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

piStar-GODA: Integração entre os projetos piStar e GODA

Leandro Santos Bergmann

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2018

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

piStar-GODA: Integração entre os projetos piStar e GODA

Leandro Santos Bergmann

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Genaína Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. João Henrique Correia Pimentel
UFRPE

Prof.a Dr.a Cláudia de Oliveira Melo
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 15 de março de 2018

Dedicatória

Dedico aos meus pais pelo apoio.

Agradecimentos

Agradeço aos meus pais pelo apoio. Agradeço à minha orientadora, Prof.a Dr.a Genaína Nunes Rodrigues, por ter aceito o desafio e ter tido paciência para chegar até o final deste trabalho.

Resumo

O framework GODA (Goal-Oriented Dependability Analysis) é utilizado para realizar a análise de dependabilidade de modelos orientados a objetivos. Este framework estende as funcionalidades do plugin Eclipse denominado TAOM4E. Esse plugin fornece uma interface gráfica para a modelagem de modelos orientados a objetivos baseada na metodologia de desenvolvimento Tropos. No entanto, o GODA apresenta um alto acoplamento com o plugin TAOM4E, o que dificulta sua manutenibilidade e configurabilidade por depender de versões específicas da ferramenta Eclipse e do JDK 1.8. Isso dificulta não apenas o uso do GODA por terceiros, como também a evolução do mesmo. O projeto piStar-GODA é uma nova solução para estes problemas, permitindo a utilização do projeto piStar como *frontend* para a modelagem dos modelos orientados a objetivos, substituindo o TAOM4E. Buscou-se desacoplar também o backend da solução, que é responsável pela análise de dependabilidade em si, focando em uma alta coesão e baixo acoplamento entre os módulos do sistema. Desta forma, é possível implementar novas funcionalidades no backend, como novos módulos de análise no GODA de forma modularizada e independente do frontend. A solução foi construída com ferramentas modernas, como uso de microserviços para desacoplar módulos de análise do GODA dependente de ferramentas externas como o PRISM e o PARAM, e testada com testes unitários.

Palavras-chave: GODA, CRGMToPRISM, piStar

Abstract

The GODA (Goal-Oriented Dependability Analysis) framework is used to do dependability analysis of goal-oriented models. This framework extends the functionalities of the Eclipse plugin denominated TAOM4E. This plugin provides a graphic interface for modeling goal-oriented models based on the Tropos development methodology. However, the GODA project is highly coupled with the TAOM4E plugin, which makes its maintainability and configurability harder because it depends on specific versions of the Eclipse tool and JDK 1.8. This makes harder not only the use of GODA by third parties, but also its evolution. The project piStar-GODA is a new solution for these problems, allowing the usage of the piStar project as frontend for the modeling of goal-oriented models, replacing TAOM4E. It was also intended to separate the backend of the solution, which is responsible for the dependability analysis, focusing on high cohesion and low coupling between the system modules. This way, it's possible to implement new functionalities on the backend, like new analysis modules on the GODA in a modularized way and independently from the frontend. The solution was built with modern tools, using microservices to uncouple GODA's analysis modules from external tools like PRISM and PARAM, and tested with unit tests.

Keywords: GODA, CRGMToPRISM, piStar

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Problema e Hipótese 4

1.3 Objetivos 4

1.3.1 Objetivo Geral 4

1.3.2 Objetivos Específicos 4

1.4 Descrição dos Capítulos 5

2 Referencial Teórico **6**

2.1 Análise de Dependabilidade Orientada a Objetivos 6

2.1.1 Modelagem Orientada a Objetivos e Contexto 7

2.1.2 Arquitetura de Implementação do GODA 7

2.2 Geração Automática de Código DTMC 8

2.3 piStar 12

2.4 Princípios Arquiteturais Adotados 16

2.4.1 SOLID 16

2.4.2 Arquitetura Orientada a Microsserviços 17

3 Proposta **20**

3.1 GODA as a Service 20

3.1.1 Implementação 21

3.1.2 Dependências 23

3.1.3 Single Responsibility Principle - SRP 25

3.1.4 Pontos de Acesso 25

3.2 Execução do piStar-GODA 32

3.3 Execução independente 35

4 Resultados Obtidos **38**

4.1 Setup dos Testes 38

4.2 Resultados 40

4.3 Lições Aprendidas e Dificuldades Encontradas 40

5 Conclusão **44**

Referências **46**

Anexo **47**

I Testes Funcionais **48**

II Resultados dos Testes Funcionais **62**

Lista de Figuras

2.1 Processo do GODA. 7
2.2 Exemplo de modelo orientado a objetivo contextual. 8
2.3 Exemplo de um CRGM. 9
2.4 Visão de alto nível da arquitetura de implementação do GODA. 10
2.5 Arquitetura de implementação do gerador de DTMC. 11
2.6 Interface do piStar com um modelo real. 13
2.7 Aplicações monolíticas e microsserviços. 19

3.1 Diagrama de implementação da solução completa. 22
3.2 Diagrama de implementação do piStar model. 22
3.3 Diagrama de implementação do piStarJ model. 23
3.4 Exemplo da execução no piStar-GODA. 33
3.5 Execução do *backend* do piStar-GODA independentemente do *frontend*. . . 36
3.6 Diagrama de Sequência representando o uso do piStar-GODA. 36

4.1 Exemplo de modelo desenhado no *frontend* do piStar-GODA. 38
4.2 Tempos obtidos ao executar os testes funcionais. 43

I.1 Teste 1. 48
I.2 Teste 2. 49
I.3 Teste 3. 49
I.4 Teste 4. 49
I.5 Teste 5. 50
I.6 Teste 6. 50
I.7 Teste 7. 50
I.8 Teste 8. 51
I.9 Teste 9. 51
I.10 Teste 10. 51
I.11 Teste 11. 52
I.12 Teste 12. 52

I.13 Teste 13. 52
I.14 Teste 14. 53
I.15 Teste 15. 53
I.16 Teste 16. 53
I.17 Teste 17. 54
I.18 Teste 18. 54
I.19 Teste 19. 55
I.20 Teste 20. 55
I.21 Teste 21. 56
I.22 Teste 22. 56
I.23 Teste 23. 56
I.24 Teste 24. 57
I.25 Teste 25. 57
I.26 Teste 26. 57
I.27 Teste 27. 58
I.28 Teste 28. 58
I.29 Teste 29. 58
I.30 Teste 30. 59
I.31 Teste 31. 59
I.32 Teste 32. 59
I.33 Teste 33. 60
I.34 Teste 34. 60
I.35 Teste 35. 61

II.1 Resultado do caso de teste 1. 62
II.2 Resultado do caso de teste 2. 62
II.3 Resultado do caso de teste 3. 62
II.4 Resultado do caso de teste 4. 62
II.5 Resultado do caso de teste 5. 63
II.6 Resultado do caso de teste 6. 63
II.7 Resultado do caso de teste 7. 63
II.8 Resultado do caso de teste 8. 63
II.9 Resultado do caso de teste 9. 63
II.10 Resultado do caso de teste 10. 63
II.11 Resultado do caso de teste 11. 63
II.12 Resultado do caso de teste 12. 63
II.13 Resultado do caso de teste 13. 64
II.14 Resultado do caso de teste 14. 64

II.15 Resultado do caso de teste 15. 64
II.16 Resultado do caso de teste 16. 64
II.17 Resultado do caso de teste 17. 64
II.18 Resultado do caso de teste 18. 64
II.19 Resultado do caso de teste 19. 64
II.20 Resultado do caso de teste 20. 64
II.21 Resultado do caso de teste 21. 65
II.22 Resultado do caso de teste 22. 65
II.23 Resultado do caso de teste 23. 65
II.24 Resultado do caso de teste 24. 65
II.25 Resultado do caso de teste 25. 65
II.26 Resultado do caso de teste 26. 65
II.27 Resultado do caso de teste 27. 65
II.28 Resultado do caso de teste 28. 65
II.29 Resultado do caso de teste 29. 66
II.30 Resultado do caso de teste 30. 66
II.31 Resultado do caso de teste 31. 66
II.32 Resultado do caso de teste 32. 66
II.33 Resultado do caso de teste 33. 66
II.34 Resultado do caso de teste 34. 66
II.35 Resultado do caso de teste 35. 66

Lista de Tabelas

2.1 Regras RGM, onde E1, E1 e E2 representam objetivos ou tarefas (estendido de [1]) 9

4.1 Classes de equivalência e respectivos casos de teste 41
4.2 Resultados dos testes funcionais 42

Capítulo 1

Introdução

1.1 Motivação

O GODA (Goal-Oriented Dependability Analysis) é um framework de análise de dependabilidade com modelos orientados a objetivos. O GODA é acoplado ao TAOM4E (Tool for Agent Oriented Modeling for Eclipse) [2]. O TAOM4E é um plugin do Eclipse baseado na metodologia de desenvolvimento Tropos. O projeto faz a automatização da geração de modelos, sendo feito com o propósito de automatizar esta análise, afim de reduzir erros decorrentes de análises manuais.

O piStar é uma ferramenta de modelagem que suporta o padrão iStar 2.0 de engenharia de requisitos. O projeto é construído com base nas linguagens HTML e Javascript, utilizando a biblioteca de diagramação JointJS. O piStar é capaz de criar modelos orientados a objetivos da mesma forma que o TAOM4E faz, mas em um browser.

As principais características do piStar que justificaram a sua utilização neste trabalho são:

- A execução ocorre completamente no browser, não sendo necessária nenhum tipo de configuração adicional
- Alta fidelidade visual, fazendo com que os diagramas tenham alta resolução mesmo quando impressos

Com o piStar, é possível modelar o diagrama desejado no próprio browser. A ferramenta é similar ao TAOM4E no que se trata das opções de diagrama, pois possui Actor, Goal, Task (equivalente ao Plan), entre outros. Além disso, possui refinamentos AND e OR, assim como no TAOM4E. Para adicionar propriedades ao diagrama, basta clicar no componente desejado e clicar em "Add Property", informando o nome da propriedade e seu valor. Por fim, a ferramente também dá a opção de salvar o modelo criado, bem

como carregar um modelo salvo anteriormente, utilizando para isso JSON. A ferramenta em execução é mostrada na Figura 2.6.

O projeto original GODA se apresenta como um projeto monolítico. De acordo com Dragoni et al. [3], aplicações monolíticas são aplicações de software em que os módulos não podem ser executados independentemente. Pode-se dizer que aplicação monolítica consiste de uma única camada de aplicação que dá suporte à interface do usuário, regras de negócio, e a manipulação de dados. Projetos monolíticos possuem as seguintes limitações [3]:

- Projetos maiores são difíceis de manter e evoluir devido às suas complexidades. Encontrar defeitos requer um tempo maior de análise do código.
- Ocorre o que é chamado de "dependency hell", na qual adicionar ou atualizar bibliotecas resulta em sistemas inconsistentes que não compilam ou executam, ou mesmo executam errôneamente.
- Qualquer mudança em um dos modulos do projeto requer a reinicialização de toda a aplicação. Isto implica que, para qualquer mudança, haverá indisponibilidade do projeto inteiro.
- O *deployment* do projeto requer um ambiente que se adeque a todos os módulos do sistema, e não para cada módulo específico.
- A escalabilidade do projeto é extremamente limitada. Para escalar, é necessário criar novas instâncias do projeto e fazer o *deployment* em ambientes diferentes. Porém, pode ser o caso de a necessidade de escalar o projeto seja apenas de um módulo específico, mas como o projeto é monolítico, deve ser criada um nova instância da aplicação inteira.
- Projetos monolíticos representam uma limitação de desenvolvimento, já que as tecnologias do projeto não podem ser alteradas sem um completo refatoramento de todo o projeto.

O GODA se apresenta como um projeto complexo, composto de vários módulos que se intercomunicam entre si. Para evolução deste projeto, é necessário um conhecimento profundo de todos os módulos, pois alterações em um dos módulos pode interferir diretamente em outros. Encontrar defeitos no projeto também é bastante desafiador, já que falhas podem acarretar um efeito cascata por envolver um forte acoplamento entre os módulos.

O GODA é construído de forma que não há uma separação clara entre o *frontend* e o *backend* do projeto. O *frontend* seria, teoricamente, o plugin TAOM4E. Porém, diversas

classes da própria plataforma Eclipse em que o TAOM4E é construído se apresentam como dependências do projeto GODA. Isto implica que o GODA não consegue funcionar sem que o plugin esteja executando no mesmo ambiente. Logo, pode-se dizer que não há distinção entre *frontend* e *backend*, já que ambos devem estar executando simultaneamente no mesmo ambiente.

O *deployment* deste tipo de projeto é feito como um único projeto, o que implica que há apenas um ponto de falha. Caso haja qualquer problema neste ponto, o projeto inteiro fica indisponível. Este é um aspecto indesejável para um projeto. Apesar dessas desvantagens, um projeto monolítico possui como vantagem o *deployment* simplificado. Há apenas um pacote e apenas um ponto de acesso, o que simplifica a infraestrutura.

No projeto GODA, é observado um alto acoplamento com o plugin TAOM4E. Além disso, esta integração envolve dependências do ambiente de desenvolvimento do Eclipse. Estas dependências fazem com que a manutenção do projeto se torna impraticável, pois não há suporte ao plugin TAOM4E, além do fato de este plugin ser recomendado para um versão legada do Eclipse. Além disso, mudanças ou atualizações nas dependências existentes podem fazer com que o projeto não funcione, pelo fato de o plugin não possuir mais suporte.

Os componentes do projeto GODA se encontram acoplados à plataforma Eclipse (Eclipse Platform). Isto significa que é necessário a reinicialização completa da aplicação caso haja uma mudança no projeto GODA ou no plugin TAOM4E. Esta reinicialização implica que há indisponibilidade do projeto neste intervalo, o que não é desejável no ponto de vista da disponibilidade. O objetivo deste trabalho é remover este acoplamento, fazendo com que o GODA seja apresentado como um serviço, e não parte de um projeto com alto acoplamento.

Para realizar o *deployment* do GODA, é necessário que o ambiente atenda todos os módulos do projeto, e não módulos individuais. Isto é visto na configuração inicial do projeto, em que é necessário seguir múltiplos passos para que o projeto seja executado corretamente. Esta configuração extensa é vista como uma desvantagem, pois pode ser o caso de apenas um módulo específico precisar ser configurado ou requerer mais recursos computacionais. No caso do GODA, a configuração deve ser feita para todo o projeto, e os recursos computacionais adicionados se aplicarão a todos os módulos.

A única forma de escalar o GODA, em sua forma original, seria executar mais instâncias do projeto em ambientes diferentes. O projeto é executado em uma máquina local. Para cada usuário utilizando o GODA, é necessário um ambiente completamente diferente, e este ambiente pode ser acessado por apenas um usuário de cada vez. Logo, é possível afirmar que o projeto é monousuário.

A coesão pode ser definida como o grau em que os elementos de um módulo estão

relacionados. Módulos com alta coesão são mais desejáveis, pois esta característica está associada a robustez, confiabilidade, reusabilidade e manutenibilidade. Para fazer com que um módulo tenham coesão, é necessário observar que este módulo executa apenas uma tarefa específica. No GODA atual, observamos que não há uma divisão clara entre *frontend* e *backend*, o que aponta para uma baixa coesão. É desejável aumentar a coesão do projeto, já que ele apresenta baixa coesão na forma em que se encontra.

No intuito de solucionar estas limitações, foi utilizado o conceito de micro-serviços.

1.2 Problema e Hipótese

O problema do GODA é que ele requer uma quantidade grande configuração para ser executado corretamente, além de possuir dependências de ferramentas que já não possuem suporte e não funcionar completamente em sistemas com sistema operacional Windows. Logo, isto representa um impedimento ao usuário.

A hipótese é a de que a integração entre o piStar e o GODA pode ser benéfico ao usuário, eliminando a necessidade de configurações adicionais.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo principal deste trabalho é a integração entre o GODA e o piStar, desacoplando da solução monolítica em Eclipse para uma versão orientada a serviços por meio de microserviços.

1.3.2 Objetivos Específicos

De forma a concluir a integração entre o piStar e o GODA, foram definidos os seguintes objetivos específicos:

- Desacoplar o GODA do Eclipse.
- Implementar uma arquitetura do GODA orientada a serviços.
- Testar a corretude da solução utilizando os mesmos testes funcionais apresentados em [4].

1.4 Descrição dos Capítulos

Este trabalho se divide em mais quatro capítulos. O Capítulo 2 apresentará um referencial teórico dos temas abordados no trabalho. O Capítulo 3 apresentará a metodologia utilizada para fornecer uma solução ao problema encontrado. O Capítulo 4 apresentará os resultados obtidos com o desenvolvimento do trabalho. Por fim, as conclusões serão apresentadas no Capítulo 5.

Capítulo 2

Referencial Teórico

Modelos orientados a objetivos são utilizados no ínicio da análise de requisitos visando explicar a razão de um sistema [5]. Tais modelos podem representar a racionalidade humanos e softwares [6], além de oferecer permitir a análise de objetivos de alto nível.

O conceito de dependabilidade é apresentado em [7] como a capacidade de entregar um serviço que possa ser justificadamente confiável. A dependabilidade engloba os seguintes conceitos:

- Disponibilidade: prontidão para serviço correto.
- Confiabilidade: continuidade para serviço correto.
- Segurança: ausência de consequências catastróficas para o usuário e para o ambiente.
- Integridade: ausência de alterações impróprias no sistema.
- Manutenibilidade: capacidade e facilidade de ser modificado ou reparado.

2.1 Análise de Dependabilidade Orientada a Objetivos

O GODA se trata de um framework para a realização de análise de dependabilidade utilizando modelos orientados a objetivos. O software foi proposto com a ideia de oferecer um meio para a administração de requisitos de dependabilidade de um sistema, mas em contexto dinâmico [8]. O GODA é acoplado ao TAOM4E [2], que se trata de uma ferramenta baseada no Tropos e tem a forma de um plugin para o Eclipse.

O GODA é baseado em análise de dependabilidade orientada a objetivos, que tem seu método definido em [8]. A Figura 2.1 traz o processo realizado pelo GODA, que tem como entrada um Modelo Orientado a Objetivos produzido de forma convencional. A

informação de contexto e de tempo de execução é inserida logo após, o que faz com que o modelo se torne um CRGM (Contextual-Runtime Goal Model). Ao se obter um CRGM, o GODA o traduz automaticamente para um DTMC *(Discrete Time Markov Chain)*.

Tendo-se um DTMC, as propriedades de dependabilidade podem ser apresentadas a partir de fórmulas PCTL (Probabilistic Computation Tree Logic) [9], permitindo a verificação do sistema.

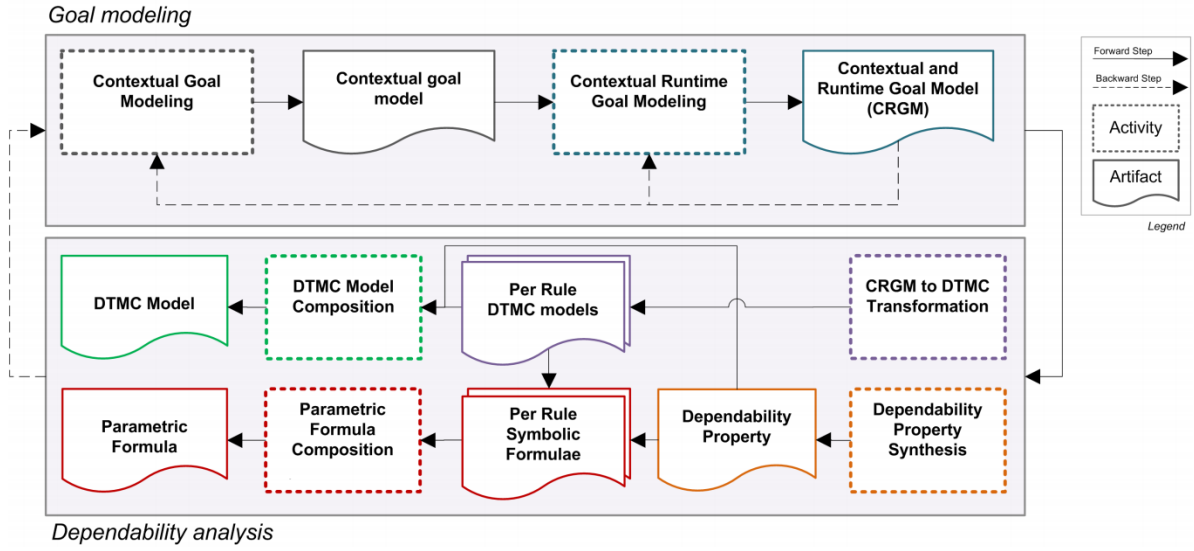

Figura 2.1: Processo do GODA (Fonte: [10]).

2.1.1 Modelagem Orientada a Objetivos e Contexto

Com a modelagem orientada a objetivos e contexto é possível analisar os requisitos dos stakeholders do sistema [11, 12, 13]. A Figura 2.2 traz um exemplo de modelo orientado a objetivo contextual.

Com o RGM (Runtime Goal Model) [1], é possível adicionar especificação comportamental a objetivos e tarefas. No caso do CRGM, objetivos se cumprem considerando a especificação contextual do CGM (Contextual Goal Model) [5] e os objetivos e tarefas instanciados em tempo de execução, segundo as regras do RGM [1]. A Tabela 2.1 define cada regra RGM e dá um significado de acordo com seu comportamento. A Figura 2.3 ilustra um CRGM.

2.1.2 Arquitetura de Implementação do GODA

O GODA faz a automatização da geração de modelos DTMC, partindo de um CRGM, para linguagens PRISM e PARAM. O ambiente de modelagem utiliza o plugin TAOM4E

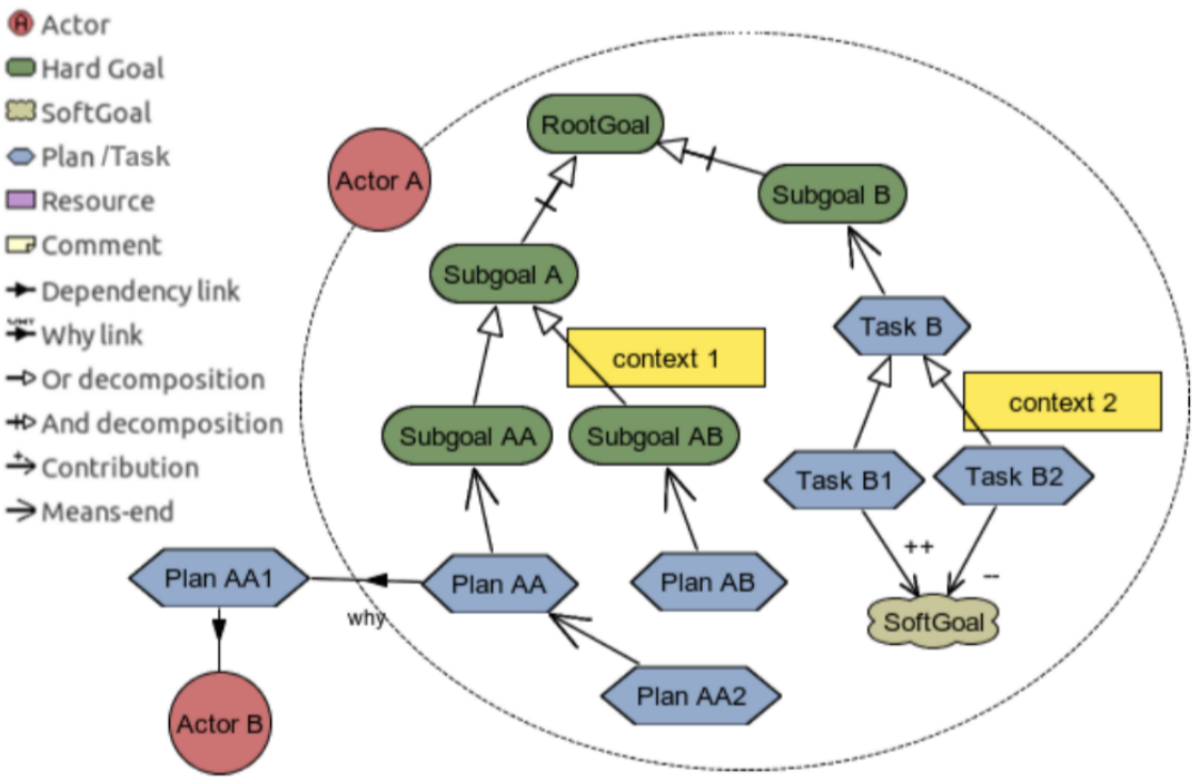

Figura 2.2: Exemplo de modelo orientado a objetivo contextual (Fonte: [8]).

[2], que utiliza a metodologia de desenvolvimento Tropos. Este plugin oferece uma interface gráfica para a modelagem de objetivos. O gerador de código DTMC foi implementado em linguagem Java, e foi integrado ao plugin TAOM4E. O propósito do GODA é gerar código DTMC a partir de CRGM automaticamente, reduzindo erros decorrentes de análises manuais, e permitindo que os analistas foquem na modelagem.

A Figura 2.4 traz uma visão de alto nível da arquitetura do GODA. O GODA recebe como entrada um arquivo CRGM no formato que o plugin TAOM4E determina e gera os modelos PRISM e PARAM como saída.

2.2 Geração Automática de Código DTMC

Uma das etapas da análise de dependabilidade é a geração automática de um modelo DTMC em linguagens PRISM e PARAM partindo de um CRGM. O processo de geração de código CRGM em DTMC foi dividido em duas fases [10]:

- Fase de análise: o arquivo de entrada Tropos contendo um modelo orientado a objetivo com um ator do sistema, é analisado utilizando um algoritmo de profundidade. Objetos de objetivos e tarefas são mantidos em memória com qualquer metadado

Tabela 2.1: Regras RGM, onde E1, E1 e E2 representam objetivos ou tarefas (estendido de [1])

Expressão	**Significado**
AND(E1;E2)	Realização sequencial de E1 e E2.
AND(E1#E2)	Realização em paralelo de E1 e E2.
OR(E1;E2)	Realização sequencial de E1 ou E2, ou ambos.
OR(E1#E2)	Realização em paralelo de E1 ou E2, ou ambos.
E+n	E deve ser realizado n vezes, com n >0.
E#n	Realização em paralelo de n instâncias de E, com n >0.
E@n	Máximo de n - 1 tentativas de realização de E,com n >0.
opt(E)	Realização de E é opcional.
try(E)?E1:E2	Se E é executado, E1 deve ser executado; senão, E2.
E1|E2	Realização alternativa de E1 or E2, não ambos.
skip	Sem ação. Útil em expressões ternárias condicionais.

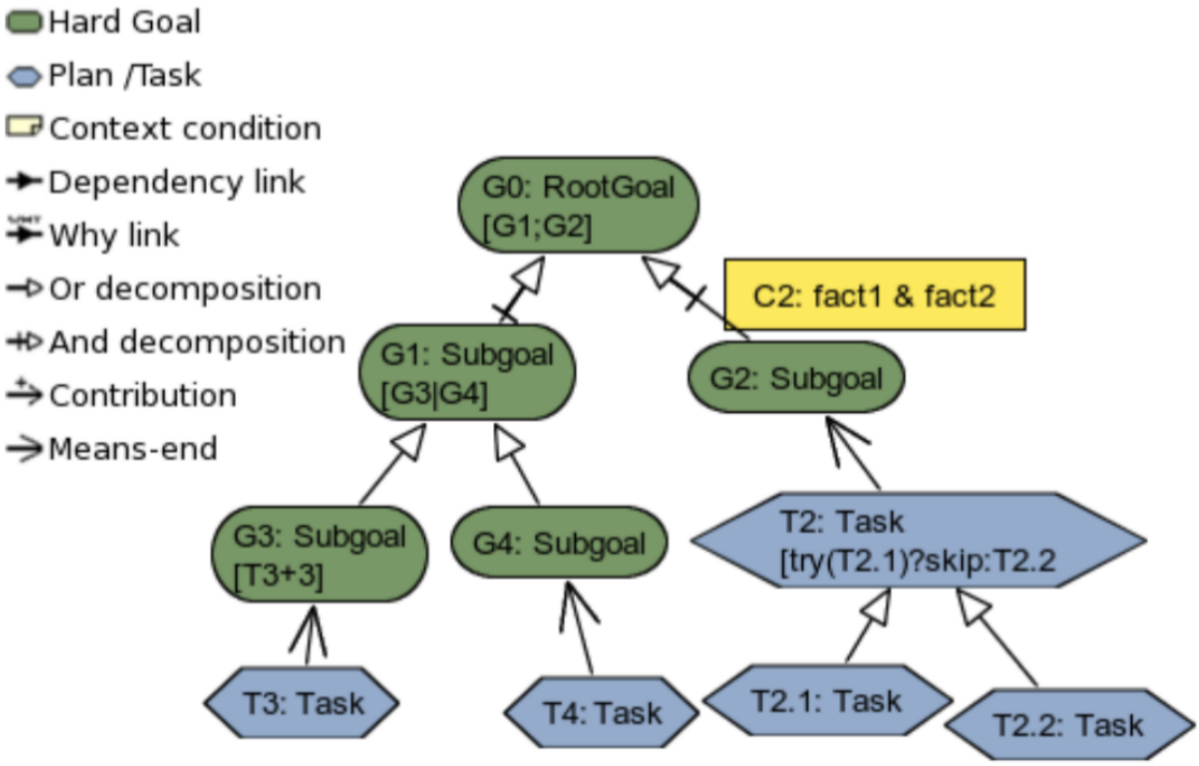

Figura 2.3: Exemplo de um CRGM (Fonte: [8]).

que seja relevante. Também são analisados as anotações de comportamento em elementos não-folhas e os efeitos de contexto em qualquer objetivo ou tarefa. Em uma decomposição da árvore, as regras comportamentais e os efeitos de contexto devem alcançar as tarefas-folhas no final dos ramos relacionados.

- Fase de escrita: Os objetos com as informação necessárias para a geração de um

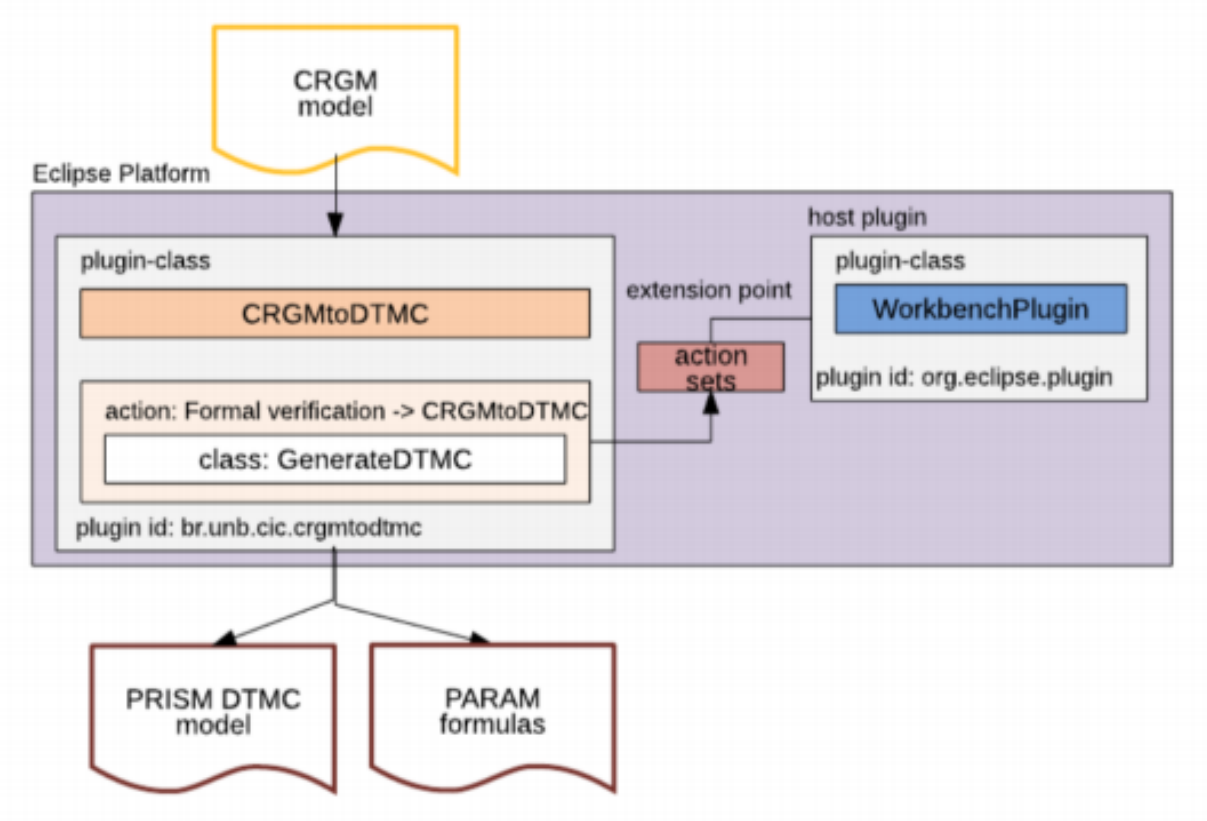

Figura 2.4: Visão de alto nível da arquitetura de implementação do GODA (Fonte: [10]).

modelo DTMC são cruzadas com a do objeto objetivo raiz. Os módulos de tarefas-folhas do DTMC são criados e concatenados à uma variável global. Depois do processamento de todos os efeitos de contexto, uma declaração das variáveis de contexto analisadas é anexada ao modelo. O arquivo de saída com o modelo DTMC é criado em uma pasta pré–configurada pelo usuário do GODA. O nome do arquivo possui em sua composição o nome do ator.

A Figura 2.5 apresenta a arquitetura de implementação de gerador CRGM em DTMC. As classes na Figura 2.5 interagem da seguinte maneira (Estendido de [10]):

1. O usuário clica no item do menu do TAOM4E chamado “Generate PRISM model”.
2. O singleton DTMCAction recebe o contexto de ambiente do modelo Tropos e cria uma nova thread para o DTMCProducer, passando o arquivo de entrada Tropos do contexto de modelagem.
3. O TroposNavigator acessa o modelo TAOM4E e recupera os atores no arquivo de entrada.

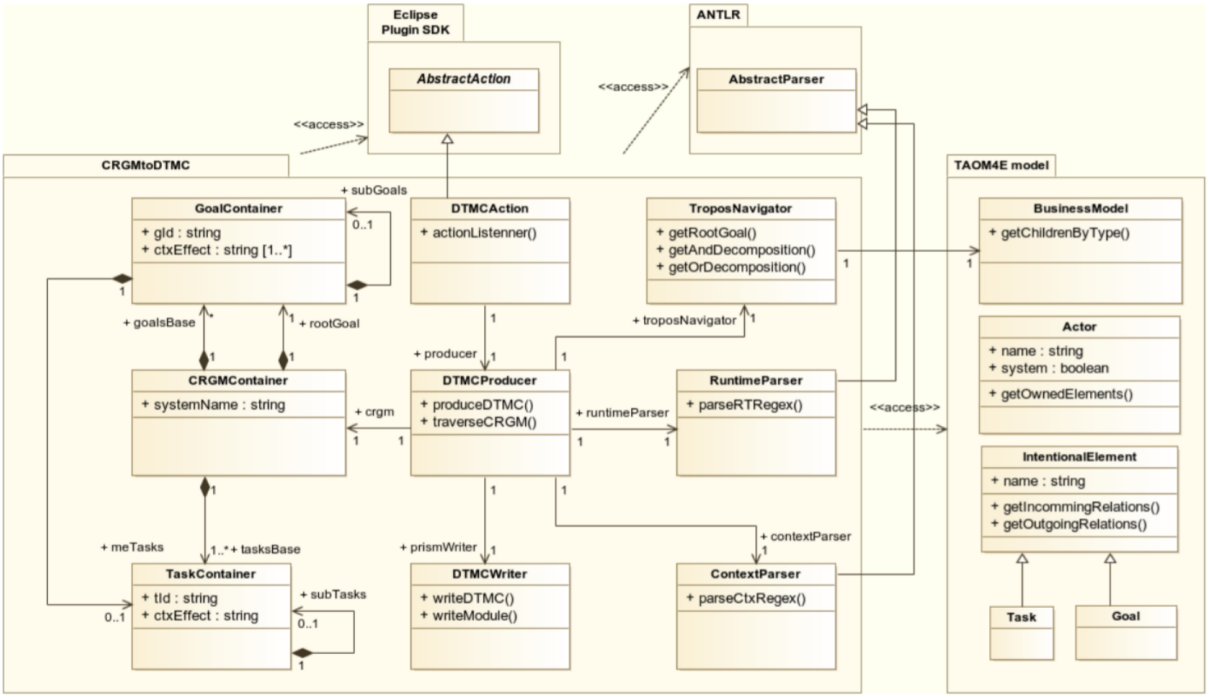

Figura 2.5: Arquitetura de implementação do gerador de DTMC (Fonte: [10]).

4. O ator é identificado e seu objetivo raiz é extraído.
5. O DTMCProducer executa o algoritmo de profundidade pelo objetivo raiz chamando o método addGoal().
6. Cada objetivo ou tarefa não-folha tem seus elementos filhos extraídos e salvos como objetos na instância da classe CRGMDefinition, e logo após ocorre uma chamada recursiva a addGoal()/addTask():
   - Anotações comportamentais são analisadas pelo RTParser e os atributos correspondentes são setados nos objetos dos elementos filhos.
   - Os efeitos de contexto em objetivos e tarefas são analisados pelo CtxParser e os atributos são setados nos objetos dos elementos filhos.
   - É feita uma chamada recursiva a addGoal()/addTask().
7. O retorno do método recursivo de um dado elemento é adicionado aos atributos do elemento pai para propagar incrementos de tempo aninhados para objetivos e tarefas subsequentes.
8. O arquivo DTMCDefinition com os objetivos e tarefas e uma referência para o objetivo raiz é passado para o método writeModel() na classe DTMCWriter.
9. Padrões da linguagem PRISM são carregados a partir de arquivos no pacote de plugins do Eclipse.

10. Um algoritmo de profundidade atravessa o container de objetos, começando pelo objetivo raiz.
    - Tarefas-folhas têm seus módulos criados substituindo seus atributos por tempo, efeitos de contexto e outras particularidades nos padrões correspondentes da linguagem PRISM.
    - Variáveis para comportamentos opcionais são anexadas ao modelo antes dos módulos de tarefas-folhas.
    - Cada chamada de criação de módulo de tarefa-folha retorna uma fórmula booleana para seu sucesso.
    - Fórmulas de sucesso de objetivos e tarefas não-folhas são criadas a partir da concatenação das fórmulas retornadas pelas chamadas recursivas com operadores lógicos da linguagem PRISM, de acordo com o tipo de decomposição do atual elemento raiz.
    - Efeitos de contexto em objetivos e tarefas têm seus pares de tipos e valores de variáveis armazenados em um conjunto de coleções sem duplicações.
11. O arquivo DTMC de saída é aberto.
12. É escrito o cabeçalho do modelo DTMC.
13. São escritos pares de variáveis de contexto como declarações de variáveis unsigned.
14. São escritos os módulos de tarefas-folhas, fórmulas de sucesso de objetivos e variáveis unsigned opcionais.
15. É fechado o arquivo de saída DTMC.

2.3 piStar

Com o piStar, é possível modelar o diagrama desejado no próprio browser. A ferramenta é similar ao TAOM4E no que se trata das opções de diagrama, pois possui Actor, Goal, Task (equivalente ao Plan), entre outros. Além disso, possui refinamentos AND e OR, assim como no TAOM4E. Para adicionar propriedades ao diagrama, basta clicar no componente desejado e clicar em "Add Property", informando o nome da propriedade e seu valor. Por fim, a ferramenta também dá a opção de salvar o modelo criado, bem como carregar um modelo salvo anteriormente, utilizando para isso JSON. A ferramenta em execução é mostrada na Figura 2.6.

A Listing 2.1 exemplifica a estrutura geral de um modelo gerado pelo piStar. Cada elemento do modelo piStar é transformado em um elemento em JSON, com seus respectivos atributos. Esta transformação do modelo em JSON é a forma com que se dá a

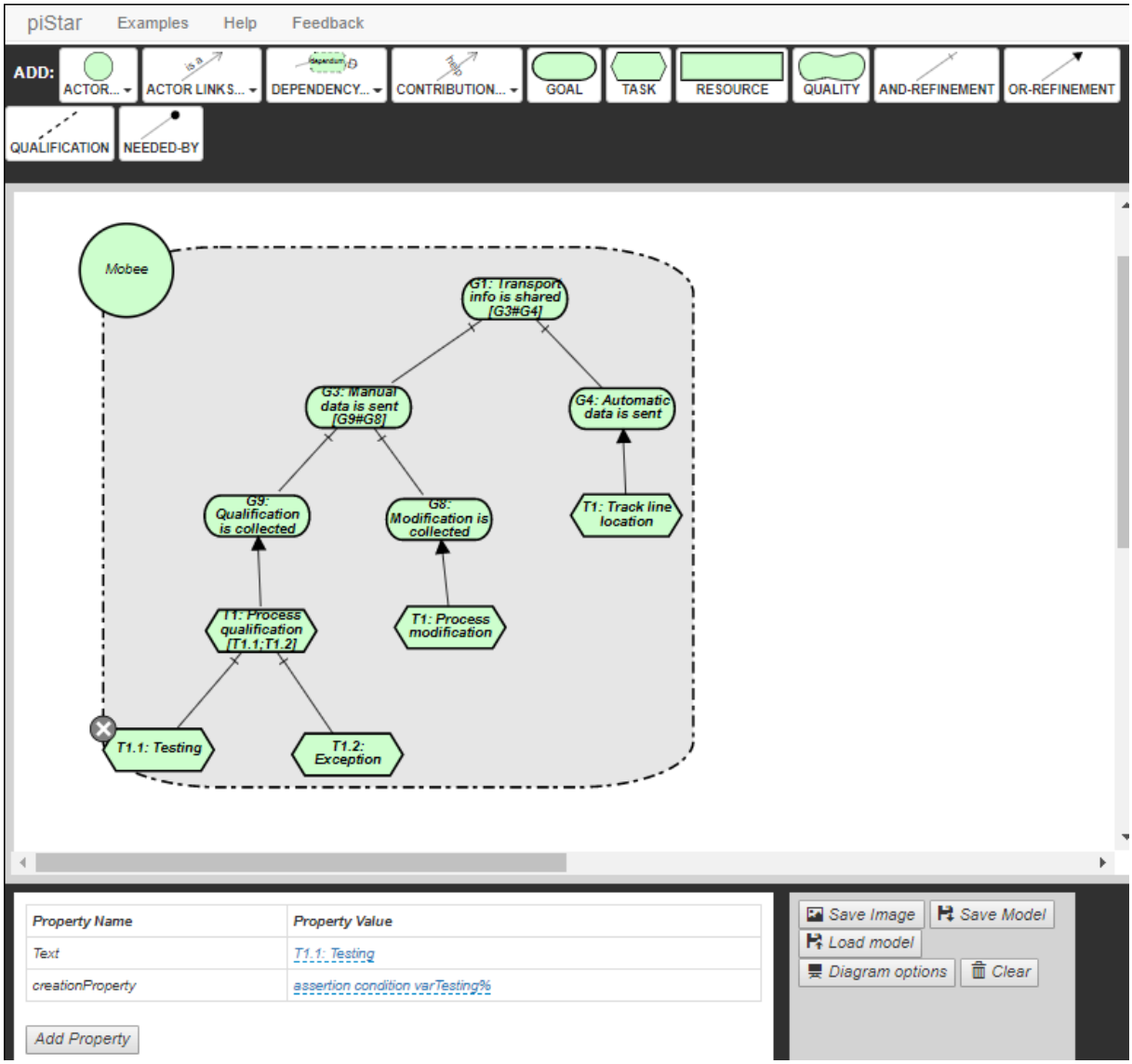

Figura 2.6: Interface do piStar com um modelo real.

integração com o GODA, através da transformação deste arquivo JSON em objetos Java. O elemento "actor" possui, além de suas coordenadas, os seguintes elementos:

- id: Um identificador único, utilizado para identificar as ligações com outros elementos.
- text: O nome do actor.
- type: O tipo é "istar.Actor" quando se trata de actor, mas é possível se tratar de uma Role ou Agent.
- nodes: Os Goals, Tasks, Qualities e Resources pertencentes a este actor.

Cada node possui, também, os atributos id, text, e type. Tanto nodes quanto actors possuem coordenadas, além de uma lista de propriedades, denotada por "customProperties". Cada customProperty possui uma chave e um valor, que são determinados pelas propriedades que o usuário adiciona pelo botão "Add Property". Apesar de ambos Goals e Task serem representados pelo atributo "nodes", eles são diferenciados a partir do atributo "type". Caso este atributo seja igual a "iStar.Goal", se trata de um Goal, e caso seja "iStar.Task", se trata de uma Task.

Os links possuem as seguintes propriedades:

- id: Um identificador único.
- type: O tipo do link, sendo os principais "istar.AndRefinementLink" e "istar.OrRefinementLink".
- source: A origem do link.
- target: O destino do link.

Os atributos "tool", "istar", "saveDate", e "diagram" não serão utilizados na integração, e portanto ignorados na transformação.

A Listing 2.1 exemplifica a estrutura geral do JSON gerado pelo piStar.

```
{
  "actors": [
    {
      <...>
      "nodes": [
        <...>
      ]
    }
  ],
  "dependencies": [
    <...>
  ],
  "links": [
    <...>
  ],
  "tool": "pistar.1.0.0",
  "istar": "2.0",
  "saveDate": "Sat, 06Jan 201823:53:39GMT",
  "diagram": {
    "width": 1881,
    "height": 1172
  }
}
```

Listing 2.1: Estrutura geral do arquivo com o JSON gerado pelo piStar

A Listing 2.2 exemplifica a estrutura de um Actor.

```
{
  "actors": [
    {
```

```
      "id": "de78133e-daa8-4484-9038-135003598cc5",
      "text": "Mobee",
      "type": "istar.Actor",
      "x": 77,
      "y": 124,
      "nodes": [
        <...>
      ],
      <...>
    },
    <...>
  ],
  "dependencies": [
    <...>
  ],
  "links": [
    <...>
  ],
  <...>
}
```

Listing 2.2: Arquivo .txt com o JSON gerado pelo piStar

A Listing 2.3 exemplifica a estrutura de um Goal.

```
{
  "actors": [
    {
      "nodes": [
        {
          "id": "3f734e92-ed0c-4152-9001-8f742bbad3db",
          "text": "G1: Transport info is shared [G3#G4]",
          "type": "istar.Goal",
          "x": 385,
          "y": 151,
          "customProperties": {
            "selected": "true"
          }
        },
        <...>
      ],
      <...>
    },
    <...>
  ],
  <...>
}
```

Listing 2.3: Arquivo .txt com o JSON gerado pelo piStar

A Listing 2.4 exemplifica a estrutura de uma Task.

```
{
  "actors": [
    {
      "nodes": [
        {
```

```
        "id": "0053dec8-999b-4374-adb2-7a609ccc2396",
        "text": "T1.1: Testing",
        "type": "istar.Task",
        "x": 77,
        "y": 537,
        "customProperties": {
          "creationProperty": "assertion condition varTesting%"
        }
      },
      <...>
    ],
    <...>
  },
  <...>
],
<...>
}
```

Listing 2.4: Arquivo .txt com o JSON gerado pelo piStar

A Listing 2.5 exemplifica a estrutura de um link entre Nodes.

```
{
  <...>
  "links": [
    {
      "id": "c538a1b1-d43a-41b3-b5d5-5af1c6fadd35",
      "type": "istar.AndRefinementLink",
      "source": "4c01389a-bcde-42f8-8d99-d3b7cbaa2b9f",
      "target": "3f734e92-ed0c-4152-9001-8f742bbad3db"
    },
    <...>
  ],
  <...>
}
```

Listing 2.5: Arquivo .txt com o JSON gerado pelo piStar

2.4 Princípios Arquiteturais Adotados

Os principios arquiteturais adotados neste projeto são descritos a seguir. O principal foco na construção do projeto foi o princípio SRP do SOLID, a partir da utilização de micro serviços. As seções a seguir explicam estes padrões e como foram utilizados no projeto.

2.4.1 SOLID

O termo "SOLID" representa um acrônimo para cinco princípios que têm a intenção de aumentar a manutenibilidade, flexibilidade e compreensão de um software. Os cinco princípios são listados a seguir [14]:

- The Single Responsibility Principle (SRP): Uma classe deve possuir uma, e apenas uma, razão para ser modificada.
- The Open Closed Principle (OCP): Deve ser possível estender o comportamento de uma classe sem modificá-la.
- The Liskov Substitution Principle (LSP): Classes derivadas devem ter a capacidade de substituir suas classes pai.
- The Interface Segregation Principle (ISP): Faça interfaces de baixa granularidade que são específicas para clientes.
- The Dependency Inversion Principle (DIP): Dependa de abstrações, não concreções.

Este trabalho irá focar mais no princípio SRP, apesar de considerar os outros quatro princípios. O SRP define que uma classe deve ter apenas um razão para ser modificada. A importância de separar as responsabilidades em classes diferentes é justificada em caso de mudança de requisitos.

Caso haja mudança de um requisito, é desejável que apenas uma classe seja alterada. Se uma classe possui mais de uma responsabilidade, haverá mais de uma razão para mudá-la. Quando temos classes com alta coesão, as responsabilidades estarão bem separadas por classes. Isto aumentará a manutenibilidade, já que haverá apenas uma razão para mudar uma classe, não afetando outras responsabilidades que não estejam com esta classe.

2.4.2 Arquitetura Orientada a Microsserviços

Microsserviços é um estilo de arquitetura inspirado por computação orientada a serviços que recentemente ganhou popularidade. Um microsserviço é um serviço que serve um propósito bem definido, além de apresentar alta coesão e baixo acoplamento. No caso deste trabalho, foi considerada a implementação de um microsserviço que implementa as funções do GODA, desacoplado do *frontend*, que é representado pelo piStar. O microsserviço GODA não terá dependências com o piStar, podendo, inclusive, ser executado sem a necessidade do *frontend*, a partir de requisições feitas diretas por aplicativos como o Postman [15].

A arquitetura orientada a microsserviços propõe soluções às limitações apresentadas por projetos monolíticos. Estas soluções são apresentadas respectivamente a seguir, de acordo com [3]:

- Microsserviços implementam uma quantidade limitada de funcionalidades, o que faz com que o código seja pequeno, limitando o escopo dos defeitos. Além disso, como microsserviços são independentes, desenvolvedores podem testar cada microsserviço sem a necessidade de testar o projeto inteiro.

- É possível evoluir microsserviços gradualmente, sem afetar o resto do projeto. Pode-se evoluir o microsserviço e fazer o *deployment* no mesmo ambiente da versão antiga, e assim gradualmente redirecionar as requisições da versão antiga para a nova versão. Este aspecto garante uma integração contínua dos módulos do projeto.
- A mudança de um módulo do sistema não requer uma reinicialização completo do projeto. Este aspecto aumenta a disponibilidade do projeto, já que apenas o módulo alterado será reinicializado.
- Microserviços geralmente são contidos e requerem pouca configuração para seu *deployment*. Isto significa que não é necessário ajustar o ambiente do *deployment* para todo o projeto, e sim apenas para o módulo específico.
- A escalabilidade de um microsserviço não requer que todo o projeto seja replicado. É possível ajustar os recursos computacionais para cada módulo, além de adicionar ou eliminar instâncias de módulos específicos.
- A única limitação tecnológica dos microsserviços são suas forma de comunicação utilizadas entre eles. É possível alterar a linguagem ou framework de um microsserviço, desde que sua forma de comunicação com outros microsserviços permaneça a mesma.

A Figura 2.7 mostra a relação entre aplicações monolíticas e microsserviços. É possível observar os aspectos de aplicações monolíticas explicadas anteriormente nesta imagem, assim como os aspectos de microsserviços.

O microsserviço GODA implementará apenas as funcionalidades do GODA, e não aquelas que concernem ao plugin TAOM4E. O GODA será o *backend* do projeto, e o piStar será o *frontend*, havendo uma separação clara entre as duas camadas. Com esta organização do projeto piStar-GODA, foi possível a implementação de teste unitários do microsserviço GODA, sem a necessidade da utilização do piStar. A implementação destes testes unitários não era possível no GODA original.

Caso haja a necessidade de evolução do piStar-GODA, é possível evoluir o *frontend* independentemente do *backend*, e vice-versa. O *backend* pode possuir várias instâncias executando, e as requisições originárias do *frontend* podem ser gradualmente redirecionadas para a versão atualizada do *backend*.

A evolução de um módulo do piStar-GODA não requer a reinicialização de toda aplicação para ser integrada. É possível evoluir apenas um microsserviço, e reiniciar este microsserviço específico enquanto os outros módulos continuam em execução. Além disso, é possível reiniciar gradativamente o módulo alterado, já que é possível deixar instâncias do código antigo executando enquanto a reinicialização é executada. Isto aumenta a disponibilidade do sistema.

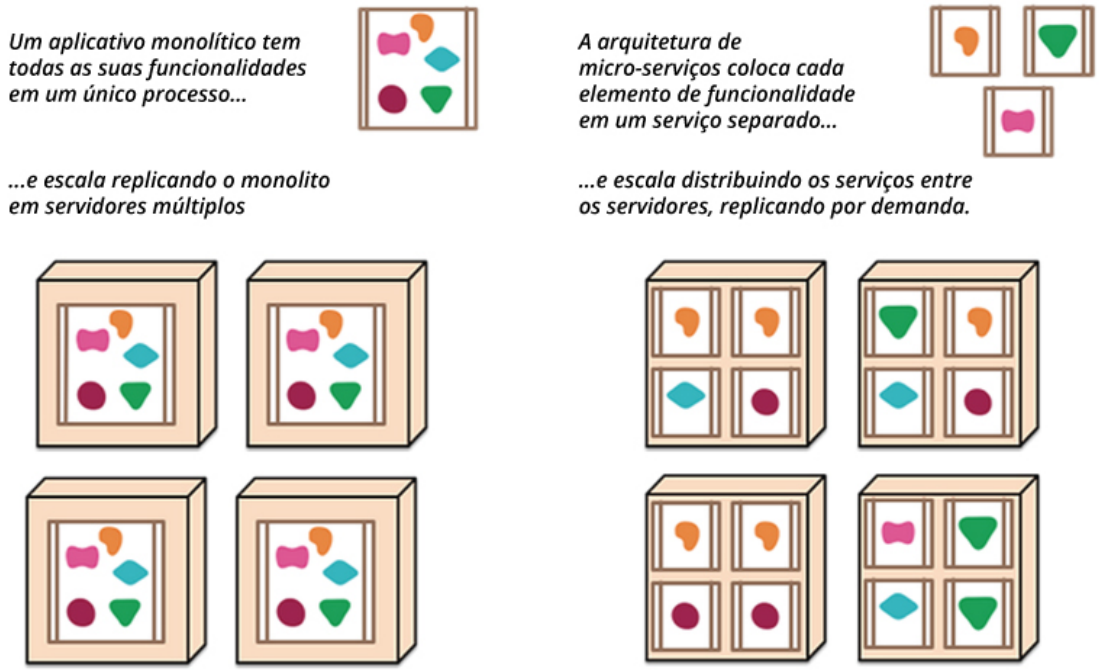

Figura 2.7: Aplicações monolíticas e microsserviços (Fonte: [16]).

A preparação do ambiente para que haja uma evolução do código de um microsserviço pode ser focada apenas no respectivo microsserviço. Não é necessário considerar o ambiente de outros microsserviços, como o *frontend*, se o que está sendo atualizado é o *backend*. Esta característica permite que ambientes independentes possam funcionar em conjunto para que o projeto como um todo execute.

Transformando o GODA em um microsserviço permite que seja possível criar qualquer quantidade de instâncias do *backend* ou do *frontend* que seja necessária. Enquanto o GODA versão Eclipse é monousuário, o piStar-GODA é multiusuário.

Caso seja desejável a alteração da tecnologia em que o microsserviço é construído, é necessário apenas atentar à forma de comunicação deste microsserviço com os outros serviços. No exemplo do piStar-GODA, o microsserviço é feito utilizando-se Spring Boot e REST, mas seria possível alterar para outro framework que também utilize REST como comunicação.

Capítulo 3

Proposta

A configuração e utilização do framework GODA não é trivial. É necessária a instalação do plugin TAOM4E com uma versão específica do Eclipse (4.5), pois não é garantido que funcionará em outras versões. Além disso, o plugin não oferece mais suporte, não dando opção de remediação de erros caso eles ocorram. É necessário também importar o código, fazer uma pré-configuração do plugin, e executá-lo na própria máquina do usuário. A realização de todos estes passos para a execução do framework é exaustiva e suscetível a erros.

Para solucionar estes problemas, é importante utilizar ferramentas modernas para desonerar o usuário da realização de todas estas configurações iniciais. Deve-se remover a dependência da plugin TAOM4E, pelo fato de que este só funciona no Eclipse, além de ser uma versão específica deste. Mais que isto, deve-se remover também a dependência do Eclipse, pois desenvolvedores que desejem contribuir para o projeto devem poder escolher entre suas IDE's favoritas.

Este capítulo apresentará a arquitetura da solução implementada, iniciando com uma introdução ao piStar e sua utilidade. Em seguida, serão apresentadas os detalhes das ferramentas utilizadas para realização da integração, além de algumas partes principais do código fonte.

3.1 GODA as a Service

Para solucionar os problemas do GODA versão Eclipse, foram construídos os projetos pistargoda-*frontend* e pistargoda-*backend*. Estes projetos representam o *frontend* e o *backend* da aplicação piStar-GODA, e podem ser executados independentemente um do outro. O *frontend* não requer qualquer tipo de configuração para ser executado, já que se trata do projeto piStar com algumas modificações. Estas modificações, porém, não

afetaram suas dependências ou tecnologias, o que significa que o projeto é composto basicamente de HTML e Javascript.

O projeto pistargoda-*backend* representa o *backend* da aplicação. Ele possui o código fonte necessário para a transformação dos modelos, mas é independente do projeto pistargoda-*frontend*, ou seja, pode ser executado independentemente. O projeto é construído em Java 8, e as principais ferramentas utilizadas são o Spring Boot e Maven. O projeto é executável em qualquer IDE, à escolha do desenvolvedor, e não requer nenhuma configuração adicional.

O projeto pistargoda-*backend* poderia ser dividido ainda mais, construindo um projeto para cada módulo, mas isto faria com que a configuração ficasse complexa e provavelmente sem agregar valor ao produto em si. Para efeito de demonstração, criou-se o projeto "pistargodaintegration"[1], que une o *frontend* e o *backend* em um projeto só, e que pode ser executado sem nenhuma configuração adicional. Sua implementação e execução são mostradas a seguir.

3.1.1 Implementação

O projeto piStar-GODA é composto de cinco módulos, sendo eles:

- GODA: Código utilizado para a transformação de modelos em PRISM e PARAM.
- Integration: Código que transforma os modelos do piStar em objetos Java, e faz a integração com o módulo GODA utilizando estes objetos.
- piStar: Código-fonte do piStar, com algumas adaptações.
- piStar Model: Modelo do piStar.
- piStarJ Model: Modelo do piStarJ que será utilizaod pelo módulo GODA.

A Figura 3.1 traz o diagrama de implementação completo para a solução desenvolvida neste trabalho.

A Figura 3.2 traz o diagrama de implementação para o modelo do piStar.

A classe "PistarModel" engloba todos os elementos necessários para que o parse do JSON ocorra de forma correta. A estrutura apresentada se trata de uma representação exata do JSON do modelo gerado pelo piStar, permitindo que o parse ocorra de forma automática através da biblioteca Gson [17]. A classe "BaseEntity" apresenta os atributos id e type, que são herdados pelas classes que a estendem. A classe "PistarActor" possui os métodos "getAllGoals()" e "getAllPlans()", que retornam uma lista de objetos "PistarNode", representando todos os Goals do Actor e todos os Plans do Actor, respectivamente.

[1] O projeto "pistargodaintegration" pode ser encontrado em `https://github.com/leandrobergmann/pistargodaintegration`

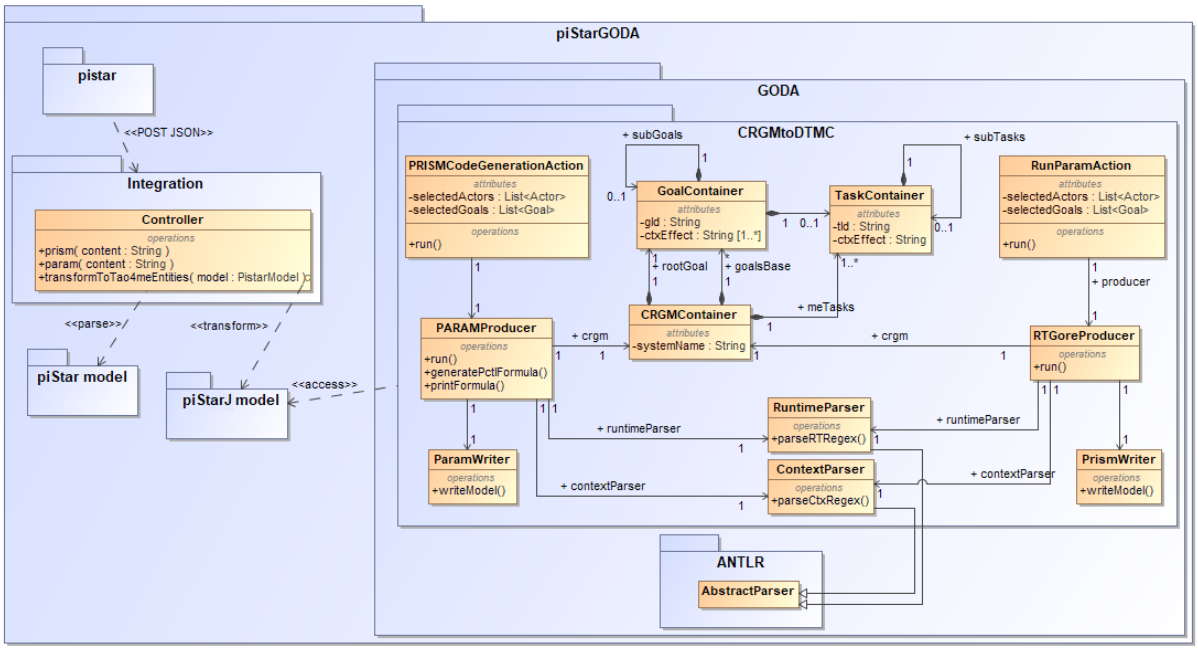

Figura 3.1: Diagrama de implementação da solução completa.

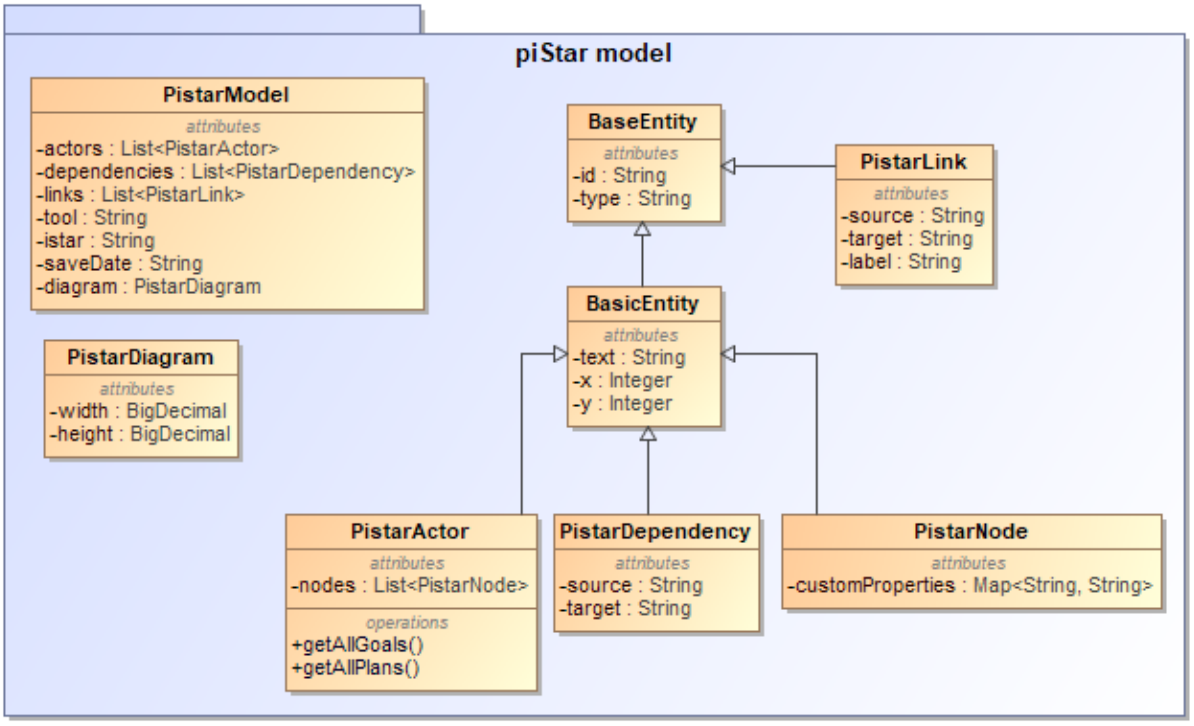

Figura 3.2: Diagrama de implementação do piStar model.

A classe "PistarNode" representa os Goals e Plans, pois estes possuem os mesmo atributos, sendo diferenciados apenas pelo atributo "type". Além disso, todos os elementos

do piStar possuem o atributo “customProperties”, que se trata de um mapa que tem como chave o nome da propriedade adicionada pelo usuário, como por exemplo “selected” e “creationProperty”, e como valor o valor inserido pelo usuário.

A Figura 3.3 traz o diagrama de implementação para o modelo do piStarJ.

O modelo do piStarJ foi construído de forma a se comportar exatamente da mesma forma que o modelo disponibilizado pelo plugin TAOM4E do Eclipse. Este modelo possui todas as características para que os métodos do módulo CRGMtoDTMC executem corretamente. O modelo é composto pelas interfaces “GeneralEntity”, “Actor”, “Goal”, e “Plan”, que definem os métodos a serem implementados. Os métodos definidos são implementados pelas classes “ActorImpl”, “GoalImpl”, e “PlanImpl”.

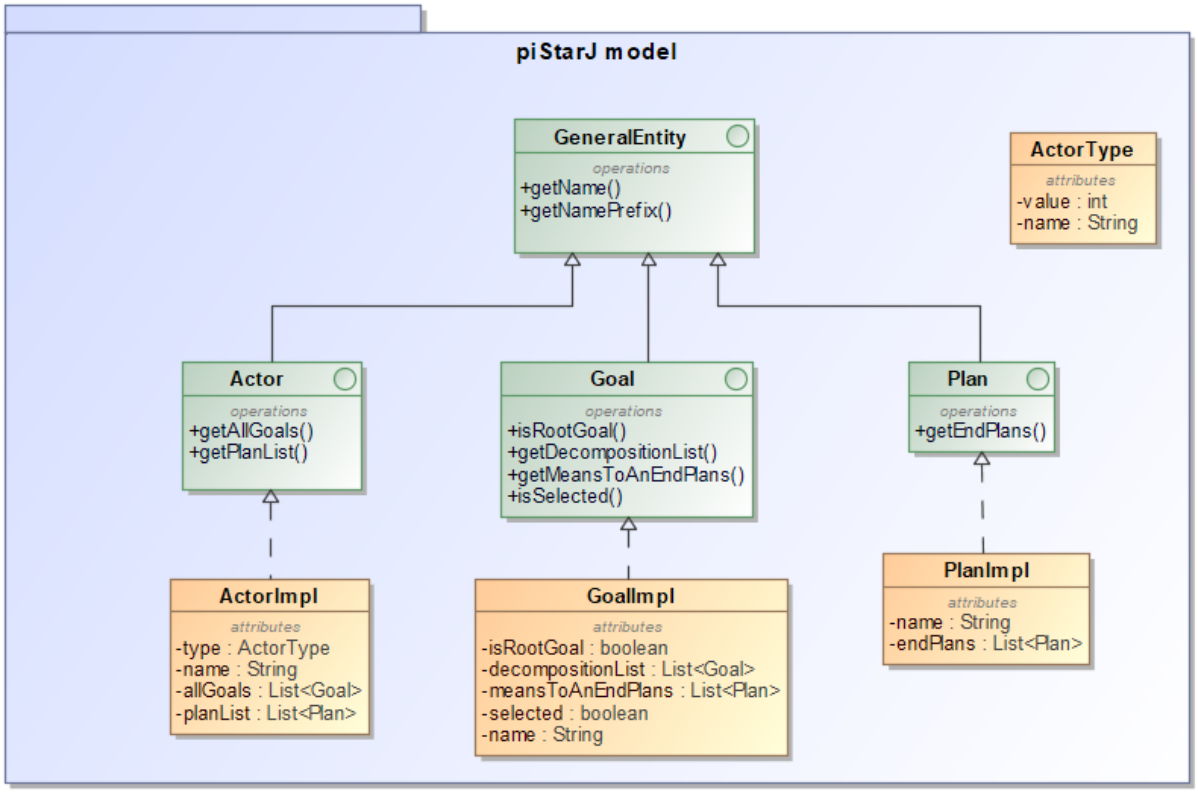

Figura 3.3: Diagrama de implementação do piStarJ model.

3.1.2 Dependências

As dependências do projeto foram simplicadas em relação ao GODA Eclipse. O arquivo “pom.xml” é o único arquivo de configuração necessário, que é utilizado pelo Apache Maven para importar as dependência do projeto. Este arquivo é mostrado na Listing 3.1

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
3           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <groupId>unb.cic</groupId>
6     <artifactId>pistarGODA</artifactId>
7     <version>0.1.0</version>
8     <parent>
9         <groupId>org.springframework.boot</groupId>
10        <artifactId>spring-boot-starter-parent</artifactId>
11        <version>1.5.9.RELEASE</version>
12    </parent>
13    <dependencies>
14        <dependency>
15            <groupId>org.springframework.boot</groupId>
16            <artifactId>spring-boot-starter-web</artifactId>
17        </dependency>
18        <dependency>
19            <groupId>org.springframework.boot</groupId>
20            <artifactId>spring-boot-starter-test</artifactId>
21            <scope>test</scope>
22        </dependency>
23        <dependency>
24            <groupId>com.google.code.gson</groupId>
25            <artifactId>gson</artifactId>
26            <version>2.8.2</version>
27        </dependency>
28        <dependency>
29            <groupId>org.antlr</groupId>
30            <artifactId>antlr4-maven-plugin</artifactId>
31            <version>4.3</version>
32        </dependency>
33    </dependencies>
34    <properties>
35        <java.version>1.8</java.version>
36    </properties>
37    <build>
38        <plugins>
39            <plugin>
40                <groupId>org.springframework.boot</groupId>
41                <artifactId>spring-boot-maven-plugin</artifactId>
42            </plugin>
43            <plugin>
44                <groupId>org.apache.maven.plugins</groupId>
45                <artifactId>maven-surefire-plugin</artifactId>
46                <configuration>
47                    <testFailureIgnore>true</testFailureIgnore>
48                </configuration>
49            </plugin>
50        </plugins>
51    </build>
52</project>
```

Listing 3.1: Arquivo de dependências

É possível verificar que o projeto possui apenas 3 dependências: Spring Boot (linhas 13 a 22), Google GSON (linhas 23 a 27), e ANTLR (linhas 28 a 32). O ANTLR já era parte do GODA versão Eclipse, e continua sendo utilizado nesta nova versão. O Google

GSON [17] serve para transformar os objetos JSON em objetos JAVA. Já o Spring Boot é o responsável pela configuração do projeto como um todo e pela construção dos pontos de acesso, como é visto a seguir. O atributo "testFailureIgnore" (linha 47) permite que o projeto possa ser construído mesmo que os testes falhem. No caso deste trabalho, é esperado que alguns testes falhem, pois é o comportamento esperado do GODA conforme [4], o que justifica esta opção.

3.1.3 Single Responsibility Principle - SRP

O princípio SRP do SOLID foi considerado ao construir a solução piStar-GODA. Cada módulo tem uma responsabilidade única, que é exemplificada ao longo deste capítulo. Além disso, buscou-se construir classes que tenham também uma responsabilidade única.

Este princípio é observado nos módulos "piStar model" e "piStarJ model". Estes módulos possuem a única responsabilidade de representar um objeto correspondente a uma etapa da transformação. O módulo "piStar model" representa o modelo construído pelo piStar, e o "piStarJ model" representa o modelo obtido após a transformação do modelo correspondente ao piStar. As classes pertencentes a esses módulos também seguem o princípio SRP, pelo fato de representarem uma entidade na sua menor forma possível.

O módulo "Integration" tem a única responsabilidade de realizar a integração entre o piStar e o GODA. A classe "Controller" deste módulo é o responsável pela integração, e a classe "Application" é responsável por inicializar a aplicação.

O módulo "GODA" é o responsável pela transformação dos modelos. Dentro deste módulo, existem diversas classes que possuem responsabilidades únicas. A classe "PRISM-CodeGenerationAction", por exemplo, é responsável pela inicialização da transformação PRISM, e a classe "RunParamAction" é responsável pela inicialização da transformação PARAM.

3.1.4 Pontos de Acesso

O *backend* do piStar-GODA possui dois pontos de acesso. Um deles é referente ao PRISM, e outro referente ao PARAM. O ponto de acesso do PRISM pode ser acessado através do caminho "/prism-dtmc", e o ponto de acesso do PARAM através do caminho "/param-dtmc". O código da Listing 3.2 mostra como os dois pontos de acesso são construídos a partir do Spring Boot.

```
1  @RestController
2  public class Controller {
3
4   @RequestMapping
5   (value = "/prism-dtmc", method = RequestMethod.POST)
6   public void prism(@RequestParam(value = "content") String content) {
```

```
7   Gson gson = new GsonBuilder().create();
8   PistarModel model = gson.fromJson(content, PistarModel.class);
9   Set<Actor> selectedActors = new HashSet<>();
10  Set<Goal> selectedGoals = new HashSet<>();
11  transformToTAOM4EEntities(model, selectedActors, selectedGoals);
12  new PRISMCodeGenerationAction(selectedActors, selectedGoals).run();
13  <...>
14 }
15
16 @RequestMapping
17 (value = "/param-dtmc", method = RequestMethod.POST)
18 public void param(@RequestParam(value = "content") String content) {
19  Gson gson = new GsonBuilder().create();
20  PistarModel model = gson.fromJson(content, PistarModel.class);
21  Set<Actor> selectedActors = new HashSet<>();
22  Set<Goal> selectedGoals = new HashSet<>();
23  transformToTAOM4EEntities(model, selectedActors, selectedGoals);
24  new RunParamAction(selectedActors, selectedGoals).run();
25  <...>
26 }
27 }
```

Listing 3.2: Pontos de acesso para PRISM e PARAM

Analisando o código, é possível verificar como ocorre a integração. Os métodos recebem um parâmetro chamado “content” (linha 6 e 18), que é uma String representando o modelo desenhado em JSON. Este JSON é transformado em objeto a partir da biblioteca de serialização e deserialização da Google, chamada GSON [17] (linhas 7-8 e 19-20).

Os objetos obtidos a partir desta transformação são representados pela classe "PistarModel"(linha 8 e 20). O método “transformToTAOM4EEntities” (linha 11 e 23) transforma o objeto da classe “PistarModel” nos objetos do módulo TAOM4E model. Estes objetos são utilizados pelas classes “PRISMCodeGenerationAction” e “RunParamAction” (linhas 12 e 24, respectivamente), que realizam a transformação originalmente feita pelo GODA. Os arquivos criados por estas classes são retornados ao final do método, que os disponibiliza para download.

A Listing 3.3 traz o método “transformToTAOM4EEntities”, que é invocado nas linhas 11 e 23 da Listing 3.2. Este método transforma as entidades do piStar para entidades do TAOM4E, além de preencher as listas “selectedActors” e “selectedGoals” (linhas 9-10 e 21-22) os Actors e Goals selecionados pelo usuário atraves da propriedade “selected” como “true”.

O método começa obtendo todos os actors do piStar (linha 2). Após, realiza um loop que percorre todos os actors obtidos (linha 3). Para cada PistarActor, um Actor do TAOM4E é construído (linha 4). Este actor possui uma lista de plans, que são preenchidos logo em seguida (linha 6-11). Feito isto, o método percorre a lista dos PistarNode’s que correspondem a Goals (linha 12). Cada Goal tem sua decompositionList preenchida (linha 13), e em seguida é adicionada ao Actor construído no começo do método (linha 16). É

verificado se o Goal é um goal raiz e setado o atributo correspondente (linhas 13 e 14). Além disso, caso o Goal esteja selecionado, ele é adicionado à lista "selectedGoals" (linha 18), e seu respectivo Actor adicionado na lista "selectedActors" (linha 20).

```
1  private void transformToTao4meEntities(PistarModel model, Set<Actor> selectedActors, Set<Goal> selectedGoals) {
2        List<PistarActor> pistarActors = model.getActors();
3        pistarActors.forEach(pistarActor -> {
4            Actor actor = new ActorImpl(pistarActor);
5            List<PistarNode> notDerivedPlans = new ArrayList<>();
6            notDerivedPlans.addAll(pistarActor.getAllPlans());
7            <...>
8            notDerivedPlans.forEach(notDerivedPlan -> {
9                Plan plan = new PlanImpl(notDerivedPlan);
10               actor.addToPlanList(plan);
11           });
12           pistarActor.getAllGoals().forEach(pistarGoal -> {
13               Goal goal = fillDecompositionList(model, pistarActor, pistarGoal, new GoalImpl(pistarGoal));
14               boolean isRootGoal = model.getLinks().stream().noneMatch(l ->
                      l.getSource().equals(pistarGoal.getId()));
15               goal.setRootGoal(isRootGoal);
16               actor.addHardGoal(goal);
17               if (goal.isSelected()) {
18                   selectedGoals.add(goal);
19                   if (!selectedActors.contains(actor)) {
20                       selectedActors.add(actor);
21                   }
22               }
23           });
24       });
25   }
```

Listing 3.3: Método transformToTAOM4EEntities

A Listing 3.4 traz o método auxiliar "fillDecompositionList" que preenche a lista "decompostionList". Esta lista é utilizada pelo GODA a partir do método "getDecompositionList()", que é mostrado na Figura 3.3.

O método se inicia obtendo todo a lista de PistarLink que corresponde às ligações que tenham como destino o Goal em questão (linha 2 a 4). Após, o método percorre a lista obtida (linha 5). Para cada ligação, é obtido o Goal que está na origem desta (linhas 6 a 8), e determinado se é uma ligação do tipo "AND" ou "OU" (linhas 9 a 14). Após esta definição, é feita uma chamada para o método "fillMeansToAndEndPlansList" (linha 16), para preenchimento da lista "meansToAndEndPlans" do Goal em questão. Para cada Goal que originou a ligação, é feita uma chamada recursiva ao mesmo método atualmente em execução, "fillDecompositionList" (linha 18), e o Goal obtido pela chamada é adicionado na lista de decomposições (linha 19).

```
1  private Goal fillDecompositionList(PistarModel model, PistarActor pistarActor, PistarNode pistarGoal, Goal
       goal) {
2        List<PistarLink> linksToGoal = model.getLinks().stream()
3                .filter(d -> d.getTarget().equals(pistarGoal.getId()) && d.getType().contains("Link"))
4                .collect(Collectors.toList());
```

```
5       linksToGoal.forEach(l -> {
6           List<PistarNode> sourceGoals = pistarActor.getAllGoals().stream()
7                   .filter(g -> l.getSource().equals(g.getId()))
8                   .collect(Collectors.toList());
9           if (!sourceGoals.isEmpty()) {
10              if (l.getType().contains("And")) {
11                  goal.setAndDecomposition(true);
12              } else if (l.getType().contains("Or")) {
13                  goal.setOrDecomposition(true);
14              }
15          }
16          fillMeansToAndEndPlansList(model, pistarActor, pistarGoal, goal);
17          sourceGoals.forEach(g -> {
18              Goal dependencyGoal = fillDecompositionList(model, pistarActor, g, new GoalImpl(g));
19              goal.addToDecompositionList(dependencyGoal);
20          });
21      });
22      return goal;
23  }
```

Listing 3.4: Método fillDecompositionList

A Listing 3.5 traz o método auxiliar "fillMeansToAndEndPlansList" que preenche a lista "meansToAnEndPlans". Esta lista é utilizada pelo GODA a partir do método "getMeansToAnEndPlans()", que é mostrado na Figura 3.3.

O método "fillMeansToAndEndPlansList" se inicia obtendo as ligações que tem como destino o Goal em questão (linhas 2 a 4). Após, a lista obtida é percorrida em um loop (linha 5). Para cada ligação da lista, é obtida a lista de Plans que corresponde às origens das ligações (linhas 6 a 8). A lista de Plans obtida é iterada em um loop (linha 10), e para cada Plan é feita uma chamada ao método "fillEndPlans" para que seja preenchida a lista "endPlans" de cada um deles (linhas 10 e 11).

```
1   private void fillMeansToAndEndPlansList(PistarModel model, PistarActor pistarActor, PistarNode pistarGoal, Goal
        goal) {
2       List<PistarLink> linksToGoal = model.getLinks().stream()
3               .filter(l -> l.getTarget().equals(pistarGoal.getId()) && l.getType().contains("Link"))
4               .collect(Collectors.toList());
5       linksToGoal.forEach(link -> {
6           List<PistarNode> sourcePlans = pistarActor.getAllPlans().stream()
7                   .filter(p -> link.getSource().equals(p.getId()))
8                   .collect(Collectors.toList());
9           sourcePlans.forEach(sp -> {
10              Plan meansToAnEndPlan = fillEndPlans(model, pistarActor, sp, new PlanImpl(sp));
11              goal.addToMeansToAnEndPlans(meansToAnEndPlan);
12          });
13      });
14  }
```

Listing 3.5: Método fillMeansToAndEndPlansList

A Listing 3.6 traz o método auxiliar “fillEndPlans” que preenche a lista “meansToAnEndPlans”. Esta lista é utilizada pelo GODA a partir do método “getEndPlans()”, que é mostrado na Figura 3.3.

O método “fillEndPlans” se inicia obtendo as ligações que têm como destino o Plan em questão (linhas 2 a 4). Após, a lista obtida é percorrida em um loop (linha 5). Para cada ligação da lista, é determinado os Plan’s correspondentes à origem da ligação (linhas 6 a 8). Para cada Plan correspondente à origem da ligação, é determinada se a ligação é do tipo “AND” ou “OR” (linhas 9 a 14). Após isto, a lista de Plan’s das origens da ligação são iterados (linha 16) e é realizada uma chamada recursiva ao mesmo método em execução, “fillEndPlans” (linha 17), com o resultado obtido desta chamada sendo adicionado na lista “endPlans” (linha 18).

```
1  private Plan fillEndPlans(PistarModel model, PistarActor pistarActor, PistarNode pistarPlan, Plan
       meansToAnEndPlan) {
2      List<PistarLink> linksToPlan = model.getLinks().stream()
3              .filter(l -> l.getTarget().equals(pistarPlan.getId()) && l.getType().contains("Link"))
4              .collect(Collectors.toList());
5      linksToPlan.forEach(link -> {
6          List<PistarNode> sourcePlans = pistarActor.getAllPlans().stream()
7                  .filter(p -> link.getSource().equals(p.getId()))
8                  .collect(Collectors.toList());
9          if (!sourcePlans.isEmpty()) {
10             if (link.getType().contains("And")) {
11                 meansToAnEndPlan.setAndDecomposition(true);
12             } else if (link.getType().contains("Or")) {
13                 meansToAnEndPlan.setOrDecomposition(true);
14             }
15         }
16         sourcePlans.forEach(p -> {
17             Plan endPlan = fillEndPlans(model, pistarActor, p, new PlanImpl(p));
18             meansToAnEndPlan.addToEndPlans(endPlan);
19         });
20     });
21     return meansToAnEndPlan;
22 }
```

Listing 3.6: Método fillEndPlans

Estas listas são utilizadas pelo código do GODA versão Eclipse e portanto são necessárias para o correto funcionamento do projeto.

A Listing 3.7 traz a classe “PRISMCodeGenerationAction”. Esta classe é instanceada quando ocorre uma chamada para o ponto de acesso do PRISM. A classe é responsável por iniciar a transformação do CRGM em DTMC na linguagem PRISM. As listas definidas como variáveis locais nesta classe (linhas 3 e 4) são apontadas para as listas inicializadas no passo anterior (linhas 7 e 8), e então utilizadas por esta classe e passadas para a classe “RTGoreProducer” (linha 16). A execução ocorre após estes passos (linha 18).

```
1  public class PRISMCodeGenerationAction {
2
```

```
3     private Set<Actor> selectedActors;
4     private Set<Goal> selectedGoals;
5
6     public PRISMCodeGenerationAction(Set<Actor> selectedActors, Set<Goal> selectedGoals) {
7         this.selectedActors = selectedActors;
8         this.selectedGoals = selectedGoals;
9     }
10
11    public void run() {
12        if (selectedActors.isEmpty())
13            return;
14        String sourceFolder = "src/main/resources/TemplateInput";
15        String targetFolder = "dtmc";
16        RTGoreProducer producer = new RTGoreProducer(selectedActors, selectedGoals, sourceFolder, targetFolder);
17        try {
18            producer.run();
19        } catch (CodeGenerationException | IOException e) {
20            e.printStackTrace();
21        }
22    }
23
24 }
```

Listing 3.7: Classe “PRISMCodeGenerationAction”

A Listing 3.8 traz a classe “RunParamAction”. Esta classe é instanceada quando ocorre uma chamada para o ponto de acesso do PARAM. A classe é responsável por iniciar a transformação do CRGM em DTMC na linguagem PARAM. As listas definidas como variáveis locais nesta classe (linhas 3 e 4) são apontadas para as listas inicializadas no passo anterior (linhas 7 e 8), e então utilizadas por esta classe e passadas para a classe “PARAMProducer” (linha 16). A execução ocorre após estes passos (linha 18).

```
1  public class RunParamAction {
2
3     private Set<Actor> selectedActors;
4     private Set<Goal> selectedGoals;
5
6     public RunParamAction(Set<Actor> selectedActors, Set<Goal> selectedGoals) {
7         this.selectedActors = selectedActors;
8         this.selectedGoals = selectedGoals;
9     }
10
11    public void run() {
12        if (selectedActors.isEmpty())
13            return;
14        String sourceFolder = "src/main/resources/TemplateInput";
15        String targetFolder = "dtmc";
16        String toolsFolder = "tools";
17        PARAMProducer producer = new PARAMProducer(selectedActors, selectedGoals, sourceFolder, targetFolder,
              toolsFolder);
18        try {
19            producer.run();
20        } catch (CodeGenerationException | IOException e) {
21            e.printStackTrace();
22        }
```

```
23      }
24
25  }
```

Listing 3.8: Classe “RunParamAction”

O *frontend* do piStar-GODA acessa o *backend* a partir da de uma requisição POST, enviando uma string que representa o objeto JSON criado a partir do modelo desenhado (linhas 4 a 8 e 19 a 23). A Listing 3.9 demonstra estas chamadas para o ponto de acesso do PRISM e do PARAM, respectivamente.

```
1  $('#runPRISMButton').click(function() {
2      var model = saveModel();
3      $.ajax({
4          type: "POST",
5          url: '/prism-dtmc',
6          data: {
7              "content": model
8          },
9          success: function() {
10             window.location.href = 'prism.zip';
11         },
12         error: function(){alert("Error!");}
13     });
14 });
15
16 $('#runPARAMButton').click(function() {
17     var model = saveModel();
18     $.ajax({
19         type: "POST",
20         url: '/param-dtmc',
21         data: {
22             "content": model
23         },
24         success: function() {
25             window.location.href = 'param.zip';
26         },
27         error: function(){alert("Error!");}
28     });
29 });
```

Listing 3.9: Chamada do *frontend* aos pontos de acesso do *backend*

A requisição POST é enviada ao se clicar nos botões “Generate PRISM DTMC Code” e “Parametric Formula Generation”. O código que adiciona estes botões é mostrado na Listing 3.10.

```
1  <button type="button" id="runPRISMButton">
2   <span class="glyphicon glyphicon-refresh" aria-hidden="true"></span>
3   Generate PRISM DTMC Code
4  </button>
5  <button type="button" id="runPARAMButton">
6   <span class="glyphicon glyphicon-refresh" aria-hidden="true"></span>
7   Parametric Formula Generation
```

```
8 </button>
```

Listing 3.10: Chamada do *frontend* aos pontos de acesso do *backend*

Como é possível observar, basta enviar uma requisição POST para o caminho "/prism-dtmc" ou "/param-dtmc" para executar o *backend*. Isto implica que o *frontend* não é necessário, já que é possível mandar uma requisição POST a partir de vários programas, como o Postman [15]. Ao enviar a requisição, automaticamente ocorre o download do arquivo zipado com os arquivos gerados, devido a requisição GET ao endereço onde o arquivo foi gerado, conforme mostrado na Listing 3.9.

3.2 Execução do piStar-GODA

A execução do piStar-GODA se dá através do browser, da mesma forma que o pistar. Os botões "Generate PRISM DTMC Code" e "Parametric Formula Generation" foram adicionados na lista de ações, e servem como interface entre o piStar e o GODA. O mesmo modelo da Figura 2.6 foi importado no piStar-GODA como exemplo, e é apresentado na Figura 3.4.

Ao clicar em "Generate PRISM DTMC Code" ou "Parametric Formula", o piStar-GODA é executado e disponibiliza para download o arquivo "prism.zip" e "param.zip", respectivamente. Este arquivo zipado possui os arquivos gerados quando estas mesmas opções são selecionadas no plugin TAOM4E. A Listing 3.11 traz o arquivo "eval_formula.sh", e o Listing 3.12 traz o arquivo "Mobee.pm", ambos arquivos gerados quando é clicado o botão "Generate PRISM DTMC Code".

```
1 #!/bin/bash
2 rTaskG8_T1="0.999";
3 rTaskG9_T1_1="0.999";
4 rTaskG9_T1_2="0.999";
5 rTaskG4_T1="0.999";
6
7
8 sed -e "s/rTaskG8_T1/$rTaskG8_T1/g" -e "s/rTaskG9_T1_1/$rTaskG9_T1_1/g" -e
     "s/rTaskG9_T1_2/$rTaskG9_T1_2/g" -e "s/rTaskG4_T1/$rTaskG4_T1/g" $1 |
     gawk '{print "scale=20;"$0}' | bc
9 exit 0;
```

Listing 3.11: Arquivo eval_formula.sh gerado pelo piStar-GODA

```
1 dtmc
2
```

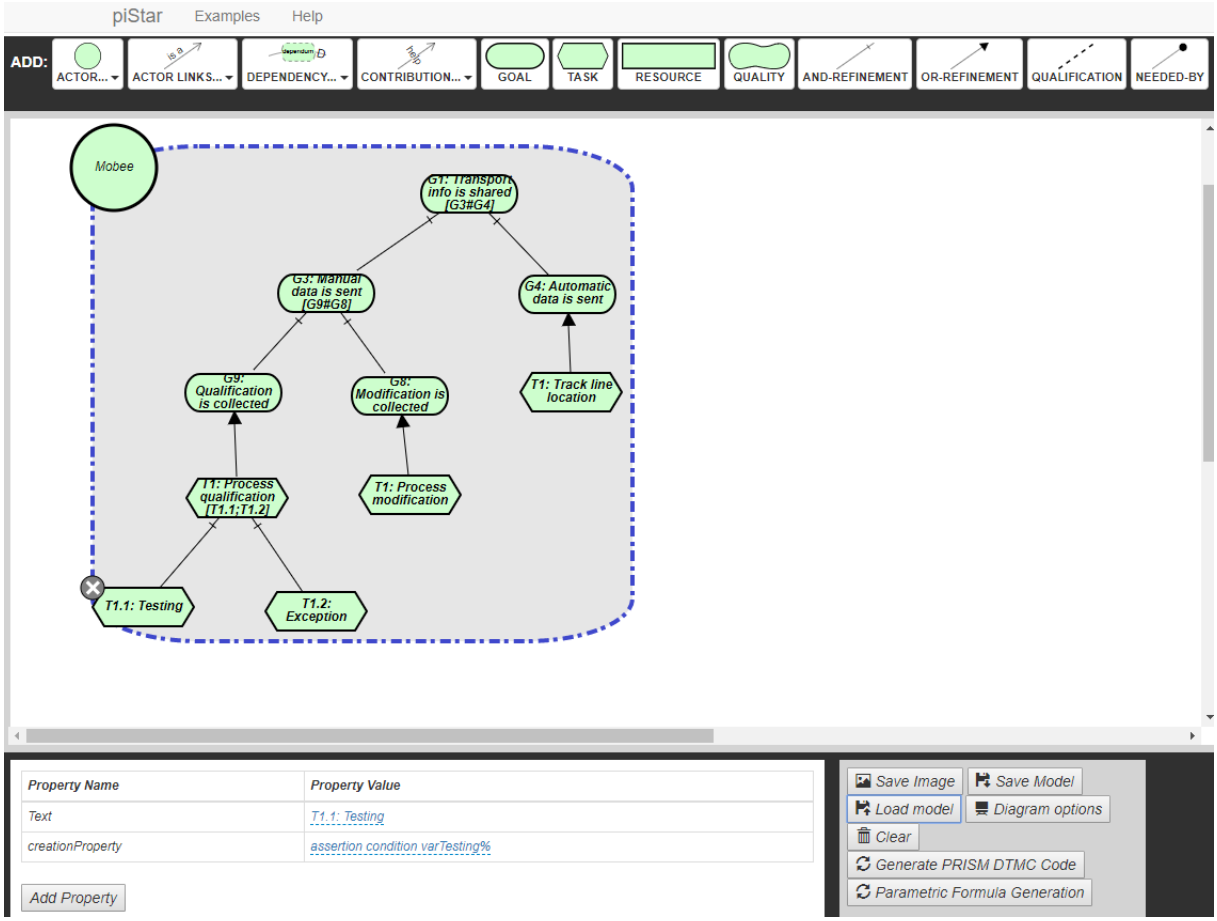

Figura 3.4: Exemplo da execução no piStar-GODA.

```
3  formula noError = true & sG8_T1 < 4 & sG9_T1_1 < 4 & sG9_T1_2 < 4 & sG4_T1 < 4;
4
5
6  const double rTaskG8_T1=0.99;
7
8  module G8_T1_ProcessModification
9    sG8_T1 :[0..4] init 0;
10
11   [success0_0] sG8_T1 = 0 -> (sG8_T1'=1);//init to running
12
13
14
15
16   [] sG8_T1 = 1 -> rTaskG8_T1 : (sG8_T1'=2) + (1 - rTaskG8_T1) : (sG8_T1'=4);//running to final state
17   [success1_1] sG8_T1 = 2 -> (sG8_T1'=2);//final state success
18   [success1_1] sG8_T1 = 3 -> (sG8_T1'=3);//final state skipped
19   [failG8_T1] sG8_T1 = 4 -> (sG8_T1'=4);//final state failure
20 endmodule
21
22 formula G8 = ((sG8_T1=2));
23
24 const double rTaskG9_T1_1=0.99;
```

```
25
26  module G9_T1_1_Testing
27    sG9_T1_1 :[0..4] init 0;
28
29    [success0_0] (varTesting) & sG9_T1_1 = 0 -> (sG9_T1_1'=1);//init to running
30    [success0_0] !(varTesting) & sG9_T1_1 = 0 -> (sG9_T1_1'=3);//init to skipped
31
32
33
34
35    [] sG9_T1_1 = 1 -> rTaskG9_T1_1 : (sG9_T1_1'=2) + (1 - rTaskG9_T1_1) : (sG9_T1_1'=4);//running to final state
36    [success0_1] sG9_T1_1 = 2 -> (sG9_T1_1'=2);//final state success
37    [success0_1] sG9_T1_1 = 3 -> (sG9_T1_1'=3);//final state skipped
38    [failG9_T1_1] sG9_T1_1 = 4 -> (sG9_T1_1'=4);//final state failure
39  endmodule
40
41  const double rTaskG9_T1_2=0.99;
42
43  module G9_T1_2_Exception
44    sG9_T1_2 :[0..4] init 0;
45
46    [success0_1] sG9_T1_2 = 0 -> (sG9_T1_2'=1);//init to running
47
48
49
50
51    [] sG9_T1_2 = 1 -> rTaskG9_T1_2 : (sG9_T1_2'=2) + (1 - rTaskG9_T1_2) : (sG9_T1_2'=4);//running to final state
52    [success0_2] sG9_T1_2 = 2 -> (sG9_T1_2'=2);//final state success
53    [success0_2] sG9_T1_2 = 3 -> (sG9_T1_2'=3);//final state skipped
54    [failG9_T1_2] sG9_T1_2 = 4 -> (sG9_T1_2'=4);//final state failure
55  endmodule
56
57  formula G9 = (((sG9_T1_1=2) | (sG9_T1_1=3 & (!(varTesting)))) & ((sG9_T1_2=2)));
58
59  formula G3 = G8 & G9;
60
61  const double rTaskG4_T1=0.99;
62
63  module G4_T1_TrackLineLocation
64    sG4_T1 :[0..4] init 0;
65
66    [success0_0] sG4_T1 = 0 -> (sG4_T1'=1);//init to running
67
68
69
70
71    [] sG4_T1 = 1 -> rTaskG4_T1 : (sG4_T1'=2) + (1 - rTaskG4_T1) : (sG4_T1'=4);//running to final state
72    [success1_1] sG4_T1 = 2 -> (sG4_T1'=2);//final state success
73    [success1_1] sG4_T1 = 3 -> (sG4_T1'=3);//final state skipped
74    [failG4_T1] sG4_T1 = 4 -> (sG4_T1'=4);//final state failure
75  endmodule
76
77  formula G4 = ((sG4_T1=2));
78
79  formula G1 = G3 & G4;
```

```
80 const bool varTesting;
```

Listing 3.12: Arquivo Mobee.pm gerado pelo piStar-GODA

Ambos os arquivos da Listing 3.11 e Listing 3.12 são idênticos aos arquivos gerados pelo GODA quando é utilizado o plugin TAOM4E para gerar PRISM, o que permite concluir que a integração foi feita com sucesso.

Também foi testado a geração de fórmulas paramétricas, a partir do ponto de acesso "/param-dtmc". Além dos arquivos gerados nas Listings 3.11 e 3.12, o ponto de acesso do PARAM gera os arquivos "reachability.pctl" e "result.out", representados nas Listings 3.13 e 3.14, respectivamente.

```
1 P=? [ true U (G1) ]
```

Listing 3.13: Arquivo reachability.pctl gerado pelo piStar-GODA

```
1 MAX
2 [rTaskG8_T1, rTaskG9_T1_1, rTaskG9_T1_2, rTaskG4_T1]
3 [[0, 1] [0, 1] [0, 1] [0, 1]]
4   ( ( ( 1*rTaskG9_T1_1 * 1*rTaskG9_T1_2 ) * 1*rTaskG8_T1 ) * 1*rTaskG4_T1 )
```

Listing 3.14: Arquivo result.out gerado pelo piStar-GODA

Ambos os arquivos da Listing 3.14 e Listing 3.13 são idênticos aos arquivos gerados pelo GODA quando é utilizado o plugin TAOM4E para geração de fórmulas paramétricas, o que permite concluir que a integração foi feita com sucesso.

3.3 Execução independente

Como explicado na seção anterior, é possível executar o *backend* do piStar-GODA independentemente do *frontend*. Isto é possível através do envio de uma requisição POST para o serviço desejado, contendo como parâmetro o JSON do modelo desenhado anteriormente pelo usuário. Logo, se o usuário desenhou o diagrama em um ambiente, salvou o arquivo JSON e deseja executar o GODA sem acessar o *frontend*, ele pode fazer como na Figura 3.5. Os resultados obtidos a partir desta estratégia foram exatamente iguais aos obtidos quanto o piStar-GODA é executado junto com o *frontend*, o que demonstra a independência entre os diferentes módulos.

A mesma estratégia se aplica à situação contrária, em que o usuário precisa executar o *frontend* sem executar o *backend*. Para isto basta executar o projeto pistargoda-*frontend*, que é feito em HTML e Javascript, e desenhar o diagrama desejado sem a utilização do projeto pistargoda-backend.

A Figura 3.6 traz o diagrama de sequência que exemplifica o uso do piStarGODA.

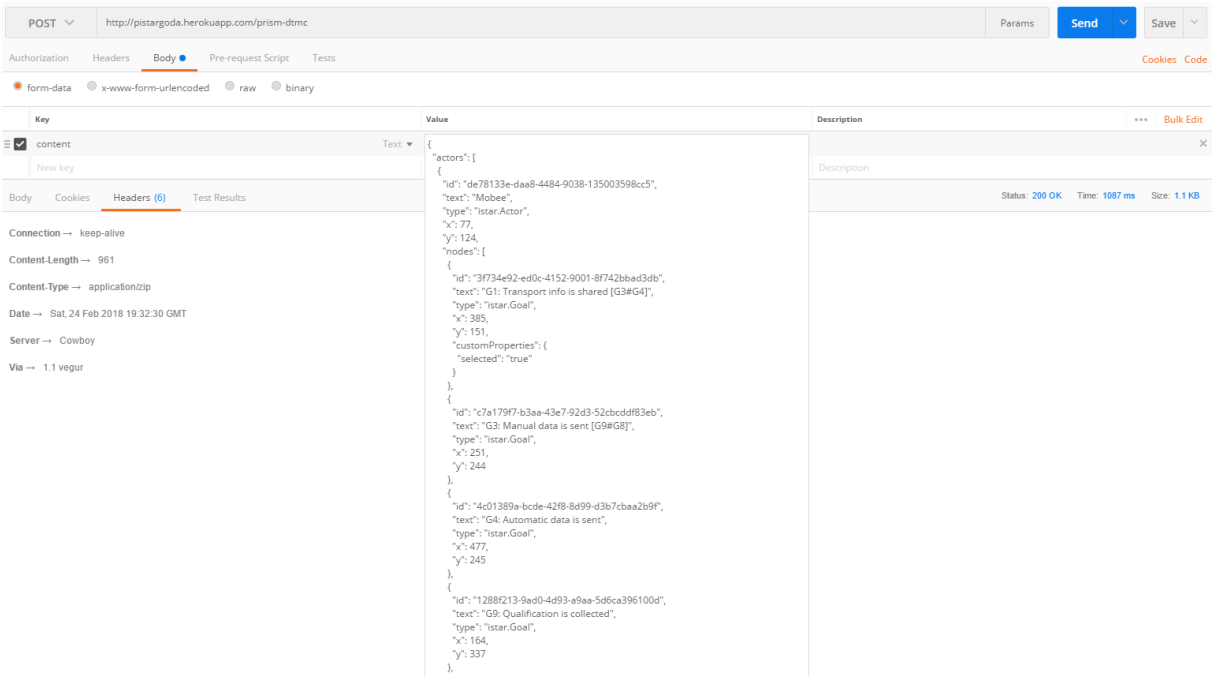

Figura 3.5: Execução do *backend* do piStar-GODA independentemente do *frontend*.

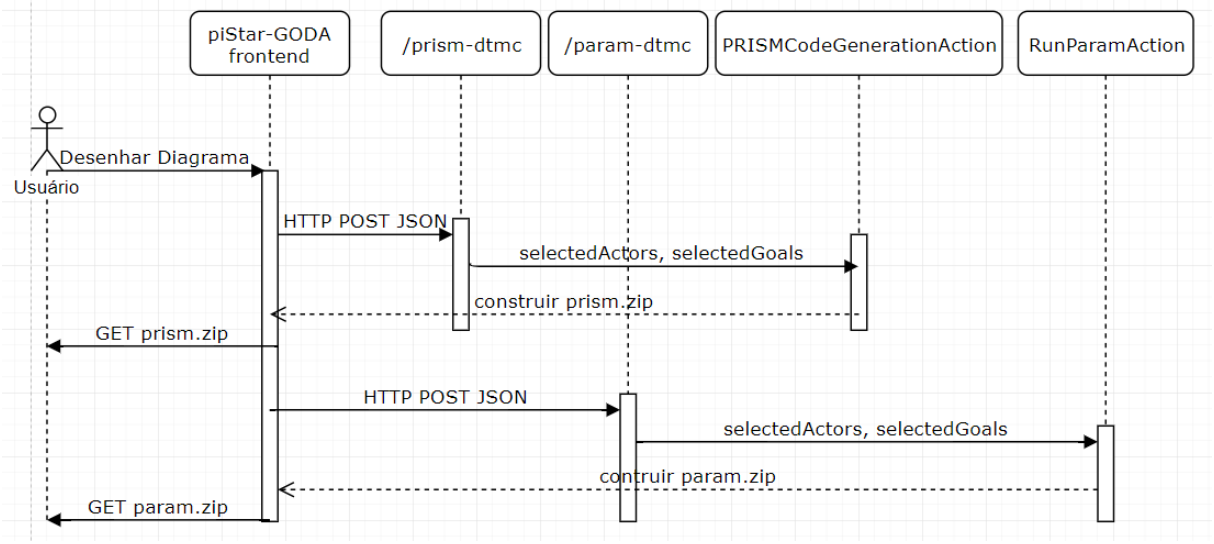

Figura 3.6: Diagrama de Sequência representando o uso do piStar-GODA.

O projeto funciona da seguinte forma, conforme Figura 3.6:

1. O usuário acessa o projeto a partir de um browser.
2. O usuário contrói um modelo.
3. O usuário clica em “Generate PRISM DTMC Code” caso queira gerar PRISM, ou em “Parametric Formula Generation” caso queira gerar PARAM.

4. O website transforma o objeto constrúido pelo usuário em um arquivo JSON.
5. O website faz uma requisição POST para o endereço "/prism-dtmc" ou "/param-dtmc", dependendo da escolha do usuário no passo 3, enviando como parâmetro o JSON criado a partir do modelo do usuário na forma de uma String.
6. A classe Controller, do módulo Integration, reconhece a requisição, e envia para o método "prism(String content)" ou "param(String content)", dependendo da escolha do usuário no passo 3.
7. A classe Controller faz o parse do JSON recebido como parâmetro, transformando-o em objetos Java de acordo com o especificado no pacote "piStar model".
8. A classe Controller transforma os objetos Java do pacote "piStar model" em objetos Java do pacote "piStarJ model".
9. A classe Controller instancia e executa a classe "PRISMCodeGenerationAction" ou "RunParamAction" do módulo "CRGMtoDTMC", dependendo da escolha do usuário no passo 3.
10. A execução ocorre conforme descrito no Referêncial Teórico, mas desta vez utilizando os objetos Java obtidos a partir da transformação feita pela classe "Controller", ao invés dos disponibilizados pelo plugin TAOM4E.
11. Ao final da execução, a browser disponibiliza para download o arquivo "prism.zip" ou "param.zip", dependendo da escolha do usuário no passo 3.

Como é possível ver na Figura 3.6, a utilização do projeto foi bastante simplificada em relação ao GODA versão Eclipse. O usuário desenha o diagrama no *frontend* da aplicação e seleciona uma opção entre PRISM e PARAM. Ao selecionar, o sistema faz uma requisição POST com o JSON do diagrama desenhado para o ponto de acesso correspondente à escolha. O ponto de acesso recebe a requisição, transforma as entidades e chama os métodos das classes "PRISMCodeGenerationAction" e "'RunParamAction", respectivamente. O arquivo zipado é construído e o *frontend* faz uma requisição GET para obter o arquivo correspondente à sua escolha.

Capítulo 4

Resultados Obtidos

4.1 Setup dos Testes

É necessário desenhar um diagrama no *frontend* para salvar o arquivo JSON com a representação do modelo. A Figura 4.1 traz um exemplo de diagrama utilizado para salvar o arquivo .txt para realização dos testes funcionais.

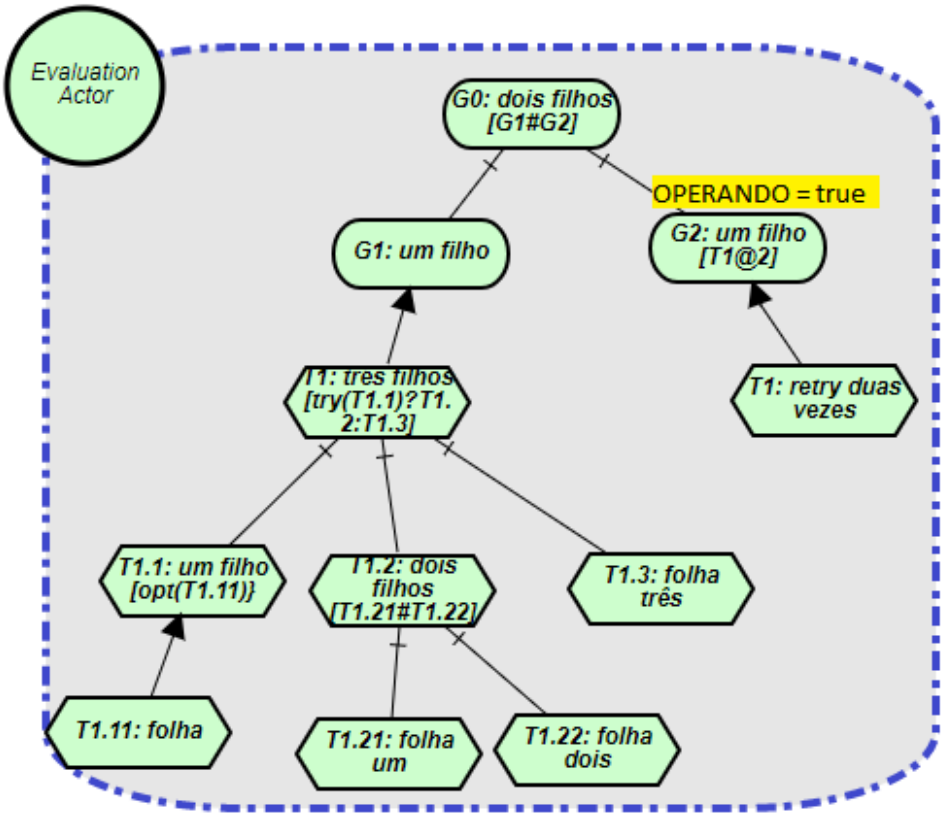

Figura 4.1: Exemplo de modelo desenhado no *frontend* do piStar-GODA.

Assim que salvo o arquivo, é possível executar o teste funcional, conforme código da Listing 4.1. A Figura II.1 traz o resultado do teste que utiliza o modelo mostrado na Figura 4.1.

A Listing 4.1 mostra o setup dos experimentos. O Spring Boot possui o JUnit como dependência, o que faz com que seja desnecessário sua declaração na lista de dependências. Cada método de teste carrega um arquivo .txt que contêm o JSON do modelo correspondente ao número do teste. Estes modelos são previamentes desenhados no frontend do piStar-GODA. Carregado este arquivo (linhas 15), é feita uma requisição POST para o ponto de acesso "/prism-dtmc" (linha 17 e 18), que recebe a requisição e executa a parte PRISM do projeto. Caso seja esperado que o teste tenha sucesso, a execução não deve gerar exceção, como mostrado no *testCase1* (linha 20). Caso contrário, é esperada uma exceção, conforme explicitado no *testCase3* (linha 32).

```
1  @RunWith(SpringRunner.class)
2  @WebMvcTest
3  @FixMethodOrder(MethodSorters.NAME_ASCENDING)
4  public class ApplicationTest {
5
6      @Autowired
7      private MockMvc mockMvc;
8
9      private String getContent(String path) throws IOException {
10         return new String(Files.readAllBytes(Paths.get("src/main/resources/testFiles/" + path)));
11     }
12
13     @Test
14     public void testCase1() throws Exception {
15         String content = getContent("Test1.txt");
16         try {
17             mockMvc.perform(post("/prism-dtmc").param("content", content))
18                     .andExpect(status().isOk());
19         } catch (Exception e) {
20             Assert.fail();
21         }
22     }
23     <...>
24     @Test
25     public void testCase3() throws Exception {
26         String content = getContent("Test3.txt");
27         try {
28             mockMvc.perform(post("/prism-dtmc").param("content", content))
29                     .andExpect(status().isOk());
30             Assert.fail();
31         } catch (Exception e) {
32             Assert.assertTrue(true);
33         }
34     }
35     <...>
36 }
```

Listing 4.1: Setup dos experimentos.

Os testes funcionais utilizados para testar a corretude da solução foram os mesmos apresentados em [4]. Essa decisão foi tomada pelo fato de que, com a integração completa, o projeto deve se comportar exatamente da mesma forma que o GODA se comporta com o plugin TAOM4E. Os resultados dos testes funcionais são apresentados no anexo 2.

A tabela 4.1 traz uma breve descrição para cada classe de equivalência definida por [4]. Além disso, a tabela traz os casos de teste que pertencem a determinada classe de equivalência.

4.2 Resultados

A tabela 4.2 traz os resultados dos testes, considerando as classes de equivalência definidas na tabela 4.1. A tabela inclui os tempos de execução para cada caso de teste em parênteses. Os testes apresentados em preto obtiveram o mesmo resultado que [4], e os apresentados em azul tiveram resultado divergente.

Comparando os resultados dos testes do piStar-GODA com os testes realizados em [4], foi possível concluir que mais testes passaram no piStar-GODA. Os casos de teste 3, 6, 9, 32, 33, 34, e 35, que estão apresentados em azul, falharam em [4], mas passaram neste trabalho. Isto indica que o código foi evoluído desde que o trabalho [4] foi realizado, o que é positivo para o projeto já que indica um desenvolvimento constante.

Observando os testes que passaram neste trabalho, pode-se concluir que os principais avanços observados na correção de bugs do GODA é relacionado às classes de equivalências LabelPatternTest, OneChildTest, OperandOperatorTest, e ContextAnnotationValueTest.

É possível verificar que nenhum dos testes apresentou tempo de execução superior ao esperado, conforme comparação com [4]. A Figura 4.2 mostra os resultados utilizados para preeencher a tabela de resultados.

4.3 Lições Aprendidas e Dificuldades Encontradas

A principal lição aprendida neste trabalho foi no que tange à utilização de novas tecnologias para realizar a integração de dois projetos diferentes. O piStar é feito basicamente em Javascript, enquanto que o GODA é composto de uma combinação de código Java integrado com bibliotecas diversas e um plugin do Eclipse, o que aumenta a complexidade da integração.

Foi necessário desenvolver a habilidade de modelar a integração antes de implementar o código, o que exigiu uma visão de alto nível do projeto. Feita a modelagem, foi necessário conhecer os detalhes do código do GODA para implementar a solução. Também

Tabela 4.1: Classes de equivalência e respectivos casos de teste

Classe de Equivalência	**Descrição**	**Casos de Teste**
CorrectModelTest	Válida, com anotações totalmente de acordo.	1, 2
LabelPatternTest	Inválida, com objetivos e/ou tarefas e/ou anotações de contexto escritos de forma incorreta no ambiente de modelagem, segundo a especificação do framework.	3, 4, 5, 6
NoChildTest	Inválida, sem nós filhos mas, com anotação de tempo de execução em sua escrita.	7, 8
OneChildTest	Inválida, com apenas um nó filho, com anotação de tempo de execução não permitida.	9, 10
MultipleChildrenTest	Inválida, com dois ou mais nós filhos sem anotação de tempo de execução.	11, 12, 13, 14
SameIdentifierGoalTest	Inválida, com dois ou mais objetivos que contenham o mesmo identificador G#.	15, 16
SameIdentifierTaskTest	Inválida, com duas ou mais tarefas irmãs com o mesmo identificador T#.	17, 18
MultipleLevelTaskIdentifierTask	Inválida, com tarefas com a notação dos identificadores inválida.	19, 20
NoRuntimeAnnotationTest	Inválida, com anotação escrita fora de colchetes ou antes da descrição.	21, 22
RuntimeAnnotationReferenceTest	Inválida, com anotações referenciando nós que não são filhos.	23, 24
RuntimeAnnotationPatternTest	Inválida, com anotações de tempo de execução escritas fora do padrão.	25, 26, 27, 28, 29, 30, 31
OperandOperatorTest	Inválida, com operandos e/ou operadores PRISM inválidos.	32, 33
ContextAnnotationValueTest	Inválida, com anotações de contexto contendo valores que não sejam int, boolean ou double.	34, 35

Tabela 4.2: Resultados dos testes funcionais

Classe de Equivalência	**Esperado**	**Não Esperado**
CorrectModelTest	1 (147 ms)	2 (50 ms)
LabelPatternTest	3 (19 ms), 6 (28 ms)	4 (48 ms), 5 (51 ms)
NoChildTest	-	7 (86 ms), 8 (57 ms)
OneChildTest	9 (34 ms)	10 (138 ms)
MultipleChildrenTest	-	11 (365 ms), 12 (172 ms), 13 (148 ms), 14 (222 ms)
SameIdentifierGoalTest	-	15 (156 ms), 16 (176 ms)
SameIdentifierTaskTest	-	17 (111 ms), 18 (84 ms)
MultipleLevelTaskIdentifierTask	-	19 (123 ms), 20 (130 ms)
NoRuntimeAnnotationTest	-	21 (89 ms), 22 (62 ms)
RuntimeAnnotationReferenceTest	-	23 (32 ms), 24 (53 ms)
RuntimeAnnotationPatternTest	25 (50 ms), 26 (15 ms), 27 (15 ms), 28 (17 ms), 29 (25 ms), 30 (18 ms), 31 (68 ms)	-
OperandOperatorTest	32 (31 ms), 33 (53 ms)	-
ContextAnnotationValueTest	34 (18 ms), 35 (17 ms)	-

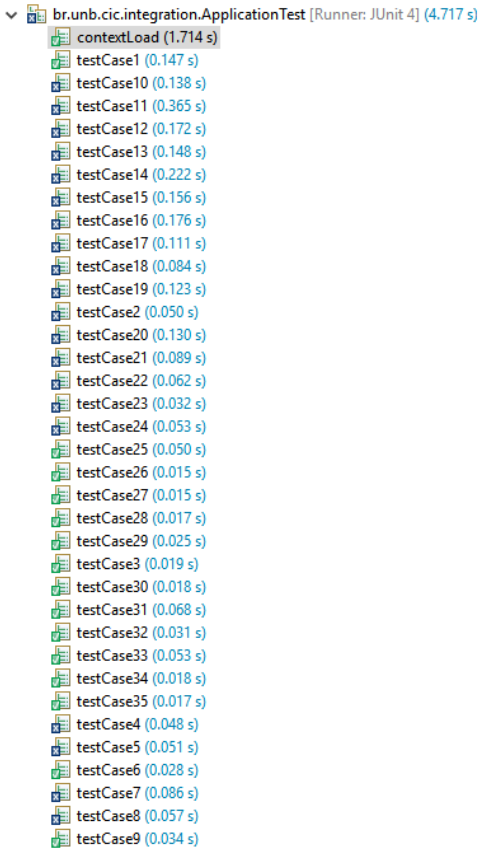

Figura 4.2: Tempos obtidos ao executar os testes funcionais.

foi possível a utilização de novas tecnologias para implementação da integração, sendo escolhido o Spring Boot para esta parte.

As dificuldades encontradas foram principalmente relativas à complexidade do código do GODA. Foi necessário substituir o plugin TAOM4E completamente, o que implica que todas as partes do código que utilizavam o plugin tiveram que ser estudadas e substituídas. O processo de implementação da solução que migrou o projeto monolítico para a nova solução demandou 3 meses de implementação.

Capítulo 5

Conclusão

O objetivo deste trabalho foi desenvolver uma solução para a integração entre o piStar e o GODA. O principal benefício desta solução foi a eliminação do plugin TAOM4E do GODA, que era de difícil configuração e não possuia mais suporte e nem desenvolvimento de novas funcionalidades. Também houve aumento da dependabilidade do projeto, como explicado a seguir.

Como objetivos auxiliares, buscou-se aumentar a disponibilidade, confiabilidade, manutenibilidade do projeto. Buscou-se alcançar estes objetivos através do conceito de microsserviço, que divide os módulos em serviços. A utilização de tecnologias modernas também auxiliou na melhoria destes aspectos. A principal vantagem do projeto é que é possível executar o frontend sem a necessidade do backend, e vice-versa, conforme demonstrado anteriormente.

A solução foi implementada utilizando Java 8, Apache Maven e Spring Boot. Estas tecnologias são modernas e possuem suporte. O projeto pode ser importado e executado na máquina dos usuários, ou hospedado em servidores na nuvem para acesso na internet. Não é necessário nenhuma configuração adicional para a execução do projeto, bastanto importá-lo e executá-lo na IDE escolhida pelo desenvolvedor. Considera-se que o trabalho obteve sucesso, apesar de todas as dificuldades encontradas.

As limitações deste trabalho estão relacionadas ao código fonte da transformação de modelos realizadas pelo GODA. Como a parte principal do trabalho se tratou de fazer a interface entre os projetos piStar e GODA, o projeto GODA continua se comportando da mesma maneira, o que significa que caso haja defeitos no projeto GODA, o piStar-GODA herdará estes defeitos. Este aspecto é observado nos resultados dos testes, já que a maioria dos testes que falhavam em [4] continuam falhando neste trabalho.

Para trabalhos futuros, é sugerida a implementação do elemento “Quality”, que está disponível no piStar e é equivalente ao “Soft Goal” do plugin TAOM4E. Além disto, é possível dividir o backend em ainda mais módulos, fazendo um microserviço para cada

módulo. Esta divisão é uma opção para aumentar a disponibilidade do projeto, mas é uma sugestão que deve ser discutida quando o piStar-GODA possuir fluxo de requisições elevada, caso em que esta divisão trará benefícios maiores.

Também há a necessidade de realizar testes unitários para o ponto de acesso PARAM, pois apenas o ponto de acesso PRISM foi testado, a partir dos mesmo testes realizados em [4]. É desejável que os teste unitários comparem o projeto atual do GODA com o piStar-GODA, pois as comparações foram feitas baseadas nos testes unitários de [4], e não em testes refeitos com o projeto atual.

Referências

[1] Dalpiaz, Fabiano, Alexander Borgida, Jennifer Horkoff e John Mylopoulos: *Runtime goal models: Keynote*. Em *IEEE 7th International Conference on Research Challenges in Information Science, RCIS 2013, Paris, France, May 29-31, 2013*, páginas 1–11, 2013. https://doi.org/10.1109/RCIS.2013.6577674. xii, 7, 9

[2] Morandini, M, D.C. Nguyen, A. Perini, A.;Siena e A. Susi: *Tool-supported development with tropos: The conference management system case study*. Volume 4951, páginas 182–196. Springer, Springer, 2008, ISBN 978-3-540-79487-5. 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 2007. 1, 6, 8

[3] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente Manuel Mazzara Fabrizio Montesi Ruslan Mustafin Larisa Safina: *Microservices: yesterday, today, and tomorrow*. 2017. https://arxiv.org/pdf/1606.04036.pdf. 2, 17

[4] Solano, Gabriela Félix: *Verificando a corretude da transformação de modelos no goda*. 2016. 4, 25, 40, 44, 45, 48

[5] Ali, Raian, Fabiano Dalpiaz e Paolo Giorgini: *A goal-based framework for contextual requirements modeling and analysis*. Requir. Eng., 15(4):439–458, novembro 2010, ISSN 0947-3602. http://dx.doi.org/10.1007/s00766-010-0110-z. 6, 7

[6] Mylopoulos, John, Lawrence Chung e Eric Yu: *From object-oriented to goal-oriented requirements analysis*. Commun. ACM, 42(1):31–37, janeiro 1999, ISSN 0001-0782. http://doi.acm.org/10.1145/291469.293165. 6

[7] Avizienis, Algirdas, Jean Claude Laprie, Brian Randell e Carl Landwehr: *Basic concepts and taxonomy of dependable and secure computing*. IEEE Trans. Dependable Secur. Comput., 1(1):11–33, janeiro 2004, ISSN 1545-5971. http://dx.doi.org/10.1109/TDSC.2004.2. 6

[8] Mendonça, Danilo Filgueira, Genaína Nunes Rodrigues, Raian Ali, Vander Alves e Luciano Baresi: *Goda: A goal-oriented requirements engineering framework for runtime dependability analysis*. Inf. Softw. Technol., 80(C):245–264, dezembro 2016, ISSN 0950-5849. https://doi.org/10.1016/j.infsof.2016.09.005. 6, 8, 9

[9] Ghezzi, Carlo e Amir Molzam Sharifloo: *Model-based verification of quantitative non-functional properties for software product lines*. Inf. Softw. Technol., 55(3):508–524, março 2013, ISSN 0950-5849. http://dx.doi.org/10.1016/j.infsof.2012.07.017. 7

[10] Mendonça, Danilo Filgueira: *Dependability verification for contextual/runtime goal modelling*. Master's thesis, 2015. 7, 8, 10, 11

[11] Dardenne, Anne, Axel van Lamsweerde e Stephen Fickas: *Goal-directed requirements acquisition*. Sci. Comput. Program., 20(1-2):3–50, abril 1993, ISSN 0167-6423. http://dx.doi.org/10.1016/0167-6423(93)90021-G. 7

[12] Yu, Eric Siu Kwong: *Modelling Strategic Relationships for Process Reengineering*. Tese de Doutoramento, Toronto, Ont., Canada, Canada, 1995, ISBN 0-612-02887-9. AAINN02887. 7

[13] Bresciani, Paolo, Anna Perini, Paolo Giorgini, Fausto Giunchiglia e John Mylopoulos: *Tropos: An agent-oriented software development methodology*. Autonomous Agents and Multi-Agent Systems, 8(3):203–236, maio 2004, ISSN 1387-2532. https://doi.org/10.1023/B:AGNT.0000018806.20944.ef. 7

[14] Bob, Uncle: *The principles of ood*. 2005. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. 16

[15] Postdot Technologies, Inc: *Postman api development environment*. https://www.getpostman.com/. 17, 32

[16] James Lewis, Martin Fowler: *Microservices*. https://martinfowler.com/articles/microservices.html. 19

[17] *Gson - a java serialization/deserialization library to convert java objects into json and back*. https://github.com/google/gson. 21, 25, 26

Anexo I

Testes Funcionais

Os testes a seguir foram propostos por [4], sendo os modelos desenhados no piStar-GODA.

1.
   - Entrada (Figura I.1): Modelo correto.
   - Saída esperada: Execução finalizada com sucesso.

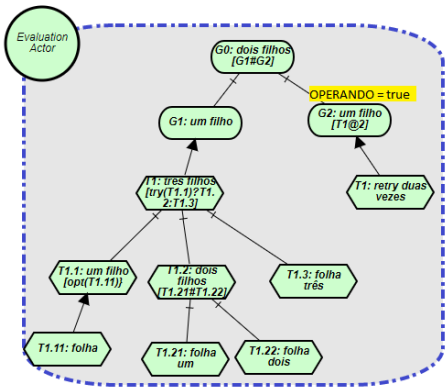

Figura I.1: Teste 1.

2.
   - Entrada (Figura I.2): Modelo Correto.
   - Saída esperada: Execução finalizada com sucesso.
3.
   - Entrada (Figura I.3): Objetivo com label fora do padrão.
   - Saída esperada: Execução interrompida devido a exceção.
4.
   - Entrada (Figura I.4): Tarefa com label fora do padrão.
   - Saída esperada: Execução interrompida devido a exceção.
5.
   - Entrada (Figura I.5): Tarefa com label fora do padrão.
   - Saída esperada: Execução interrompida devido a exceção.

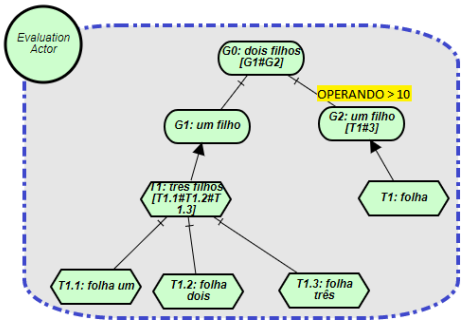

Figura I.2: Teste 2.

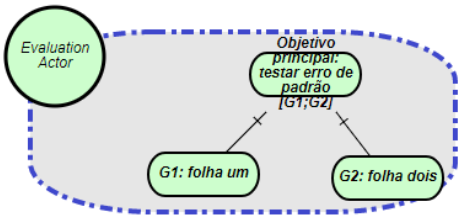

Figura I.3: Teste 3.

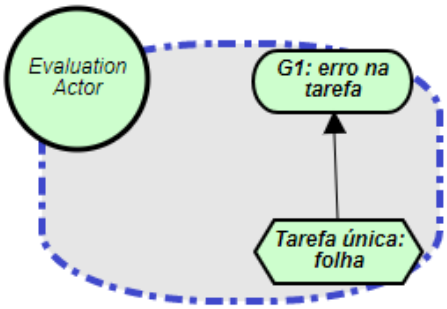

Figura I.4: Teste 4.

6.
   - Entrada (Figura I.6): Anotação de contexto com label fora do padrão.
   - Saída esperada: Execução interrompida devido a exceção.
7.
   - Entrada (Figura I.7): Objetivo sem nó filho e com anotação de tempo de execução.
   - Saída esperada: Execução interrompida devido a exceção.
8.
   - Entrada (Figura I.8): Tarefa sem nó filho e com anotação de tempo de execução.

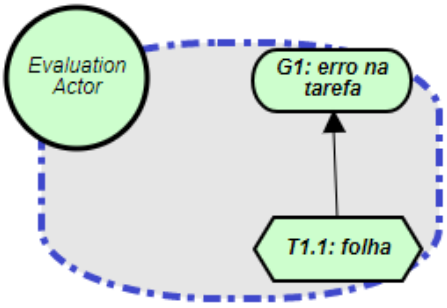

Figura I.5: Teste 5.

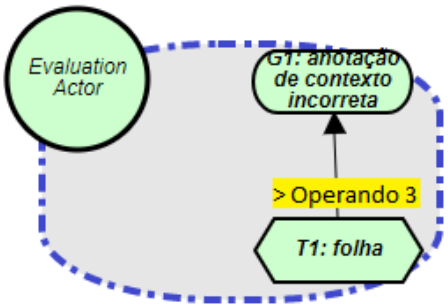

Figura I.6: Teste 6.

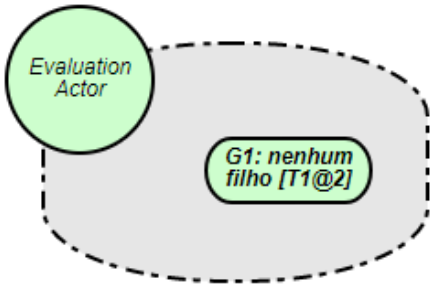

Figura I.7: Teste 7.

- Saída esperada: Execução interrompida devido a exceção.

9.
   - Entrada (Figura I.9): Objetivo com um nó filho, apresentando anotação de tempo de execução inválida.
   - Saída esperada: Execução interrompida devido a exceção.

10.
    - Entrada (Figura I.10): Tarefa com um nó filho, apresentando anotação de tempo de execução inválida.

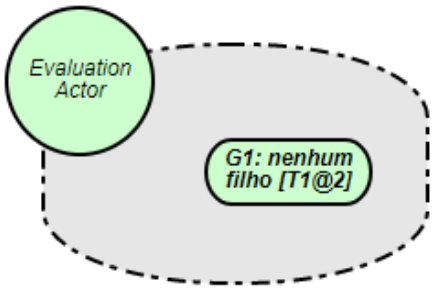

Figura I.8: Teste 8.

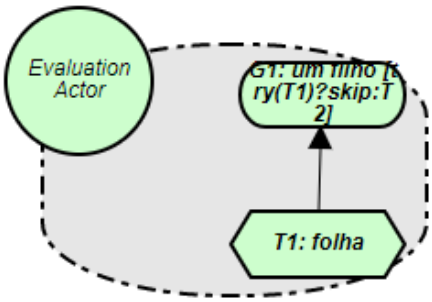

Figura I.9: Teste 9.

- Saída esperada: Execução interrompida devido a exceção.

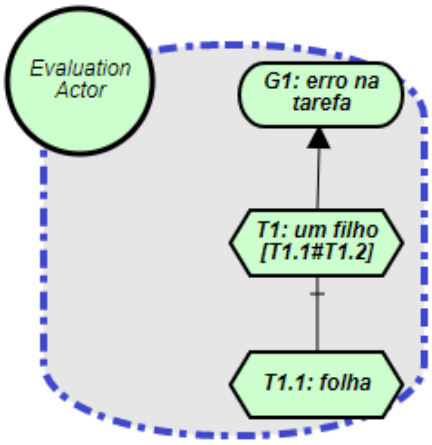

Figura I.10: Teste 10.

11.
- Entrada (Figura I.11): Objetivo com dois nós filhos, sem anotação de tempo de execução.
- Saída esperada: Execução interrompida devido a exceção.

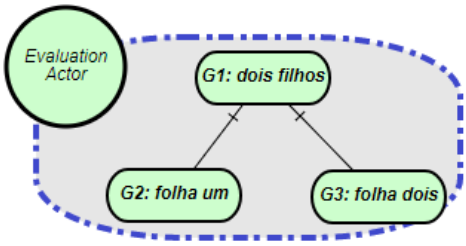

Figura I.11: Teste 11.

12.
- Entrada (Figura I.12): Objetivo com quatro nós filhos, sem anotação de tempo de execução.
- Saída esperada: Execução interrompida devido a exceção.

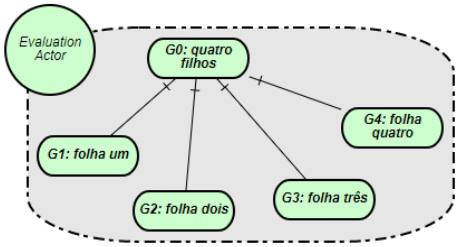

Figura I.12: Teste 12.

13.
- Entrada (Figura I.13): Tarefa com dois nós filhos, sem anotação de tempo de execução.
- Saída esperada: Execução interrompida devido a exceção.

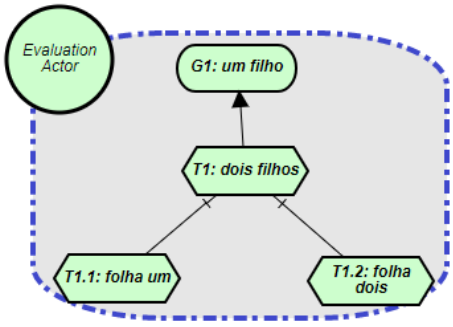

Figura I.13: Teste 13.

14.
- Entrada (Figura I.14): Tarefa com quatro nós filhos, sem anotação de tempo de execução.

- Saída esperada: Execução interrompida devido a exceção.

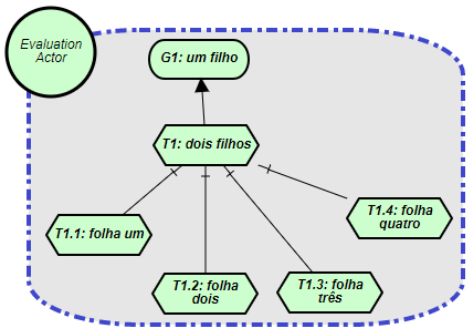

Figura I.14: Teste 14.

15.
- Entrada (Figura I.15): Dois objetivos tendo o mesmo identificador.
- Saída esperada: Execução interrompida devido a exceção.

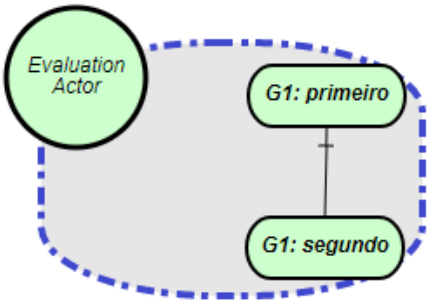

Figura I.15: Teste 15.

16.
- Entrada (Figura I.16): Quatro objetivos tendo o mesmo identificador.
- Saída esperada: Execução interrompida devido a exceção.

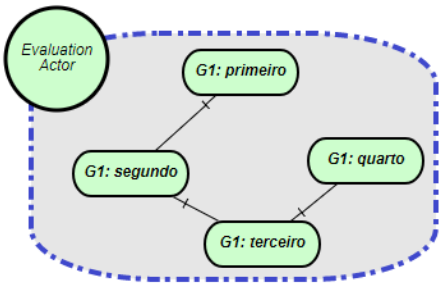

Figura I.16: Teste 16.

17.
- Entrada (Figura I.17): Duas tarefas irmãs com profundidade três e o mesmo identificador.
- Saída esperada: Execução interrompida devido a exceção.

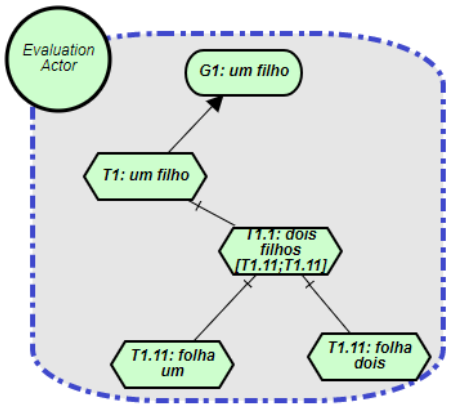

Figura I.17: Teste 17.

18.
- Entrada (Figura I.18): Três tarefas irmãs com profundidade dois e o mesmo identificador.
- Saída esperada: Execução interrompida devido a exceção.

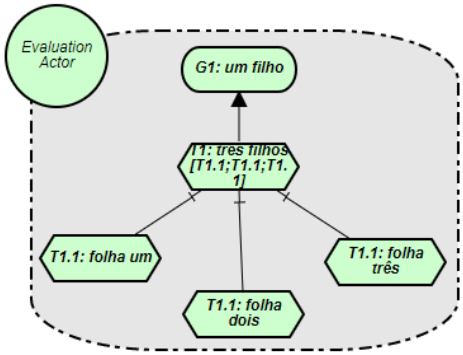

Figura I.18: Teste 18.

19.
- Entrada (Figura I.19): Tarefa com profundidade dois e identificador fora do padrão.
- Saída esperada: Execução interrompida devido a exceção.

20.
- Entrada (Figura I.20): Tarefa com profundidade três e identificador fora do padrão.
- Saída esperada: Execução interrompida devido a exceção.

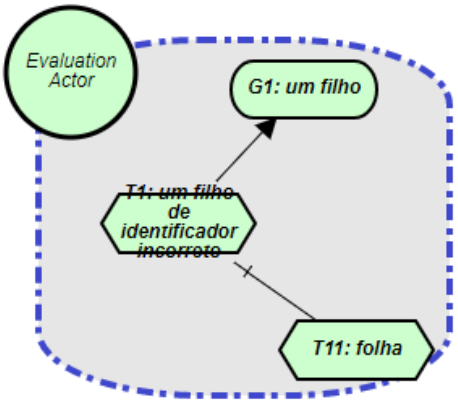

Figura I.19: Teste 19.

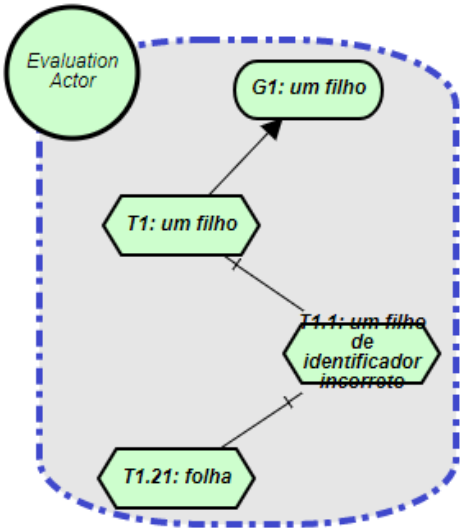

Figura I.20: Teste 20.

21.
- Entrada (Figura I.21): Anotação de tempo de execução fora de colchetes.
- Saída esperada: Execução interrompida devido a exceção.

22.
- Entrada (Figura I.22): Anotação de tempo de execução antes da descrição do objetivo ou tarefa.
- Saída esperada: Execução interrompida devido a exceção.

23.
- Entrada (Figura I.23): Anotação de tempo de execução referenciando um nó que não seja filho.
- Saída esperada: Execução interrompida devido a exceção.

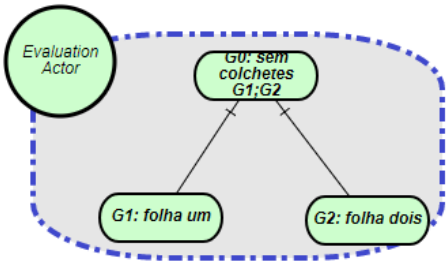

Figura I.21: Teste 21.

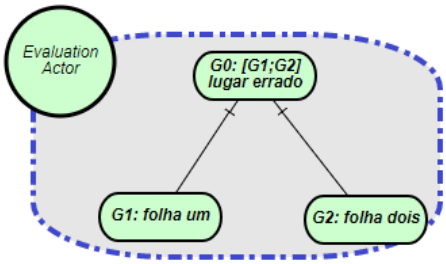

Figura I.22: Teste 22.

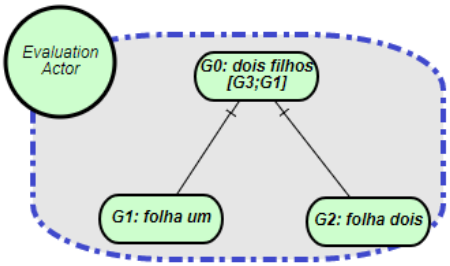

Figura I.23: Teste 23.

24.
- Entrada (Figura I.24): Anotação de tempo de execução referenciando mais de um nó que não seja filho.
- Saída esperada: Execução interrompida devido a exceção.

25.
- Entrada (Figura I.25): Expressão "[E1;E2]", que não segue o padrão.
- Saída esperada: Execução interrompida devido a exceção.

26.
- Entrada (Figura I.26): Expressões "[E1#E2]" e "[E#n]", que não seguem o padrão.
- Saída esperada: Execução interrompida devido a exceção.

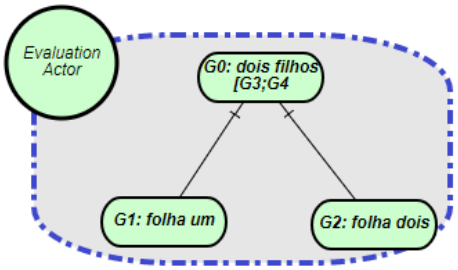

Figura I.24: Teste 24.

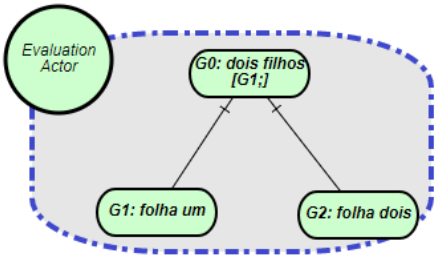

Figura I.25: Teste 25.

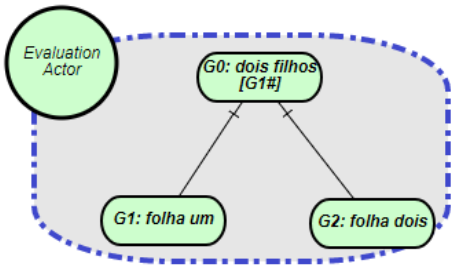

Figura I.26: Teste 26.

27.
- Entrada (Figura I.27): Expressão “[E+n]”, que não segue o padrão.
- Saída esperada: Execução interrompida devido a exceção.

28.
- Entrada (Figura I.28): Expressão “[E@n]”, que não segue o padrão.
- Saída esperada: Execução interrompida devido a exceção.

29.
- Entrada (Figura I.29): Expressão “[E1|E2]”, que não segue o padrão.
- Saída esperada: Execução interrompida devido a exceção.

30.
- Entrada (Figura I.30): Expressão “[opt(E)]”, que não segue o padrão.

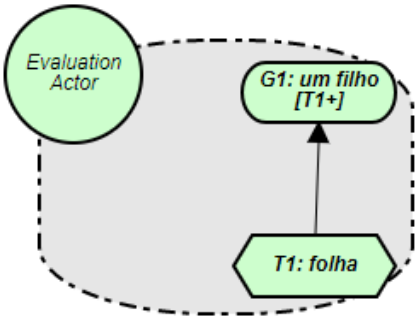

Figura I.27: Teste 27.

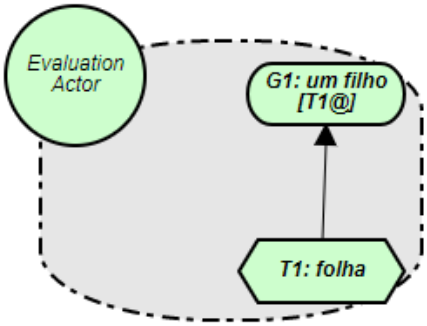

Figura I.28: Teste 28.

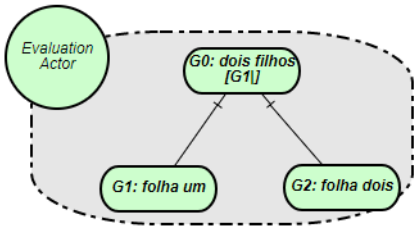

Figura I.29: Teste 29.

- Saída esperada: Execução interrompida devido a exceção.

31.
- Entrada (Figura I.31): Expressão "[try(E)?E1:E2]", que não segue o padrão.
- Saída esperada: Execução interrompida devido a exceção.

32.
- Entrada (Figura I.32): Operando de anotação de contexto que se inicia com dígito.

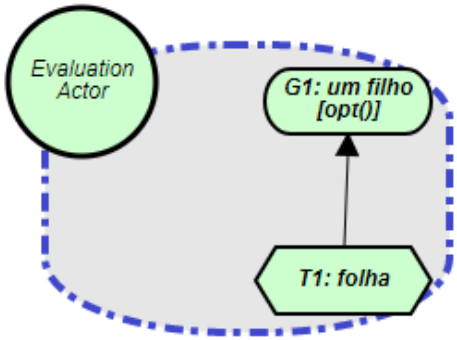

Figura I.30: Teste 30.

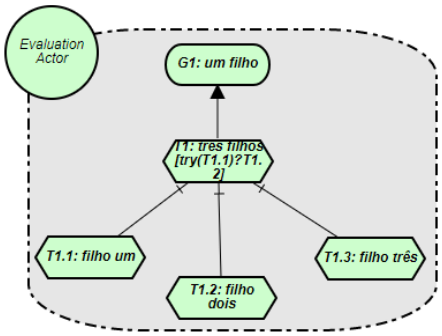

Figura I.31: Teste 31.

- Saída esperada: Execução interrompida devido a exceção.

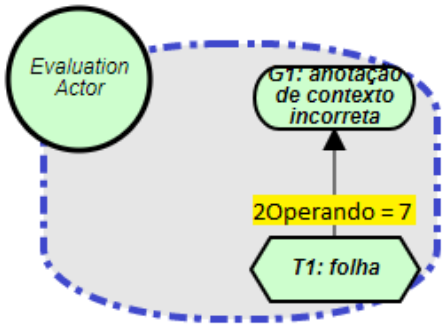

Figura I.32: Teste 32.

33.
- Entrada (Figura I.33): Operador PRISM, de anotação de contexto, que é inválido.
- Saída esperada: Execução interrompida devido a exceção.

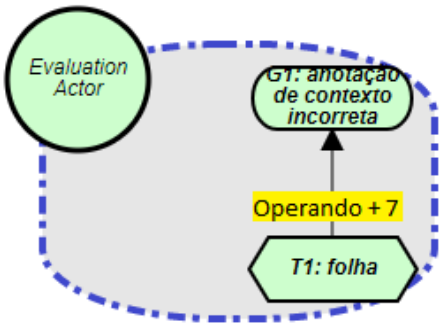

Figura I.33: Teste 33.

34.
- Entrada (Figura I.34): Anotação de contexto com valor de comparação do tipo string.
- Saída esperada: Execução interrompida devido a exceção.

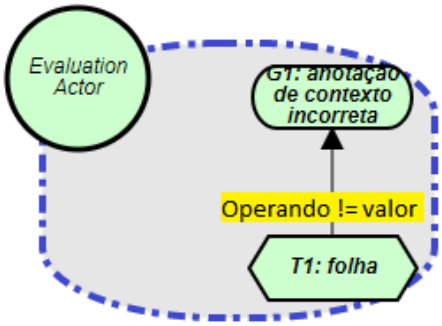

Figura I.34: Teste 34.

35.
- Entrada (Figura I.35): Anotação de contexto contendo valor de comparação do tipo char.
- Saída esperada: Execução interrompida devido a exceção.

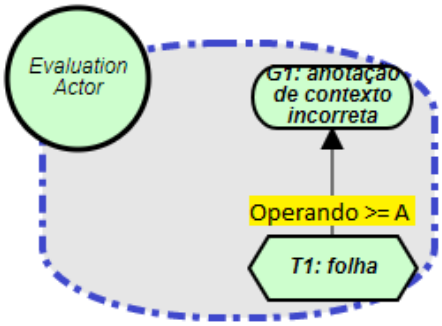

Figura I.35: Teste 35.

Anexo II

Resultados dos Testes Funcionais

A execuçao de cada teste unitário é apresentada nas figuras abaixo. Os testes foram realizados individualmente, e portando cada tempo de execução engloba também o tempo necessário para inicializar a aplicação.

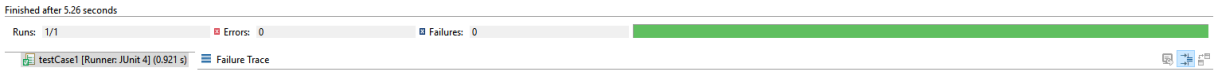

Figura II.1: Resultado do caso de teste 1.

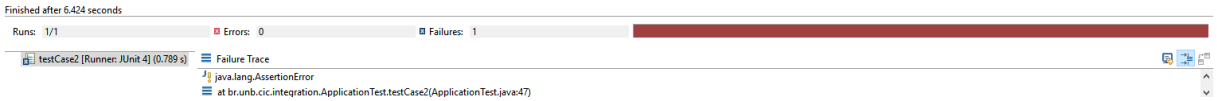

Figura II.2: Resultado do caso de teste 2.

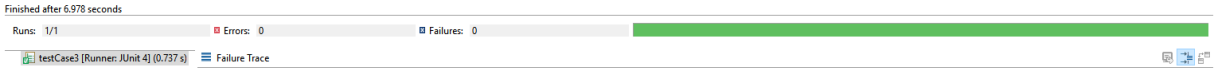

Figura II.3: Resultado do caso de teste 3.

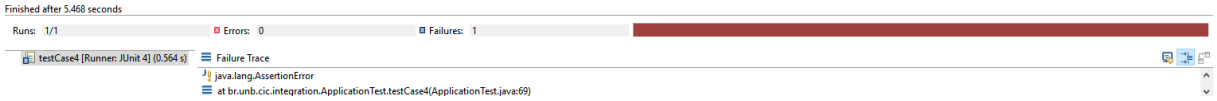

Figura II.4: Resultado do caso de teste 4.

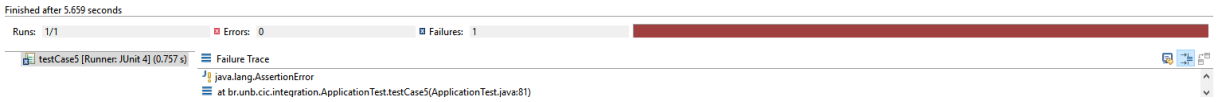

Figura II.5: Resultado do caso de teste 5.

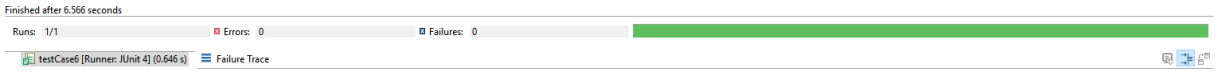

Figura II.6: Resultado do caso de teste 6.

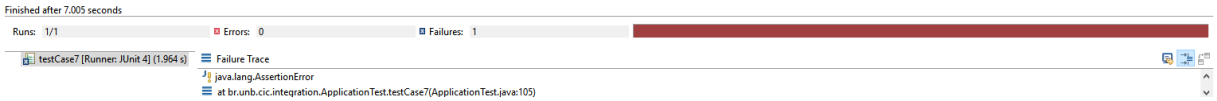

Figura II.7: Resultado do caso de teste 7.

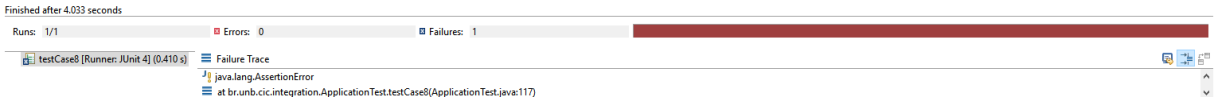

Figura II.8: Resultado do caso de teste 8.

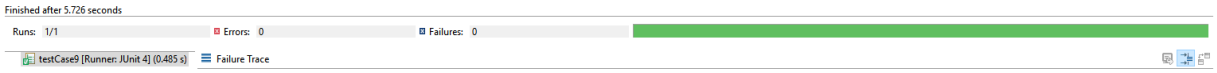

Figura II.9: Resultado do caso de teste 9.

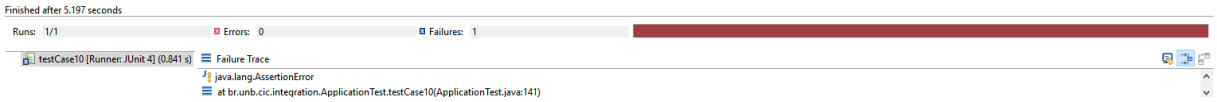

Figura II.10: Resultado do caso de teste 10.

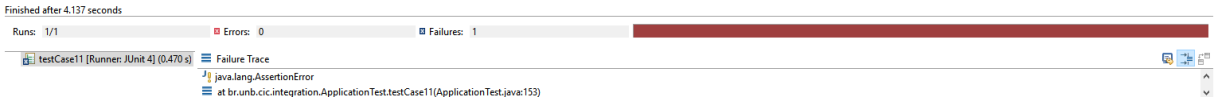

Figura II.11: Resultado do caso de teste 11.

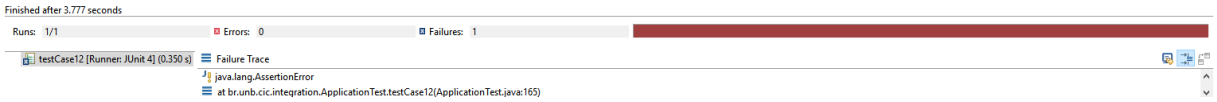

Figura II.12: Resultado do caso de teste 12.

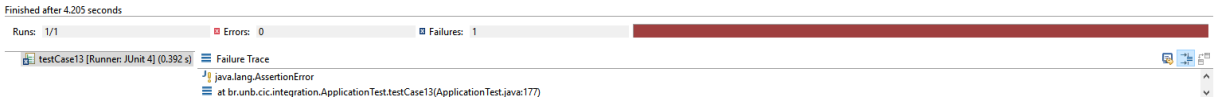

Figura II.13: Resultado do caso de teste 13.

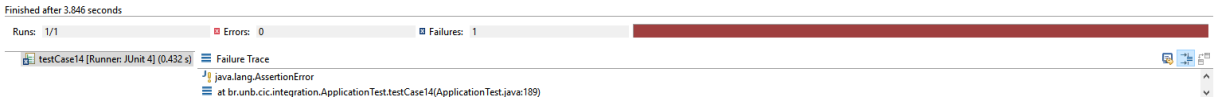

Figura II.14: Resultado do caso de teste 14.

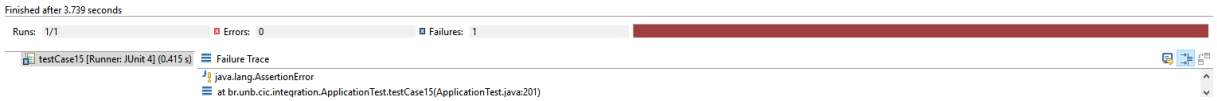

Figura II.15: Resultado do caso de teste 15.

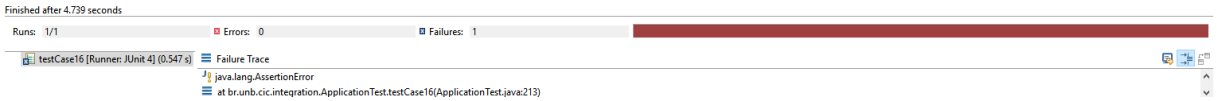

Figura II.16: Resultado do caso de teste 16.

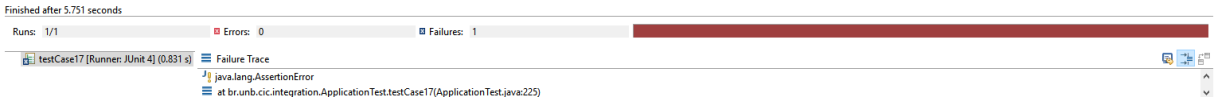

Figura II.17: Resultado do caso de teste 17.

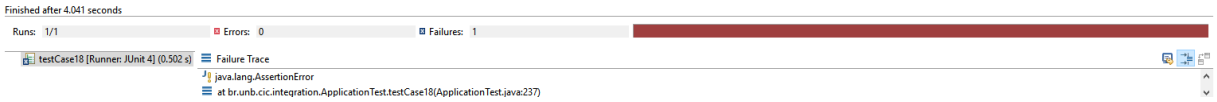

Figura II.18: Resultado do caso de teste 18.

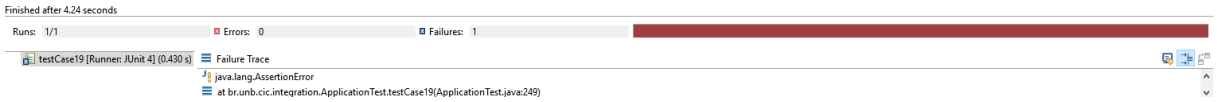

Figura II.19: Resultado do caso de teste 19.

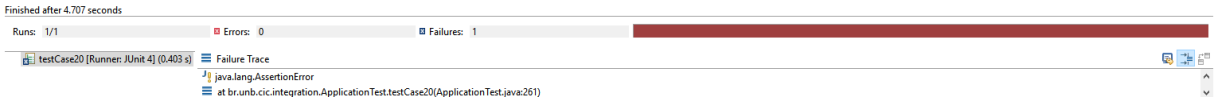

Figura II.20: Resultado do caso de teste 20.

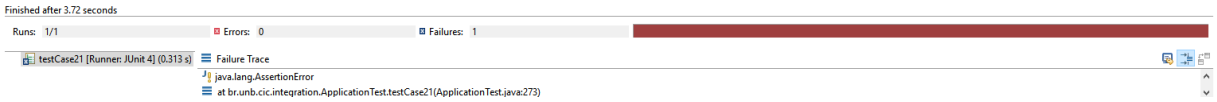

Figura II.21: Resultado do caso de teste 21.

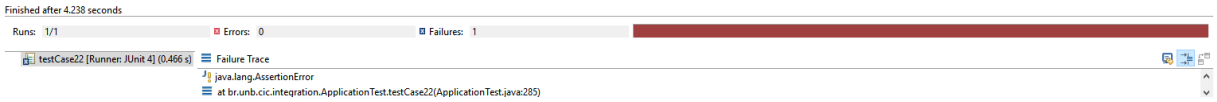

Figura II.22: Resultado do caso de teste 22.

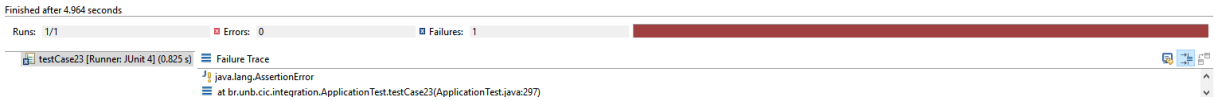

Figura II.23: Resultado do caso de teste 23.

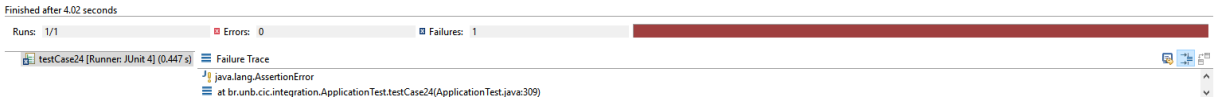

Figura II.24: Resultado do caso de teste 24.

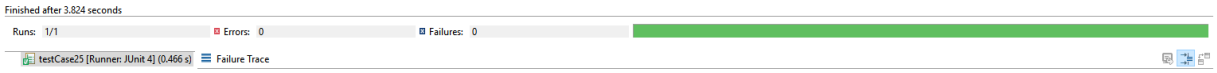

Figura II.25: Resultado do caso de teste 25.

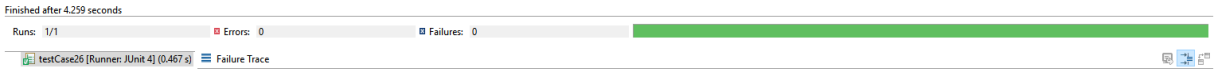

Figura II.26: Resultado do caso de teste 26.

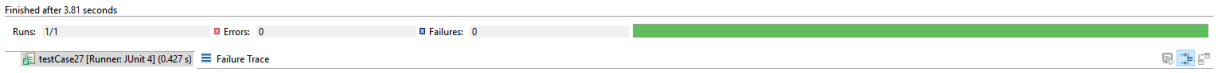

Figura II.27: Resultado do caso de teste 27.

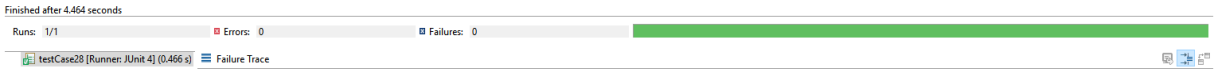

Figura II.28: Resultado do caso de teste 28.

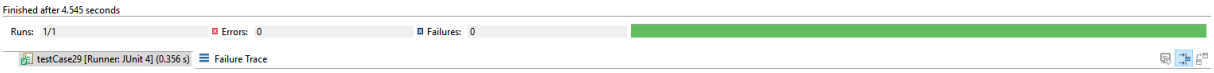

Figura II.29: Resultado do caso de teste 29.

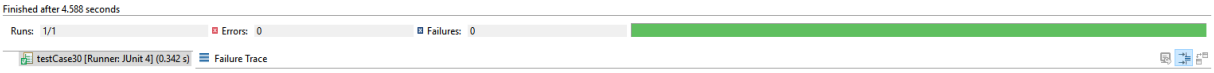

Figura II.30: Resultado do caso de teste 30.

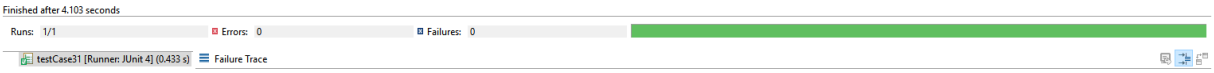

Figura II.31: Resultado do caso de teste 31.

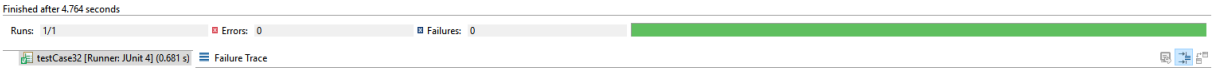

Figura II.32: Resultado do caso de teste 32.

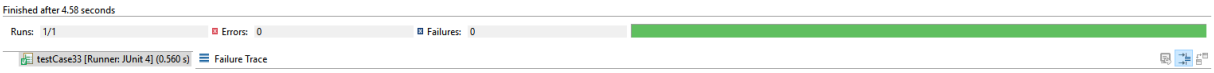

Figura II.33: Resultado do caso de teste 33.

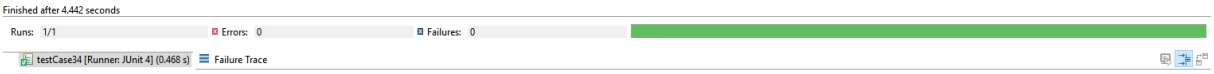

Figura II.34: Resultado do caso de teste 34.

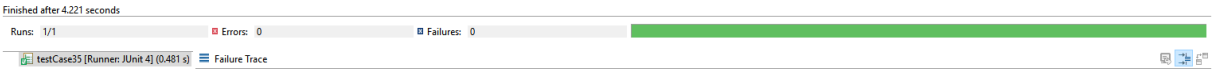

Figura II.35: Resultado do caso de teste 35.