caso é uma temperatura. Para implementar esse método, foi empregado a seguinte metodologia:

1. Usar a estratégia de passeio para determinar a próxima transição a ser efetuada;
2. Se a próxima transição a ser efetuada for *null*, então ocorreu um erro na leitura do evento e deve-se gerar uma temperatura errada;
3. Caso contrário determina-se se a temperatura associada ao estado de destino da transição é um intervalo de temperaturas;

4. Em caso afirmativo, após aplicação de uma estratégia de computação de um valor pertencente ao intervalo de temperaturas, é retornado o valor de temperatura computado;

5. Caso contrário é retornado diretamente o valor da temperatura.

Empregando a metodologia acima, foi possível construir o fluxograma do método *nextFSMData* (ilustrado na figura abaixo).

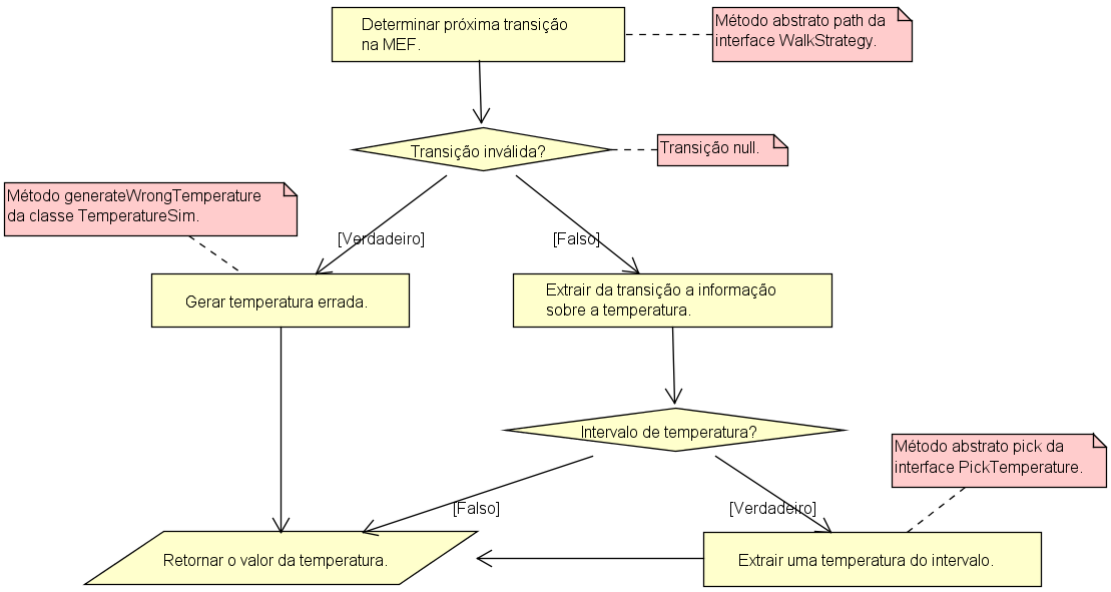

Figura 5.7: Diagrama de classes do tipo de dados temperatura.

No apêndice C é possível visualizar o código completo da classe *TemperatureSim*.

Além do método *nextFSMData*, a classe *TemperatureSim* também contém os seguintes métodos:

- *initTemperatureWithInRangeInfo* - recebe o arquivo de configurações do simulador de temperaturas e extrai a informação que permite definir que algoritmo será empregado na discretização e determinação de uma temperatura num intervalo de temperaturas;

- *initTemperatureInfo* – recebe o arquivo de configurações do simulador de temperaturas e extrai informações referentes a temperatura (temperaturas máximas, mínimas e tipo de abstração);

- *initErrorInfo* – recebe o arquivo de configurações do simulador de temperaturas e extrai a informação sobre a probabilidade de ocorrência de erros na simulação;

- *initFiniteStateMachine* – cria MEF da temperatura;

- *initWalkStrategy* – define que estratégia de passeio será empregue na MEF;

- *generateWrongTemperature* – gera temperaturas fora dos limites máximo e mínimo de uma temperatura aceitável.

5.7.4 Tipo de dados da temperatura

A temperatura pode ser um único valor real ou um intervalo de valores reais. O tipo de dados de uma temperatura é representado pela classe abstrata *TemperatureData* que tem o *isIntervalOfTemperatures* como método abstrato. Esse método testa se uma temperatura é um intervalo de temperaturas ou um único valor. As classes que implementam essa interface são a *TemperatureValue* e a *TemperatureInterval*.

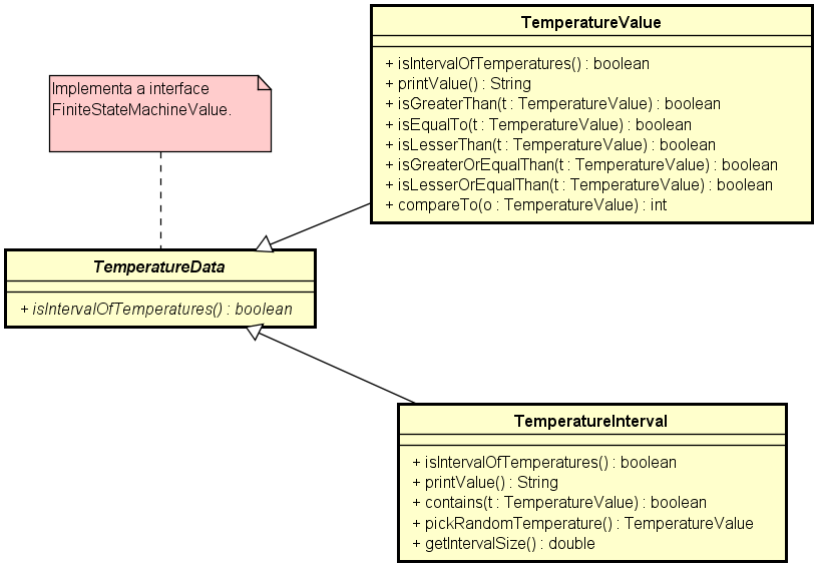

Figura 5.8: Diagrama de classes do tipo de dados temperatura.

A classe *TemperatureValue* recebe um valor real e constrói uma temperatura única. Essa classe também implementa as operações de comparação entre temperaturas:

- *isEqualTo* – compara se é igual à temperatura recebida;
- *isGreaterThan* – compara se é maior do que a temperatura recebida;
- *isLesserThan* – compara se é menor do que a temperatura recebida;
- *isGreaterOrEqualThan* – compara se é maior ou igual do que a temperatura recebida;
- *isLesserOrEqualThan* – compara se é menor ou igual do que a temperatura recebida.

A classe *TemperatureInterval* recebe duas temperaturas do tipo *TemperatureValue*, onde uma é maior do que a outra, e cria um intervalo de temperaturas. Além dos *getters* e *setters* a classe *TemperatureInterval* contém os seguintes métodos:

- *contains* – recebe uma temperatura e retorna verdadeiro se a temperatura pertence ao intervalo de temperaturas;
- *pickRandomTemperature* – computa uma temperatura aleatória dentro de um intervalo de temperaturas;
- *getIntervalSize* – retorna o tamanho do intervalo de temperaturas.

Para que a temperatura seja considerada um tipo de dados compatível com os dados do MEF, a classe abstrata *TemperatureData* implementa a interface *FiniteStateMachineData*.

Na seção anterior, foi mencionada a aplicação de uma estratégia para a computação de um valor de temperatura pertencente a um intervalo de temperaturas. Essas estratégias são definidas pela interface *PickTemperature*. Essa interface contém o método abstrato *pick*, que quando implementado deve retornar uma temperatura pertencente a um determinado intervalo de temperaturas.

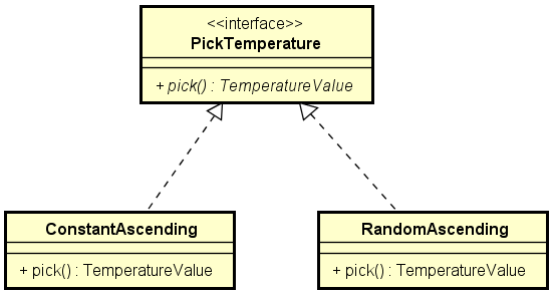

Figura 5.9: Diagrama de classes das estratégias de escolha de uma temperatura dentro de um intervalo.

A classe *ConstantAscending* recebe um intervalo de temperaturas e uma constante de ascensão. A partir dessa constante de ascensão e tendo em conta sempre o último valor de temperatura computado nesse intervalo, são computados os novos valores de temperaturas pertencentes ao intervalo. Já a classe *RandomAscending*, tal e qual o nome indica, gera um valor aleatório ascendente, em relação ao último valor aleatório gerado nesse intervalo.

Essas estratégias existem apenas para disponibilizar ao usuário de SES uma alternativa ao método *pickRandomTemperature* da classe *TemperatureInterval*, que é totalmente aleatório, e para garantir que todos os valores de temperatura de uma amostra sejam coerentes com o movimento ascendente do balão estratosférico.

5.7.5 Estratégia de passeio

Para exemplificar uma estratégia de passeio particular ao simulador de temperaturas, ou seja, uma que dependa apenas dele (ou das estruturas e dados que o compõe) foi implementado uma estratégia que consiste na maximização da próxima transição na MEF. Tendo em conta as possíveis transições do estado corrente, é escolhido a transição que maximiza o valor da temperatura a ser colocado na amostra.

Essa estratégia de passeio é implementado pela classe *WalkByHighTemperature*, que recebe a probabilidade de erro, temperaturas máximas e mínimas, e o tipo de abstração da temperatura.

5.7.6 Geração de erros

O método que implementa a geração de erros por meio da perturbação de dados, é o *generateWrongTemperature*, da classe *TemperatureSim*. Esse método simplesmente gera temperaturas além dos limites máximo e mínimo de uma temperatura correta.

A implementação do método *generateWrongTemperature* pode ser encontrada no apêndice C.

5.8 Discussões

Apesar de ter sido implementado apenas a simulação da temperatura atmosférica e da MEF da temperatura, usada nas simulações, ser bastante simplista e limitada, visto que ela modela apenas o comportamento da temperatura ao longo de um movimento ascendente na atmosfera, o funcionamento e os resultados das amostras de temperaturas gerados pelo SES foram satisfatórios (no apêndice D são apresentados os resultados de várias simulações feitas para diferentes configurações do SES).

Após a capacidade de gerar simulações de eventos, o maior feito deste projeto foi implementar, de forma razoavelmente simples, a escalabilidade do SES no quesito de implementação e adição de simuladores para novos eventos.

Capítulo 6

Considerações finais

Aqui serão apresentadas as considerações finais e globais deste trabalho.

A primeira seção destina-se a apresentação das contribuições e conclusões obtidas a partir deste trabalho, e a segunda seção apresenta os possíveis trabalhos futuros a serem desenvolvidos.

6.1 Contribuições e conclusões

As contribuições deste trabalho são:

- Implementação de um simulador de eventos modelados em *Statecharts*;
- Geração automatizada dos casos de teste;
- Utilização de estratégias de passeio sobre a MEF para gerar os casos de testes;
- Uma arquitetura escalável a vários tipos de simulações e estratégias de passeio sobre a MEF.

E permitem tirar as seguintes conclusões:

- Comparado com a simulação aleatória de eventos, a modelagem dos mesmos por meio de *Statecharts* e por conseguinte MEF, oferece um maior controle sobre o casos de teste a serem gerados, visto que as geração dos mesmos vai depender da estratégia de varredura empregada na MEF e da própria MEF;
- Escalabilidade fácil e prática, visto que dependem de implementações de interfaces ou extensão de classes abstratas.

6.2 Trabalhos futuros

Com a "conclusão" deste trabalho, abrem-se portas para muitos trabalhos futuros. Eis alguns dos trabalhos que podem ser desenvolvidos:

- Implementação e inclusão de novos tipos de simulações;
- Implementação e inclusão de novas estratégias de passeio sobre a MEF;

- Implementação de testes unitários e maximização da cobertura do código;
- Usar SES para testar o SWPDC;
- Fazer uma análise e estudo comparativo com outros métodos de geração automática de casos de teste;
- Implementar um construtor de MEF a partir da discretização de imagens de funções (contínuas) que modelam eventos.

Capítulo 7

Referências Bibliográficas

[1] ACHUTTI, Camila. *Verificação e Validação de Softwares de Missão Crítica*. Monografia - Universidade de São Paulo, Instituto de Matemática e Estatística, São Paulo, 2013. Disponível em: <http://bcc.ime.usp.br/principal/tccs/2013/camila/monografia.pdf>. Acesso em: 18 de out 2015.

[2] WASOWSKI, Andrzej. *Modeling Reactive Systems with IAR visualSTATE Statecharts*. Disponível em: <https://www.itu.dk/∼wasowski/teach/statecharts/handout.pdf>. Acesso em: 30 de nov 2015.

[3] HAREL, David. *STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*. Science of Computer Programming, 8, 231–274, 1987.

[4] MATHUR, Aditya. *Test Generation: Statecharts*. West Lafayette, Indiana, USA: Purdue University, 2005. 65.

[5] *GTSC: Geração Automática de Casos de Teste Baseada em Statecharts, Manual do Usuário*. Disponível em: <http://www.lac.inpe.br/∼valdivino/ManualUsuario_GTSCv2.pdf>. Acesso em: 18 out 2015.

[6] *Wikipedia, the free encyclopedia: Document Object Model*. Disponível em: <https://en.wikipedia.org/wiki/Document_Object_Model>. Acesso em: 16 nov 2015.

[7] *Mkyong, How to read XML file in Java – (DOM Parser)*. Disponível em: <http://www.mkyong.com/java/how-to-read-xml-file-in-java-dom-parser/>. Acesso em: 16 nov 2015.

[8] *Mkyong, How to read XML file in Java – (SAX Parser)*. Disponível em: <http://www.mkyong.com/java/how-to-read-xml-file-in-java-sax-parser/>. Acesso em: 16 nov 2015.

[9] RODRIGUES, Diego; RODRIGUES, Diogo. *GERAÇÃO DE CASOS DE TESTES PARA PROTOCOLOS USANDO O MÉTODO ROUND-TRIP PATH*. Monografia - Faculdade de Tecnologia do Estado de São Paulo, São José dos Campos, São Paulo, 2009. Disponível em: <http://fatecsjc.edu.br/trabalhos-de-graduacao/wp-content/uploads/2012/03/BDR1_diego_diogo2009.pdf >. Acesso em: 30 de nov 2015.

[10] AMARAL, A. S. M. S.; VELOSO, R. R.;VIJAYKUMAR, N. L.;FRANCES, C. R. L.; OLIVEIRA, Edvar.
On proposing a Markup Language for Statecharts to be used in Performance Evaluation.
International Journal of Computational Intelligence, 1-7, 2004.

[11] *Glenn Research Center, Nasa: Earth Atmosphere Model*.
Disponível em: <https://www.grc.nasa.gov/www/k-12/airplane/atmosmet.html>.
Acesso em: 11 nov 2015.

Parte II

Parte subjetiva

Capítulo 8

Aprendizado

8.1 Desafios e frustrações

8.1.1 Frustrações

Pela minha felicidade, as frustrações ao longo deste trabalho foram poucas:

- Devido as limitações do GTSC, não foi possível usá-lo para automatizar a geração da MEF da temperatura;
- Como neste trabalho a função para a computação da temperatura na atmosfera terrestre era contínua, seria útil a implementação de um discretizador de imagens de funções contínuas que descrevem eventos, isso facilitaria na criação das MEF dos eventos.

8.1.2 Desafios

Os principais desafios que encontrei ao longo do desenvolvimento deste trabalho foram:

- Desenvolver uma forma para a modelagem da temperatura em MEF;
- Garantir a máxima escalabilidade ao simulador, nos quesitos simulações e configurações.

8.2 Disciplinas cursadas relevantes ao desenvolvimento do TCC

As principais disciplinas do curso do BCC que julgo terem me fornecido o conhecimentos necessários para produzir este trabalho foram:

- MAC0110 (Introdução à Computação): Primeira experiência com programação JAVA.
- MAC0323 (Estruturas de Dados): Os conceitos de estruturas de dados são sempre aplicados na construção de softwares, por isso, é muito importante conhece-las e saber quais suas vantagens e desvantagens;
- MAC0338 (Análise de Algoritmos): Forneceu os meios para construir algoritmos eficientes.
- MAC0441 (Programação Orientada a Objetos): Nessa disciplina teve um projeto legado de grande porte, onde foi possível aplicar conceitos como padrões de projeto.

- MAC0242 (Laboratório de Programação II), MAC0332 (Engenharia de Software), MAC0340 (Laboratório de Engenharia de Software) e MAC0342 (Laboratório de Programação eXtrema): Os projetos nessas disciplinas foram todos de grande porte e usando a linguagem de programação JAVA, ou seja, foi possível sedimentar e aprimorar os conhecimentos em JAVA previamente adquiridos e amadurecer como programador.

8.3 Próximos passos

No que diz respeito ao meu trabalho de conclusão de curso, pretendo aperfeiçoar o simulador já implementado e expandí-lo com implementações de novas simulações de eventos, e quem sabe publicar um artigo científico, expandir os meus conhecimentos em ciência da computação, principalmente na área de engenharia de software, por meio de um mestrado, e finalmente, concluir o BCC no próximo ano.

Capítulo 9

Agradecimentos

Primeiramente, agradeço a Deus por me ter dado ânimo e capacidade para concluir este trabalho, de seguida a minha família, pelo apoio incondicional e muita paciência, a minha orientadora e ao meu tutor pelos conselhos, a todos os meus amigos e colegas do BCC e do IME, e por fim, mas não menos relevante, a todos os que foram meus professores ao longo do curso.

Parte III

Anexos

Apêndice A

Implementação da classe *TemperatureParser*

Atributos e constantes:

```
private final String TAG_STATES = "STATES";
private final String TAG_STATE = "STATE";
private final String TAG_STATE_ATTRIBUTE_NAME = "NAME";
private final String TAG_STATE_ATTRIBUTE_TYPE = "TYPE";
private final String TAG_INPUTS = "INPUTS";
private final String TAG_INPUT = "INPUT";
private final String TAG_INPUT_ATTRIBUTE_EVENT = "EVENT";
private final String TAG_TRANSITIONS = "TRANSITIONS";
private final String TAG_TRANSITION = "TRANSITION";
private final String TAG_TRANSITION_ATTRIBUTE_SOURCE = "SOURCE";
private final String TAG_TRANSITION_ATTRIBUTE_DESTINATION = "DESTINATION";
private TemperatureValueBuilder temperatureValueBuilder;
private String temperatureAbstraction;
```

Construtor:

```
public TemperatureParser(File xmlFile, String temperatureAbstraction) {
  initParser(xmlFile);
  this.temperatureAbstraction = temperatureAbstraction;
  temperatureValueBuilder = TemperatureValueBuilder.getInstance();
  statesMap = findAllStates();
  eventsMap = findAllEvents();
}
```

Implementação do método *getAllTransitions*:

```
@Override
public ArrayList<FiniteStateMachineTransition> getAllTransitions() {
FiniteStateMachineTransition auxTransition;
ArrayList<FiniteStateMachineTransition> transitionsList = new
    ArrayList<FiniteStateMachineTransition>();
Node node = doc.getElementsByTagName(TAG_TRANSITIONS).item(0);
  if (node.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) node;
    NodeList list = element.getElementsByTagName(TAG_TRANSITION);
    for (int i = 0; i < list.getLength(); i++) {
      Node nodeAux = list.item(i);
      if (nodeAux.getNodeType() == Node.ELEMENT_NODE) {
```

```
        Element elementAux = (Element) nodeAux;
        String originStateName =
            elementAux.getAttribute(TAG_TRANSITION_ATTRIBUTE_SOURCE);
        String destinStateName =
            elementAux.getAttribute(TAG_TRANSITION_ATTRIBUTE_DESTINATION);
        String inputEventName =
            elementAux.getElementsByTagName(TAG_INPUT).item(0).getTextContent();
        auxTransition = new FiniteStateMachineTransition(
              (State)statesMap.get(originStateName),
              (State)statesMap.get(destinStateName),
              (Event)eventsMap.get(inputEventName));
        transitionsList.add(auxTransition);
      }
    }
  }
  return transitionsList;
}
```

Implementação do método *findAllStates*:

```
@Override
protected Map<String, State> findAllStates() {
  State auxState;
  Map<String, State> statesMap = new HashMap<>();
  Node node = doc.getElementsByTagName(TAG_STATES).item(0);
  if (node.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) node;
    NodeList list = element.getElementsByTagName(TAG_STATE);
    for (int i = 0; i < list.getLength(); i++) {
      Node nodeAux = list.item(i);
      if (nodeAux.getNodeType() == Node.ELEMENT_NODE) {
        Element elementAux = (Element) nodeAux;
        if (!statesMap.containsKey(
          elementAux.getAttribute(TAG_STATE_ATTRIBUTE_NAME))) {
          if (temperatureAbstraction.equals("state")) {
            auxState = new State(
              temperatureValueBuilder.buildFromString(
                elementAux.getAttribute(TAG_STATE_ATTRIBUTE_NAME)));
          } else {
            auxState = new State(
              new FiniteStateMachineDefaultData(elementAux.getAttribute(
                TAG_STATE_ATTRIBUTE_NAME)));
          }
          statesMap.put(elementAux.getAttribute(TAG_STATE_ATTRIBUTE_NAME),
              auxState);
        } else {
          auxState = (State)
              statesMap.get(elementAux.getAttribute(TAG_STATE_ATTRIBUTE_NAME));
        }
        if
            (elementAux.getAttribute(TAG_STATE_ATTRIBUTE_TYPE).equals("inicial"))
            {
          auxState.setInitialState(true);
        } else if
```

```
            (elementAux.getAttribute(TAG_STATE_ATTRIBUTE_TYPE).equals("final")) {
          auxState.setFinalState(true);
        }
      }
    }
  }
  return statesMap;
}
```

Implementação do método *findAllEvents*:

```
@Override
protected Map<String, Event> findAllEvents() {
  Event auxEvent;
  Map<String, Event> eventsMap = new HashMap<>();
  Node node = doc.getElementsByTagName(TAG_INPUTS).item(0);
  if (node.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) node;
    NodeList list = element.getElementsByTagName(TAG_INPUT);
    for (int i = 0; i < list.getLength(); i++) {
      Node auxNode = list.item(i);
      if (auxNode.getNodeType() == Node.ELEMENT_NODE) {
        Element auxElement = (Element) auxNode;
        if (temperatureAbstraction.equals("event")) {
          auxEvent = new Event(
            temperatureValueBuilder.buildFromString(
              auxElement.getAttribute(TAG_INPUT_ATTRIBUTE_EVENT)));
        } else {
          auxEvent = new Event(
            new FiniteStateMachineDefaultData(
              auxElement.getAttribute(TAG_INPUT_ATTRIBUTE_EVENT)));
        }
        eventsMap.put(auxElement.getAttribute(TAG_INPUT_ATTRIBUTE_EVENT),
           auxEvent);
      }
    }
  }
  return eventsMap;
}
```

Apêndice B

XML da MEF da temperatura

```
<MFEE>
  <STATES>
    <STATE NAME="15.04" TYPE="inicial"/>
    <STATE NAME="8.55" TYPE="normal"/>
    <STATE NAME="-17.41" TYPE="normal"/>
    <STATE NAME="-4.43" TYPE="normal"/>
    <STATE NAME="2.06" TYPE="normal"/>
    <STATE NAME="-23.90" TYPE="normal"/>
    <STATE NAME="-33.63" TYPE="normal"/>
    <STATE NAME="[-55.00,-45.00]" TYPE="normal"/>

    <STATE NAME="-56.46" TYPE="final"/>

    <STATE NAME="-50.48" TYPE="normal"/>
    <STATE NAME="-41.51" TYPE="normal"/>
    <STATE NAME="-45.99" TYPE="normal"/>
    <STATE NAME="-38.52" TYPE="normal"/>
    <STATE NAME="-16.09" TYPE="normal"/>
    <STATE NAME="-20.58" TYPE="normal"/>
    <STATE NAME="[-10.00,0.00]" TYPE="final"/>
  </STATES>
  <EVENTS>
    <EVENT VALUE="1" NAME="5000"/>
    <EVENT VALUE="1" NAME="1000"/>
    <EVENT VALUE="1" NAME="2000"/>
    <EVENT VALUE="1" NAME="3000"/>
    <EVENT VALUE="1" NAME="6000"/>
    <EVENT VALUE="1" NAME="7500"/>
    <EVENT VALUE="1" NAME="10000"/>

    <EVENT VALUE="1" NAME="[11000,25000]"/>

    <EVENT VALUE="1" NAME="27000"/>
    <EVENT VALUE="1" NAME="30000"/>
    <EVENT VALUE="1" NAME="28500"/>
    <EVENT VALUE="1" NAME="38500"/>
    <EVENT VALUE="1" NAME="31000"/>
    <EVENT VALUE="1" NAME="40000"/>
    <EVENT VALUE="1" NAME="37000"/>
```

```
    <EVENT VALUE="1" NAME="41000"/>
    <EVENT VALUE="1" NAME="40230"/>
    <EVENT VALUE="1" NAME="41700"/>
  </EVENTS>
  <INPUTS>
    <INPUT EVENT="5000"/>
    <INPUT EVENT="1000"/>
    <INPUT EVENT="2000"/>
    <INPUT EVENT="3000"/>
    <INPUT EVENT="6000"/>
    <INPUT EVENT="7500"/>
    <INPUT EVENT="10000"/>

    <INPUT EVENT="[11000,25000]"/>

    <INPUT EVENT="27000"/>
    <INPUT EVENT="30000"/>
    <INPUT EVENT="28500"/>
    <INPUT EVENT="38500"/>
    <INPUT EVENT="31000"/>
    <INPUT EVENT="40000"/>
    <INPUT EVENT="37000"/>
    <INPUT EVENT="41000"/>
    <INPUT EVENT="40230"/>
    <INPUT EVENT="41700"/>
  </INPUTS>
  <OUTPUTS>
  </OUTPUTS>
  <TRANSITIONS>
    <TRANSITION SOURCE="15.04" DESTINATION="-17.41">
      <INPUT INTERFACE="L">5000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="15.04" DESTINATION="8.55">
      <INPUT INTERFACE="L">1000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-17.41" DESTINATION="-23.90">
      <INPUT INTERFACE="L">6000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="8.55" DESTINATION="2.06">
      <INPUT INTERFACE="L">2000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="8.55" DESTINATION="-4.43">
      <INPUT INTERFACE="L">3000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-4.43" DESTINATION="-23.90">
      <INPUT INTERFACE="L">6000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-4.43" DESTINATION="-33.63">
      <INPUT INTERFACE="L">7500</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-23.90" DESTINATION="-33.63">
      <INPUT INTERFACE="L">7500</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-33.63" DESTINATION="[-55.00,-45.00]">
```

```
      <INPUT INTERFACE="L">10000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-23.90" DESTINATION="[-55.00,-45.00]">
      <INPUT INTERFACE="L">10000</INPUT>
    </TRANSITION>

    <TRANSITION SOURCE="[-55.00,-45.00]" DESTINATION="-56.46">
      <INPUT INTERFACE="L">[11000,25000]</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-56.46" DESTINATION="-56.46">
      <INPUT INTERFACE="L">[11000,25000]</INPUT>
    </TRANSITION>

    <TRANSITION SOURCE="-56.46" DESTINATION="-50.48">
      <INPUT INTERFACE="L">27000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-56.46" DESTINATION="-41.51">
      <INPUT INTERFACE="L">30000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-56.46" DESTINATION="-45.99">
      <INPUT INTERFACE="L">28500</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-50.48" DESTINATION="-16.09">
      <INPUT INTERFACE="L">38500</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-50.48" DESTINATION="-38.52">
      <INPUT INTERFACE="L">31000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-41.51" DESTINATION="-38.52">
      <INPUT INTERFACE="L">31000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-45.99" DESTINATION="[-10.00,0.00]">
      <INPUT INTERFACE="L">40000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-16.09" DESTINATION="[-10.00,0.00]">
      <INPUT INTERFACE="L">41000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-38.52" DESTINATION="[-10.00,0.00]">
      <INPUT INTERFACE="L">40230</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-20.58" DESTINATION="[-10.00,0.00]">
      <INPUT INTERFACE="L">41700</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-38.52" DESTINATION="-20.58">
      <INPUT INTERFACE="L">37000</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="2.06" DESTINATION="-33.63">
      <INPUT INTERFACE="L">7500</INPUT>
    </TRANSITION>
    <TRANSITION SOURCE="-33.63" DESTINATION="-50.48">
      <INPUT INTERFACE="L">27000</INPUT>
    </TRANSITION>
  </TRANSITIONS>
</MFEE>
```

Apêndice C

Implementação da classe *TemperatureSim*

Atributos e constantes:

```
private double maxTemperature;
private double minTemperature;

private double probabilityOfError;

private String pickTemperatureStrategy;

Stack<FiniteStateMachineTransition> processedTransitions;
Stack<String> processedValues;

static PickTemperature pickTemperature; // temporary
```

Construtor:

```
public TemperatureSim(int numOfSamples, int sizeOfSample, ArrayList<String>
    stopFinalStates) {
  super.numOfSamples = numOfSamples;
  super.sizeOfSample = sizeOfSample;
  super.stopFinalStatesNames = stopFinalStates;
  super.stopAtFinalStates = (stopFinalStates == null) ? false : true;

  XMLReader xmlReader = new
     XMLReader("src/br/usp/ime/bcc/mac499/ses/sims/temperature/config.xml");
  Document doc = xmlReader.getDocument();
  initTargetInfo(doc);
  initTemperatureInfo(doc);
  initErrorInfo(doc);
  initStrategyInfo(doc);
  initPickTemperatureWithinRangeInfo(doc);

  super.targetFile = new File("targets/" + targetName);

  initFiniteStateMachine();
  initWalkStrategy();

  processedTransitions = new Stack<FiniteStateMachineTransition>();
  processedValues = new Stack<String>();
  processedValues.push((new Double(super.initialValue)).toString());
```

```
}
```

Implementação dos métodos auxíliares de inicialização:

```
private void initPickTemperatureWithinRangeInfo(Document doc) {
  Node node = doc.getElementsByTagName("temperature_within_range").item(0);
  if (node != null) {
    if (node.getNodeType() == Node.ELEMENT_NODE) {
      Element element = (Element) node;
      String attributeValue = element.getAttribute("pick");
      switch (attributeValue) {
        case "random_ascending":
        case "constant_ascending":
          pickTemperatureStrategy = attributeValue;
          break;
        default:
          throw new PickTemperatureException("Unknown pick temperature
              strategy.");
      }
    }
  }
}

protected void initTemperatureInfo(Document doc) {
  Node node = doc.getElementsByTagName("temperature").item(0);
  if (node.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) node;
    maxTemperature = Double.parseDouble(element.getAttribute("max"));
    minTemperature = Double.parseDouble(element.getAttribute("min"));
    super.eventAbstraction = element.getAttribute("abs");
    super.initialValue = Double.parseDouble(element.getAttribute("ini"));
  }
}

protected void initErrorInfo(Document doc) {
  Node node = doc.getElementsByTagName("error").item(0);
  if (node.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) node;
    probabilityOfError = Float.parseFloat(element.getAttribute("prob"));
  }
}

private void initFiniteStateMachine() {
  FiniteStateMachineParser fsmParser = new TemperatureParser(targetFile,
      super.eventAbstraction);
  finiteStateMachine = new FiniteStateMachine(fsmParser.getAllStates(),
      fsmParser.getAllEvents(),
                        fsmParser.getAllTransitions());
}

private void initWalkStrategy() {
  switch (strategyName) {
    case "WalkByHighTemperature":
      walkStrategy = new WalkByHighTemperature(probabilityOfError,
```

```
            super.eventAbstraction);
        break;
      case "WalkByProbability":
        walkStrategy = new WalkByProbability(probabilityOfError);
        break;
      default:
        throw new WalkStrategyException("Unknown " + strategyName + " walk
            strategy.");
    }
}
```

Implementação do método *generateWrongTemperature*:

```
private TemperatureData generateWrongTemperature() {
  double lowestTemperature = -143.00, highestTemperature = 56.70;
  double value, range, min, max;
  double random = Math.random();
  if (random > 0.5) {
    max = highestTemperature;
    min = maxTemperature;
  } else {
    max = minTemperature;
    min = lowestTemperature;
  }
  range = Math.abs(max - min);
  value = max - (Math.random() * range);
  return new TemperatureValue(value);
}
```

Implementação do método abstrato *nextFSMData*:

```
@Override
protected FiniteStateMachineData nextFSMData() {
  TemperatureData temperature;
  FiniteStateMachineTransition pickedTransition =
      walkStrategy.path(finiteStateMachine.getCurrentState());
  if (pickedTransition == null) {
    System.out.print(" w:"); // temporary
    return generateWrongTemperature();
  } else {
    if (super.eventAbstraction.equals("state")) {
      temperature = (TemperatureData) pickedTransition.getDestin().getData();
    } else {
      temperature = (TemperatureData) pickedTransition.getTrigger().getData();
    }
    if (temperature.isIntervalOfTemperatures()) {
      if (pickTemperatureStrategy != null) {
        if (!pickedTransition.equals(processedTransitions.peek())) {
          switch (pickTemperatureStrategy) {
            case "random_ascending":
              pickTemperature = new
                  RandomAscending((TemperatureInterval)temperature);
              break;
            default:
```

```
                pickTemperature = new
                    ConstantAscending((TemperatureInterval)temperature,1);
                break;
            }
          }
          temperature = pickTemperature.pick();
        } else {
          temperature =
              ((TemperatureInterval)temperature).pickRandomTemperature();
        }
      }
      finiteStateMachine.applyTransition(pickedTransition);
      processedTransitions.push(pickedTransition);
      processedValues.push(temperature.printData());
      return temperature;
    }
}
```

Apêndice D

Amostras de temperaturas geradas pelo SES

Neste apêndice serão apresentadas algumas amostras de temperaturas geradas pelo SES, tendo em conta algumas combinações de configurações.
Os valores de temperatura que precedem o indicador *w*, são as temperaturas erradas da amostra.

D.1 Execução 1

Algumas configurações do simulador no momento da geração da amostra:

- estratégia de passeio: *WalkByProbability*;
- estratégia de escolha de uma temperatura num intervalo: *random_ascending*;
- condição de parada de uma amostra: tamanho da amostra.

Amostras geradas:

```
Sample 1:
 15.04 -17.41 -23.90 -45.85 -56.46 -41.51 -38.52 -20.58 -1.98 -0.56 -0.06 -0.02
    0.00
Sample 2:
 15.04 8.55 2.06 -33.63 -54.91 -56.46 -56.46 -41.51 w: -138.42 -38.52 -7.66
    -4.34 -2.19
Sample 3:
 15.04 -17.41 -23.90 -46.01 w: 47.38 -56.46 -45.99 -6.74 -4.08 -0.04 -0.01 0.00
    0.00
Sample 4:
 15.04 8.55 2.06 -33.63 -54.36 -56.46 -56.46 -56.46 -56.46 -45.99 -5.08 -0.56
    -0.19
Sample 5:
 15.04 8.55 2.06 w: -113.23 -33.63 -52.76 -56.46 -45.99 -1.26 -1.02 -0.17 -0.09
    -0.01
```

D.2 Execução 2

Algumas configurações do simulador no momento da geração da amostra:

- estratégia de passeio: *WalkByProbability*;

- estratégia de escolha de uma temperatura num intervalo: *random_ascending*;
- condição de parada de uma amostra: estado de parada.

Amostras geradas:

```
Sample 1:
 15.04 w: 47.41 8.55 2.06 -33.63 -54.82 -56.46
Sample 2:
 15.04 8.55 -4.43 -33.63 w: 52.97 -50.48 -38.52 -20.58 -2.12
Sample 3:
 15.04 w: -69.58 -17.41 -23.90 -33.63 -54.59 -56.46
Sample 4:
 15.04 8.55 2.06 -33.63 w: -134.13 -47.48 -56.46
Sample 5:
 15.04 -17.41 -23.90 -33.63 w: -99.14 -53.43 -56.46
```

D.3 Execução 3

Algumas configurações do simulador no momento da geração da amostra:

- estratégia de passeio: *WalkByProbability*;
- estratégia de escolha de uma temperatura num intervalo: *constant_ascending*;
- condição de parada de uma amostra: tamanho da amostra.

Amostras geradas:

```
Sample 1:
 15.04 w: 33.49 8.55 2.06 -33.63 -50.48 -38.52 -5.90 -4.90 -3.90 -2.90 -1.90
    -0.90
Sample 2:
 15.04 -17.41 -23.90 -33.63 -46.00 -56.46 -56.46 -56.46 -45.99 -2.30 -1.30
    -0.30 0.00
Sample 3:
 15.04 8.55 2.06 -33.63 -50.48 -16.09 -6.57 -5.57 -4.57 -3.57 -2.57 -1.57 -0.57
Sample 4:
 15.04 8.55 -4.43 -23.90 -33.63 -50.48 -16.09 -3.60 -2.60 -1.60 -0.60 0.00 0.00
Sample 5:
 15.04 -17.41 -23.90 -54.94 -56.46 -56.46 -41.51 -38.52 -20.58 -6.00 -5.00
    -4.00 -3.00
```

D.4 Execução 4

Algumas configurações do simulador no momento da geração da amostra:

- estratégia de passeio: *WalkByHighTemperature*;
- estratégia de escolha de uma temperatura num intervalo: *random_ascending*;
- condição de parada de uma amostra: tamanho da amostra.

Amostras geradas:

```
Sample 1:
 15.04 w: 47.09 8.55 2.06 -33.63 -50.48 -16.09 -6.28 -3.93 -1.92 -1.28 -0.69
    -0.03
Sample 2:
 15.04 8.55 2.06 -33.63 w: -108.52 -50.48 -16.09 -8.01 -3.37 -1.79 -0.88 -0.67
    -0.36
Sample 3:
 15.04 w: 52.74 8.55 2.06 -33.63 -50.48 -16.09 -8.17 -4.21 -2.26 -1.64 -0.01
    0.00
Sample 4:
 15.04 8.55 2.06 -33.63 -50.48 -16.09 -1.69 -0.62 -0.32 -0.24 -0.11 -0.05 -0.04
Sample 5:
 15.04 w: 43.10 8.55 2.06 -33.63 -50.48 -16.09 -9.90 -9.00 -2.46 -2.15 -1.91
    -0.35
```

Apêndice E

Fluxograma do SES

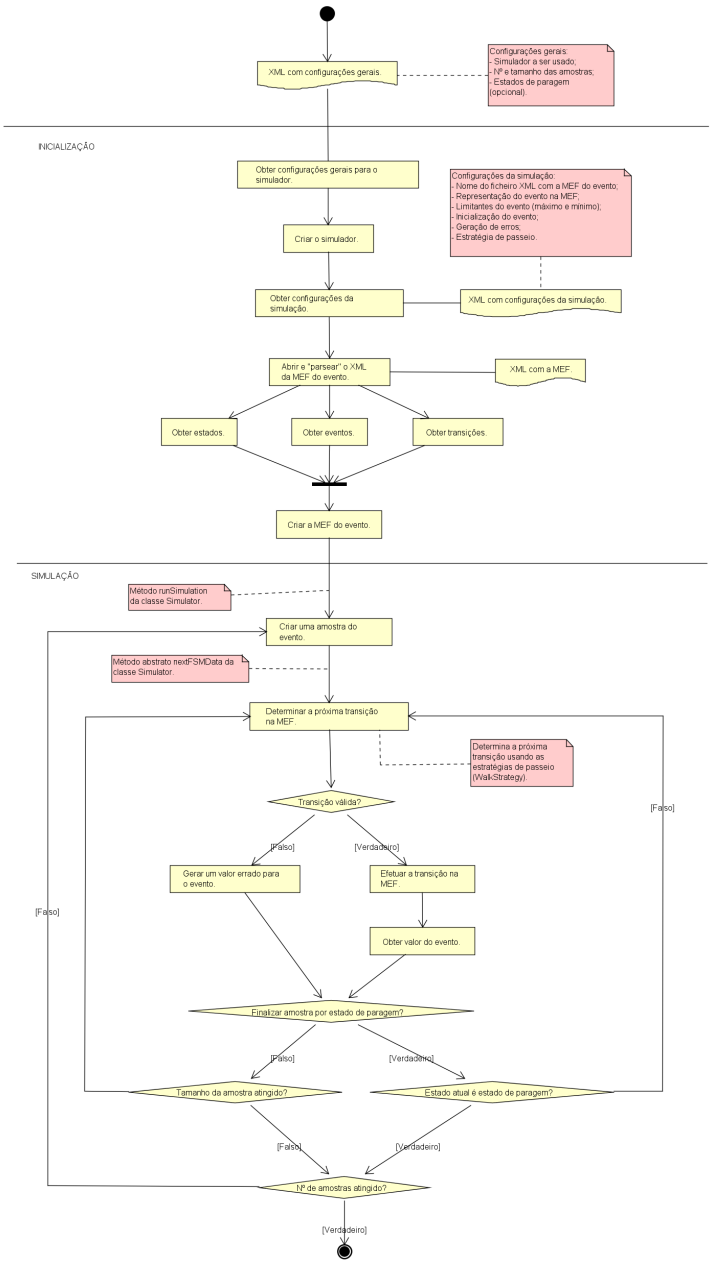

Figura E.1: Fluxograma geral do SES

Apêndice F

Diagramas das principais classes do SES

F.1 Diagrama do simulador abstrato

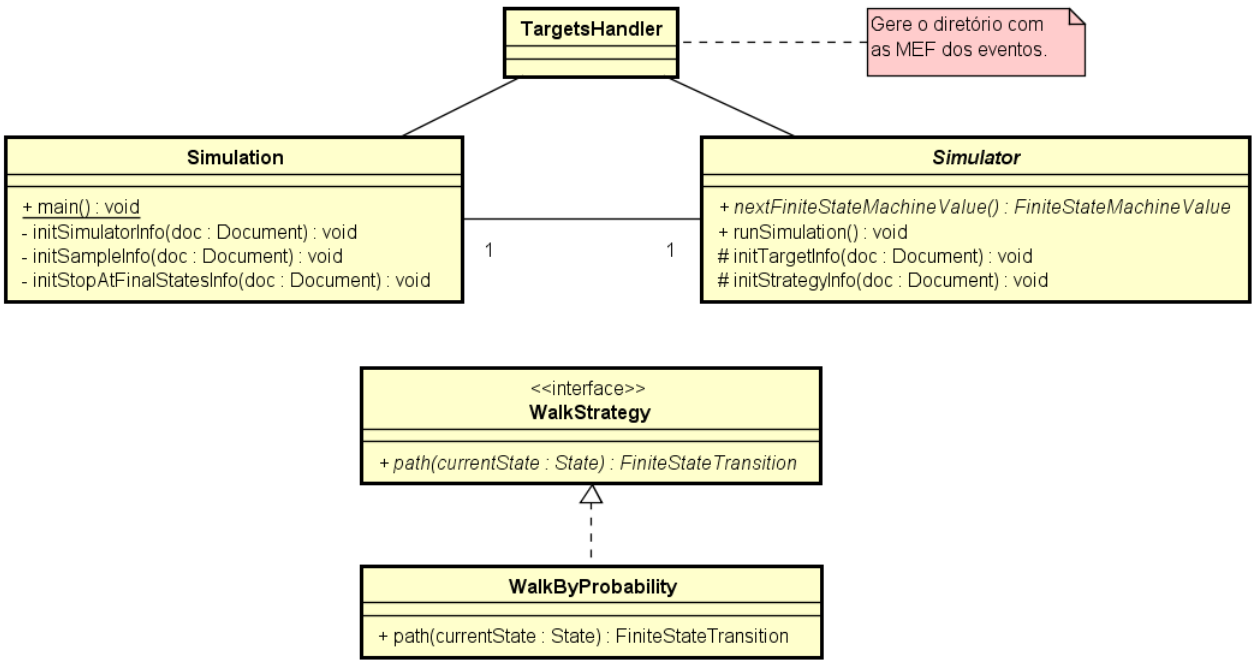

Figura F.1: Simulador abstrato

F.2 Diagrama do simulador de temperaturas atmosféricas

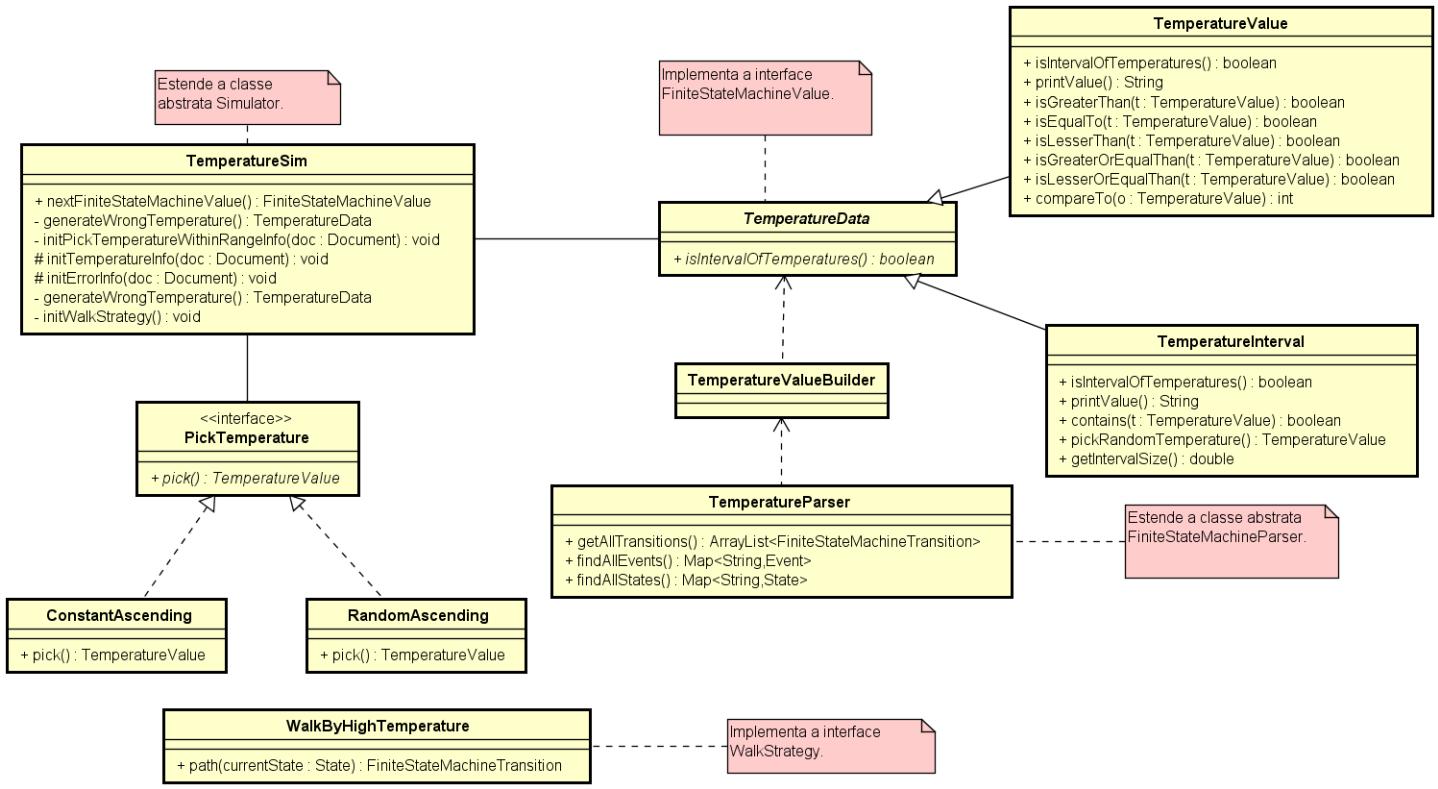

Figura F.2: Simulador de temperaturas atmosféricas

Apêndice G

Extras

G.1 Ambiente de desenvolvimento

- Sistema operacional Windows 10 (64 bits):
  https://www.microsoft.com/pt-br/software-download/windows10
- Liguagem de programação JAVA (versão 8):
  http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html
- Eclipse Mars (IDE):
  https://eclipse.org/
- Software GTSC:
  http://www.lac.inpe.br/∼valdivino/ManualUsuario_GTSCv2.pdf

G.2 Links

- Página do TCC:
  https://linux.ime.usp.br/ miranda/mac0499/
- Repositório do Simulador de Eventos modelados em *Statecharts*:
  https://github.com/MirStation/Simulador-de-Eventos-modelados-em-Statecharts-SES-

G.3 Pacotes e pastas no projeto

Principais pacotes e pastas do projeto:

- pasta *target* - onde arquivos XML com a MEF são guardados;
- pacote *ses/parser* - onde o parser está implementado;
- pacote *ses/fsm* - onde a MEF está implementada;
- pacote *ses/sim* - onde o simulador abstrato está implementado;
- pacote *ses/sims* - onde os simuladores específicos estão implementados.

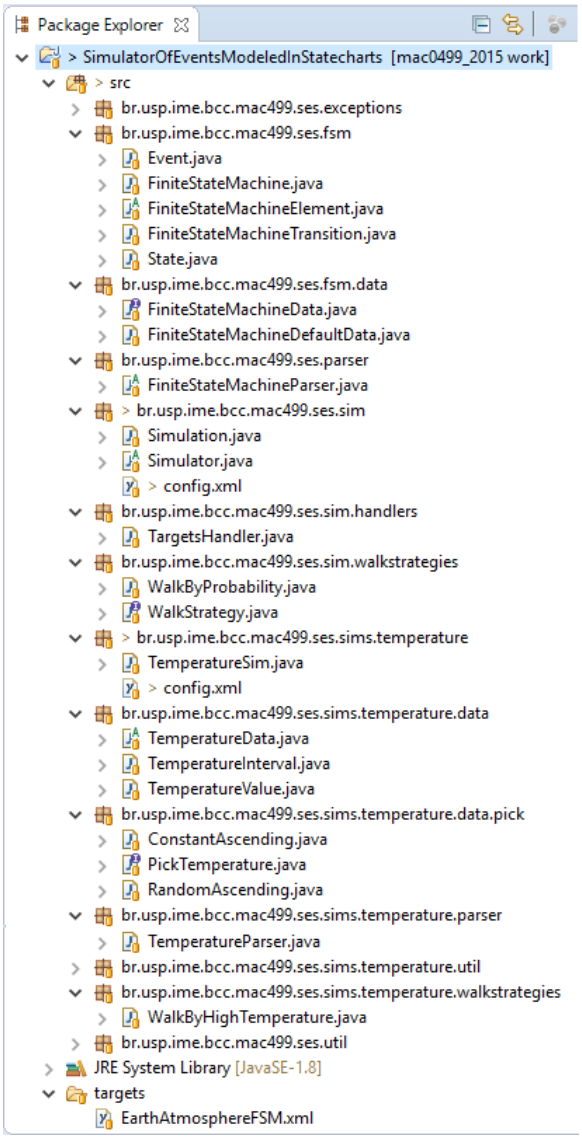

Figura G.1: Estrutura de pacotes do projeto.

G.4 Instalação e uso

- Instalar o JAVA, de preferência a versão 8;
- Baixar e instalar o eclipse;
- Baixar o código-fonte do simulador a partir do repositório Github;
- Importar o código-fonte do projeto no eclipse;
- Usar os arquivos *ses/sim/config.xml* e *ses/sims/temperature/config.xml* para configurar a simulação;
- Rodar o projeto no eclipse normalmente.