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

Verificando a Corretude da Transformação de Modelos no GODA

Gabriela Félix Solano

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

Orientadora
Prof.ª Dr.ª Genaína Nunes Rodrigues

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientadora) — CIC/UnB
Prof. Dr. Rodrigo Bonifácio de Almeida — CIC/UnB
Prof. Dr. Vander Ramos Alves — CIC/UnB

CIP — Catalogação Internacional na Publicação

Solano, Gabriela Félix.

Verificando a Corretude da Transformação de Modelos no GODA / Gabriela Félix Solano. Brasília : UnB, 2016.
131 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. GODA, 2. modelo orientado a objetivos, 3. CRGM, 4. teste funcional

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

Verificando a Corretude da Transformação de Modelos no GODA

Gabriela Félix Solano

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

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

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Dr. Vander Ramos Alves
CIC/UnB

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

Brasília, 07 de julho de 2016

Dedicatória

Este trabalho é dedicado à minha família, que sempre me apoiou durante toda a minha formação.

Agradecimentos

Agradeço a Deus em primeiro lugar, por todas as bênçãos e proteção que Ele me concede diariamente.

Agradeço aos meus pais, Jairo e Silvânia, que sempre foram minha base, e nunca mediram esforços para que eu pudesse alcançar meus objetivos. Às minhas irmãs e melhores amigas, Júlia e Karolina, por estarem ao meu lado em todos os momentos, serem minhas parceiras fiéis para o que der e vier. Ao meu cachorro Bob, por toda alegria, animação e amor com que me conforta todos os dias. À todos os meus familiares, pelo suporte e torcida durante essa jornada.

Agradeço à minha orientadora, Prof.ª Dr.ª Genaína Nunes Rodrigues, por ter aceitado o desafio deste trabalho, me ensinando e guiando nessa etapa da graduação. À todos os professores com quem tive a oportunidade de aprender e trabalhar na Universidade de Brasília. Ao Gabriel Rodrigues, pela boa vontade e ajuda que foi fundamental para o desenvolvimento do trabalho.

Agradeço aos amigos que a universidade me deu, que sempre contribuíram para o meu crescimento e fizeram da graduação uma época especial e divertida.

Por fim, agradeço ao programa Ciência sem Fronteiras, pela melhor experiência da minha vida. Ao amigos que fiz no exterior, em especial Will, Isa e Fernando, pelo carinho que tiveram ao longo desse período.

Resumo

O *framework* GODA (*Goal-Oriented Dependability Analysis*) realiza análise de dependabilidade em modelos orientados a objetivos. Uma etapa importante é o processo de geração automática de modelo DTMC (*Discrete-Time Markov Chains*) a partir de um modelo CRGM (*Contextual and Runtime Goal Model*). O modelo CRGM apresenta notações específicas não testadas. Erros neste modelo podem acarretar em problemas na geração do modelo DTMC. Como o GODA é integrado à diferentes ferramentas, a atividade de teste funcional consegue verificar a funcionalidade geral deste *framework*.

Esse trabalho teve como objetivo o desenvolvimento de uma suíte de testes que verifique a boa formação dos modelos CRGM. Para isso, foi escolhida a abordagem de teste funcional, utilizando o critério Teste Funcional Sistemático. A partir da especificação do programa, classes de equivalência foram definidas e, em seguidas, casos de teste foram identificados. A implementação dos testes foi feita utilizando a linguagem de programação Java, e o conjunto de testes foi automatizado utilizando a ferramenta JUnit.

Os resultados mostraram falhas na validação de anotações utilizadas no modelo CRGM. O desenvolvimento da suíte de testes proposta foi importante para expor problemas que podem acarretar numa geração de modelos DTMC incorretos, devido a erros no CRGM.

Palavras-chave: GODA, modelo orientado a objetivos, CRGM, teste funcional

Abstract

GODA (Goal-Oriented Dependability Analysis) framework performs dependability analysis on goal models. An important step is the CRGM (Contextual and Runtime Goal Model) to DTMC (Discrete-Time Markov Chains) automated code generation. CRGM has untested notations. Errors in this model could result in problems during the DTMC model generation. Since GODA integrates many different tools, functional testing activity can control the overall functionality of this framework.

The aim of this work was the development of a test suit that verifies well formedness of the CRGM model. The functional testing approach was chosen, using the Systematic Functional Testing criterion. From de software specification, equivalence classes were defined and then test cases were identified. The tests were implemented in Java, and automated using JUnit.

The results showed validation failures of CRGM notes. The development of the test suit proposed was important to expose problems that can lead to incorrect DTMC models due to errors in CRGM.

Keywords: GODA, goal model, CRGM, functional testing

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Modelos orientados a objetivos representam uma ontologia intencional usada nas fases iniciais de análise de requisitos para explicar o porquê de um sistema de *software* [1]. Eles têm sido utilizados para representar a racionalidade de tanto humanos quanto *software* [22], e oferecem construções úteis para analisar objetivos de alto nível e maneiras de os satisfazer. Essas caractísticas são essenciais para a análise e o projeto de um sistema, que deve refletir a racionalidade dos *stakeholders* e a adaptação à contextos variáveis [10] [23].

Dependabilidade é a capacidade de entregar um serviço que possa ser justificadamente confiável [2]. Tal confiança tem que considerar não apenas a corretude técnica do serviço, mas também a habilidade de unir as intenções dos *stakeholdesrs* com os interesses estratégicos. Uma definição alternativa que fornece o critério para decidir se um serviço é seguro é a de que a dependabilidade de um sistema é a habilidade de evitar falhas no serviço que são mais frequentes e mais severas do que é aceitável [2]. Ainda segundo [2], dependabilidade é um conceito integrado que engloba os seguintes atributos:

- 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 de sofrer modificações e raparações.

O GODA (*Goal-Oriented Dependability Analysis*) é um *framework* que realiza análise de dependabilidade em modelos orientados a objetivos. Ele captura a propriedade de satisfatibilidade de objetivos em tempo de execução de um sistema de *software*, onde podem haver diferentes contextos. Este *framework* foi proposto com o intuito de prover meios designados para administrar os requisitos de dependabilidade de sistemas operando em um contexto dinâmico [19].

O GODA integra os poderes de CGM (*Contextual Goal Model*), RGM (*Runtime Goal Model*) e PMC (*Probabilistic Model Checking*). Durante o planejamento do *software*, ele pode ajudar com decisões de projeto e implementação; durante a execução, ele ajuda o sistema a se auto-adaptar, analisando as diferentes alternativas e selecionando uma com a maior probabilidade de dependabilidade. O GODA é implementado estendendo

o TAOM4E [20]: uma ferramenta baseada no TROPOS incorporado no *plugin* de implementação do Eclipse. Além disso, ele integra diferentes ferramentas, como PRISM e PARAM.

Uma importante etapa do processo de funcionamento do GODA é a transformação de modelos CRGM (*Contextual an Runtime Goal Model*) em DTMC (*Discrete-Time Markov Chains*). Essa transformação é feita a partir de uma geração automática de um modelo DTMC em linguagens PRISM e PARAM a partir de um CRGM. Mais detalhes sobre o funcionamento do GODA e da sua transformação de modelos serão dados no próximo capítulo.

A motivação principal deste trabalho foi o fato do CRGM possuir notações específicas não testadas. Como o GODA é integrado a diferentes ferramentas, a atividade de teste é muito importante para garantir a confiabilidade de seus resultados. Para garantir uma correta transformação de modelos e, consequentemente, uma correta análise de dependabilidade feita pelo GODA. Portanto, é necessário verificar a corretude dos modelos CRGM utilizados.

1.1 Problema e Hipótese

O problema encontrado no GODA é que, no momento, não há meios de permitir ao usuário verificar se o modelo CRGM gerado está correto. Assim, a análise de dependabilidade do modelo pode não ser confiável.

A hipótese é a de que problemas na geração dos modelos PRISM e PARAM podem advir de erros relativos à boa formação da geração de modelos CRGM.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo principal deste trabalho é o desenvolvimento de suíte de testes funcionais para o modelo CRGM.

1.2.2 Objetivos Específicos

De forma a verificar a corretude do modelo CRGM, as metas estabelecidas foram:

- Planejamento de testes funcionais para erros relativos à boa formação da geração de modelos CRGM;
- Desenvolvimento dos testes funcionais planejados;
- Reportar os resultados obtidos com a execução dos testes funcionais.

1.3 Descrição dos Capítulos

Este trabalho se divide em mais quatro capítulos. O Capítulo 2 apresentará um referencial teórico dos temas abordados no trabalho. Será apresentado o funcionamento

do GODA, mais detalhes sobre seu processo de transformação de modelos, assim como uma introdução teórica sobre o tipo de teste a ser realizado. 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

2.1 Análise de Dependabilidade Orientada a Objetivos (GODA)

O método de análise de dependabilidade orientada a objetivos é baseado em [19]. A Figura 2.1 ilustra o processo do GODA, que inicia com um Modelo Orientado a Objetivos produzido através de modelagem orientada a objetivos convencional. Então, é inserida informação de contexto e informação de tempo de execução neste modelo, tornando-se um CRGM. Uma vez que o CRGM do sistema está completo, ele é automaticamente traduzido em um DTMC. Propriedades de dependabilidade são, então, apresentadas como fórmulas PCTL (*Probabilistic Computation Tree Logic*) [11], e a verificação do sistema pode ser realizada.

Setas tracejadas no processo indicam uma abordagem iterativa em que atividades de modelagem possam ser realizadas múltiplas vezes até uma especificação completa e coerente do sistema *to-be* ser atingida. Por exemplo, restrições de contexto podem acabar com nenhuma alternativa para atingir um determinado objetivo, o que deve ser o caso de uma nova iteração sobre a modelagem orientada a objetivos. Além disso, a análise em tempo de projeto pode revelar violações das restrições de dependabilidade para uma ou mais alternativas elicitadas, o que justifica a ligação tracejada entre a análise de dependabilidade e a modelagem do sistema.

2.1.1 Modelagem Orientada a Objetivos e Contexto

Modelagem orientada a objetivos fornece um meio de analisar os diversos requisitos de diferentes *stakeholders* em um sistema de *software* [8] [24] [5]. Objetivos pertencem à atores, e atores inter-dependem uns dos outros para alcançar seus objetivos. Objetivos são realizados por tarefas-folha, que denotam processos executáveis por um ator. Em um modelo orientado a objetivos estruturado em árvores, objetivos são decompostos em sub-objetivos ou refinados por uma tarefa *means-end*; enquanto que tarefas não-folhas são decompostas em outras sub-tarefas. Uma decomposição E requer que todos os sub-nós sejam satisfeitos, enquanto que uma decomposição OU requer que ao menos um sub-nó seja satisfeito. Apenas um tipo de decomposição por nó é permitido. Os caminhos alternativos (decomposições OU) são avaliados através de objetivos qualitativos chama-

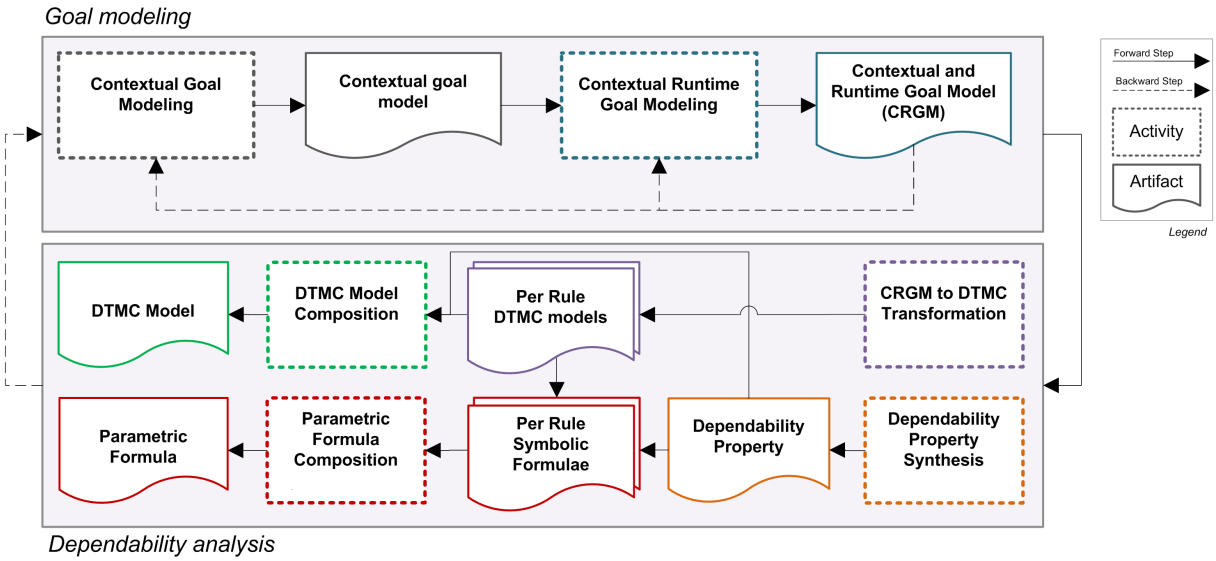

Figura 2.1: Processo de análise de dependabilidade orientada a objetivos [18]

dos de *soft-goals*. *Links* de contribuição identificam o impacto positivos ou negativo de alternativas em *soft-goals*.

A ativação de um objetivo, a decisão sobre as alternativas aplicáveis para alcançá-lo, assim como a qualidade de cada uma dessas alternativas, podem ser dependente de contexto. Modelos Orientados a Objetivos Contextual (CGM) [1] são propostos para capturar a inter-relação entre modelos orientados a objetivos e contexto, e suportam uma adaptação em tempo de execução onde atores mudam para uma estratégia alternativa que se encaixa dado um contexto, tanto em termos de aplicabilidade quanto de qualidade. A Figura 2.2 ilustra o modelo orientado a objetivo contextual.

2.1.2 Checagem Probabilística de Modelos (PMC)

Muitos sistemas são suscetíveis a vários fenômenos de natureza estocástica e ao não-determinismo em seus comportamentos [18]. Em contraste às técnicas de checagem de modelos em que a corretude absoluta de um sistema é verificada, a checagem probabilística de modelos é sobre computar as probabilidades com as quais as propriedades de interesse são verificadas no sistema. No GODA, tal probabilidade representa a estimativa de sucesso na execução de uma tarefa, ou seja, a confiabilidade da tarefa [19]. O PMC permite, também, que sejam feitas declarações quantitativas sobre o comportamento do sistema, expressas como probabilidades ou expectativas, além de declarações qualitativas feitas por checagens de modelo convencional [15].

A checagem probabilística de modelos permite uma verificação automatizada das propriedades comportamentais desejadas em um sistema inspecionando todos os estados do modelo do sistema [3]. PMC é adequado para modelagem e raciocínio sobre requisitos de dependabilidade. Dado um modelo de sistema e uma ou mais propriedades de dependabilidade, o PMC pode ser projetado para verificar a conformidade do modelo em cumprir tantos os requisitos funcionais quanto os requisitos não-funcionais do sistema.

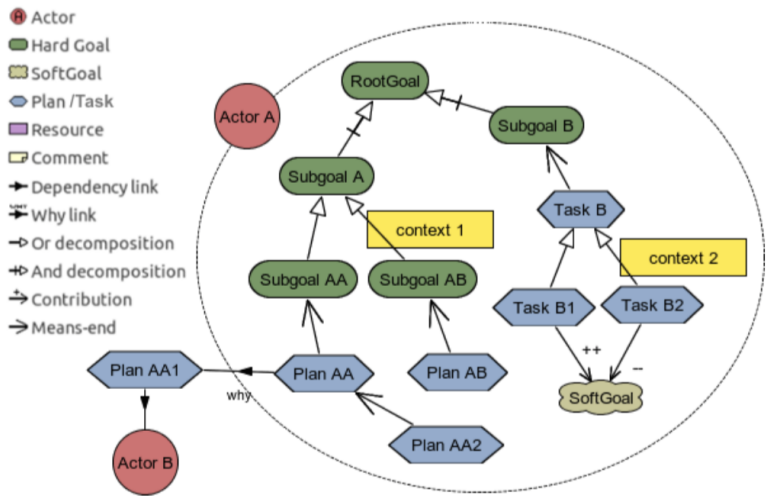

Figura 2.2: Representação genérica de um modelo orientado a objetivo contextual [19]

Entre os tipos mais populares de sistemas de transição empregados em PMC estão aqueles baseados em cadeias de Markov, por exemplo, o DTMC e o MDP (*Markov Decision Processes*) [15]. A técnica PMC permite a previsão da performance e da dependabilidade de sistemas baseado em eventos probabilísticos e comportamentais descritos nos modelos probabilísticos [18]. O PMC, como uma técnica de verificação de modelo, requer [18]:

- Uma descrição do sistema a ser analisado, normalmente dado em alguma linguagem de modelagem alto-nível, como DTMC;
- Uma especificação formal da propriedades quantitativas do sistema a serem analisadas, normalmente expressa em variantes de lógica temporal, como PCTL (*Probabilistic Computation Tree Logic*) [11].

A técnica PMC utilizada no GODA é suportada pelo verificador de modelo probabilístico PRISM [16], que suporta tanto cadeias de markov de tempo discreto e contínuo, quanto MDP. PRISM é adequado para diferentes tipos de avaliações de modelo dependendo do nível de abstração, do tipo de modelo probabilístico e das propriedades PCTL a serem analisadas [18].

O ambiente de análise oferecido pela ferramenta PRISM, porém, é limitada pela verificação de uma combinação única de variáveis inicializadas por vez [18]. Uma checagem de modelo paramétrica provê maior flexibilidade de análise, já que variáveis constantes no modelo podem ser substituídas por parâmetros [12]. Assim, considerando o PMC, a ferramenta PARAM amplia a linguagem PRISM com a palavra reservada *param*, utilizada para variáveis que descrevem probabilidades de transição de estado [18]. Então, dado um modelo probabilístico com variáveis *param* adicionais e uma propriedade PCTL, uma fórmula paramétrica correspondente é gerada.

2.1.3 Modelo Orientado a Objetivo Contextual e em Tempo de Execução (CRGM)

Como descrito na Sub-seção 2.1.1, o CGM é um modelo de requisito que captura a relação entre estratégias alternativas para alcançar metas e o espaço de mudanças de contexto para que o cumprimento de um objetivo dependente de contextos seja possível.

O RGM [7] incrementa objetivos e tarefas com especificação comportamental e faz com que seja possível verificar se tarefas e objetivos são compatíveis com suas especificações de classe, que é uma característica principal de dependabilidade.

Um CRGM refina o conceito de um CGM e RGM. Neste caso, objetivos são cumpridos com o sistema levando em consideração a especificação contextual do CGM [1], e a cooperação de objetivos e tarefas instanciados em tempo de execução, seguindo as regras de tempo de execução do RGM [7]. A Tabela 2.1 sintetiza uma descrição textual de cada regra RGM e seu significado correspondente em termos de qual comportamento ele especifica.

A definição matemática de um CRGM [19] é descrita a seguir.

Definição 1 *Um CRGM é uma tupla* $CR_M = (M, rt\_annot(), ctx\_annot(), ID())$ *onde:*

- M *é um modelo orientado a objetivos em tempo de projeto [7] definido como uma tupla* (N, R) *onde* N *é o conjunto de objetivos e tarefas no modelo, e* R *é o conjunto de relacionamentos entre os elementos de* N*.*
- $rt\_annot()$ *é uma função de anotação de tempo de execução que retorna a anotação de tempo de execução associada à um nó* $n \in N$*, onde* $rt\_annot(n)$ *pode ser uma única regra de tempo de execução ou uma composição de tais regras.*
- $ctx\_annot()$ *é uma função de anotação de contexto que retorna a fórmula de contexto associada à um nó* $n \in N$*.*
- $ID()$ *é uma função índice que mapeia todo* $n \in N$ *em um identificador* $ID(n) = prefix(n) + counter(n)$*, onde* $prefix(n)$ *retorna um* G *ou um* T *para objetivos e tarefas, respectivamente. A função* $counter(n)$ *retorna:*
  - *um inteiro positivo incrementado único para identificar unicamente cada objetivo. Ex.:* G_1, G_2, G_3*;*
  - *um inteiro incrementado progressivamente para tarefas means-end. Ex.:* T_1, T_2, T_3*;*
  - *o mesmo inteiro da tarefa means-end correspondente junto com um número decimal que corresponde ao nível de profundidade da tarefa. Ex.:* $T_{1.1}, T_{1.2}, T_{1.11}, T_{1.12}$*.*

A Figura 2.3 ilustra um CRGM com os objetivos e tarefas IDs definidos pelas funções $ID()$, $rt\_annot()$ e $ctx\_annot()$. O objetivo principal $G0$ é a realização de uma execução sequencial dos sub-objetivos $G1$ e $G2$, enquanto que o sub-objetivo $G1$ é a realização de uma execução alternativa do sub-objetivo $G3$ ou do sub-objetivo $G4$. Tarefas-folhas $T3$ e $T4$ realizam a execução dos sub-objetivos $G3$ e $G4$, respectivamente. É possível observar que a realização do sub-objetivo $G3$ requer a execução de no máximo três instâncias da tarefa $T3$ (denotado por $[T3+3]$), no caso de a alternativa selecionada ser a $G3$. Por outro lado, o sub-objetivo $G2$ é cumprido com a execução da tarefa $T2$, que é uma execução condicional da tarefa $T2.1$, ou da tarefa $T2.2$, caso $T2.1$ renha sido contida da execução.

Tabela 2.1: Regras RGM, onde E1, E1 e E2 representam objetivos ou tarefas em um modelo orientado a objetivos (estendido de Dalpiaz et al. [7]).

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.

Finalmente, o contexto $C2$ é uma fórmula predicado especificada como uma conjuntura dos fatos contextuais $fact1$ e $fact2$. Como resultado, o sub-objetivo $G2$ só poderá ser executado caso $C2$ seja verdadeiro.

2.1.4 Arquitetura de Implementação do GODA

O *framework* GODA automatiza a geração de um modelo DTMC, a partir de um CRGM, em linguagens de modelagem PRISM e PARAM. Os ambientes de modelagem e transformação estenderam uma ferramenta *open source* existente que dá suporte à metodologia de desenvolvimento TROPOS chamada TAOM4E [20]. TAOM4E fornece um ambiente gráfico para modelagem de objetivos baseado no *Eclipse Modelling Framework* (EMC) e *Graphical Editing Framework* (GEF). Além disso, a ferramenta também suporta um modelo dirigido a agente de geração de código. O gerador de código de CRGM

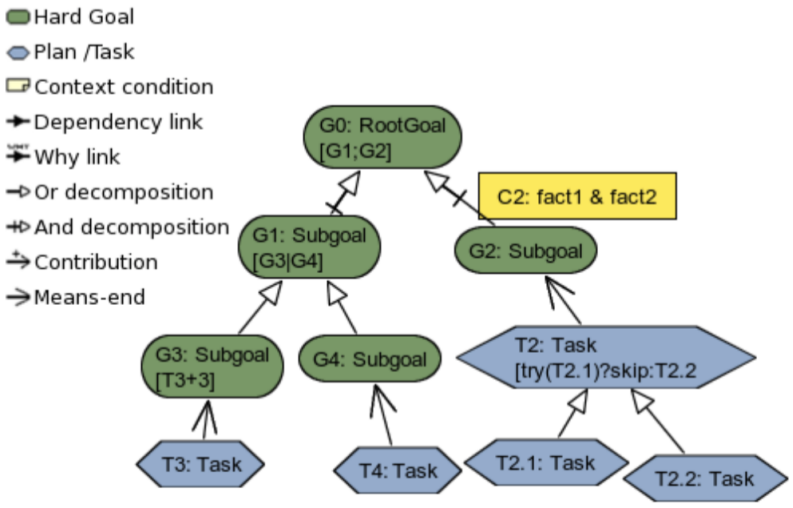

Figura 2.3: Exemplo de um CRGM genérico com objetivos e tarefas definidos pela função $ID()$ e também a $rt\_annot()$ e $ctx\_annot()$ para diferentes objetivos e tarefas[19]

em DTMC também foi implementado como um *plugin* do Eclipse em linguagem JAVA e integrado ao ambiente de modelagem de modelos fornecido pela ferramenta TAOM4E [1].

O propósito da geração automática de código CRGM em DTMC é otimizar o passo de verificação formal abstraindo a modelagem probabilística *know-how* da análise, e reduzir a sobrecarga e os erros causados pela geração manual do modelo de verificação. Isso deve aumentar a viabilidade de adotar o GODA pois deixa analistas focados com a modelagem e analisa o sistema em diferentes contextos de operação. A integração da geração CRGM em DTMC ao ambiente TAOM4E forma o ambiente de modelagem e análise para o GODA. A Figura 2.4 apresenta uma visão de alto nível da arquitetura implementada neste *framework*. Como entrada, o GODA recebe um arquivo CRGM no formato TAOM4E e como saída, ele gera os modelos de verificação PRISM e PARAM do CRGM baseados nas propriedades de acessibilidade dos objetivos.

2.2 Geração Automática de Código DTMC

Uma etapa importante da análise de dependabilidade realizada pelo GODA é a geração automática de um modelo DTMC em linguagens PRISM e PARAM a partir de um CRGM. O propósito dessa automação é otimizar o passo de verificação formal, abstraindo a modelagem probabilística *know-how* dos analistas e reduzindo o *overhead* e erros causados pela geração manual do modelo de verificação [18]. Este capítulo apresentará o processo de geração de código CRGM em DTMC. Com o entendimento desta geração, a atividade de teste é melhor realizada.

[1] *Framework* GODA pode ser baixado no repositório do *GitHub* em https://github.com/lesunb/CRGMToPRISM

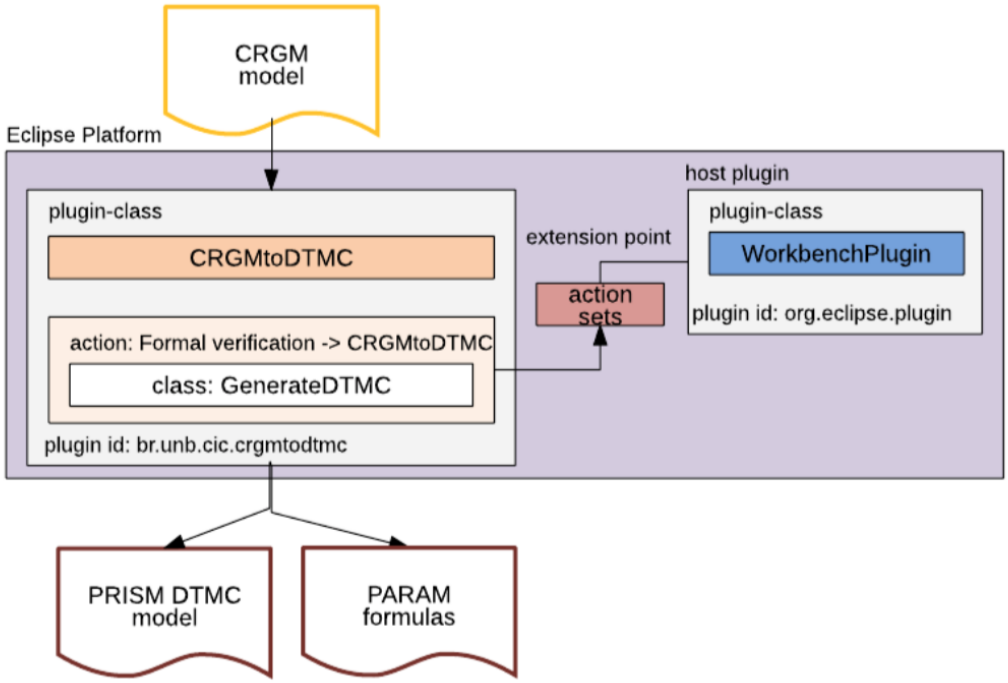

Figura 2.4: Visão de alto nível da arquitetura de implementação do GODA [18]

Segundo [18], o processo de geração de código CRGM em DTMC foi dividido em duas fases:

- Fase de análise: o arquivo de entrada do TROPOS, contendo um modelo orientado a objetivo com um ator do sistema, é analisado utilizando um algoritmo de profundidade. Objetivos e tarefas são mantidos em memória com metadados relevantes em um container de objetos. As anotações de comportamento em elementos não-folhas e os efeitos de contexto em qualquer objetivo ou tarefa também são analisados. Em uma decomposição da árvore, tanto as regras comportamentais quanto os efeitos de contexto devem alcançar as tarefas-folhas no final dos ramos relacionados.
- Fase de escrita: o container de objetos com todas as informação necessárias para a geração de um modelo DTMC são cruzadas com a do container do objetivo raiz. Módulos de tarefas-folhas do DTMC são criados e concatenados à uma variável global. Para cada objetivo G_i no modelo, é concatenada ao modelo uma fórmula de sucesso declarada como G_1. Depois de processar todos os efeitos de contexto, apenas uma declaração das variáveis de contexto analisadas é anexada ao modelo. Finalmente, o arquivo de saída com o modelo DTMC é criado em uma pasta definida na sessão de preferências do ambiente de modelagem. O nome do arquivo contém o nome do ator do modelo.

A Figura 2.5 apresenta a arquitetura de implementação de gerador CRGM em DTMC. As classes na Figura 2.5 interagem da seguinte maneira [18]:

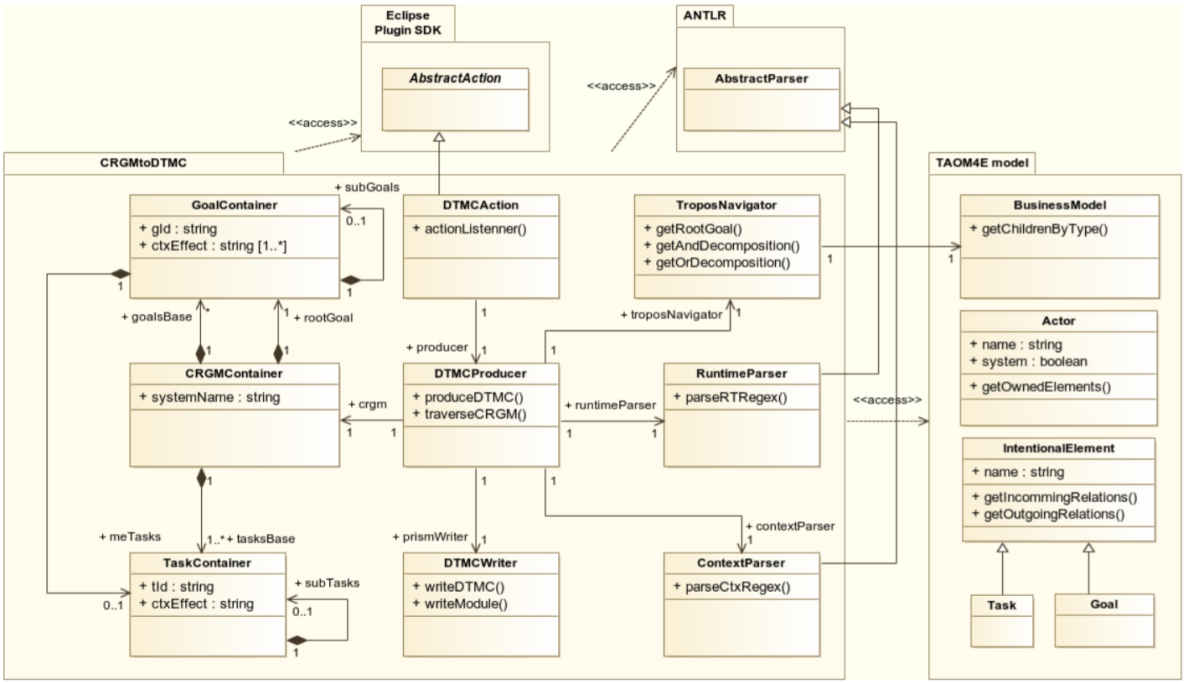

Figura 2.5: Arquitetura de implementação do gerador CRGM em DTMC [18]

1. O usuário clica em um item específico do menu da ferramenta de modelagem orientada a objetivos TAOM4E chamado *Generate PRISM model* no menu *GODA*.
2. O *singleton DTMCAction* recebe o contexto de ambiente do modelo orientado a objetivos TROPOS e cria uma nova *thread* para o *DTMCProducer*, passando o arquivo de entrada TROPOS do contexto de modelagem.
3. O *DTMCProducer* instancia o *TroposNavigator* com o arquivo de entrada do TROPOS.
4. O *TroposNavigator* acessa o modelo de negócios do modelo TAOM4E e recupera os atores no arquivo de entrada.
5. O ator do sistema é identificado e seu objetivo principal (ou objetivo raiz) é extraído.
6. O *DTMCProducer* começa o algoritmo de profundidade pelo objetivo raiz chamando o método *addGoal()*.
7. Cada objetivo ou tarefa não-folha tem seus elementos filhos extraídos e salvos como containers na instância da classe *CRGMDefinition* antes de uma chamada recursiva a *addGoal()*/*addTask()*:
   - Anotações comportamentais são analisadas pelo *RTParser* e os atributos correspondentes são setados nos containers dos elementos filhos.
   - Todos os efeitos de contexto em objetivos e tarefas são analisados pelo *CtxParser* e os atributos correspondentes são setados nos containers dos elementos filhos.

- É realizada uma chamada recursiva a *addGoal()/addTask()*.

8. O retorno da chamada recursiva de um dado elemento é adicionado aos atributos do elemento pai para propagar incrementos de tempo aninhados para objetivos/tarefas subsequentes.
9. O arquivo *DTMCDefinition* com todos os objetivos e tarefas e uma referência para o objetivo raiz é passado para o método *writeModel()* na classe *DTMCWriter*.
10. Padrões da linguagem PRISM para a criação do modelo DTMC são carregados a partir de arquivos no pacote de *plugins* do Eclipse.
11. 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 comportamentais nos padrões correspondentes da linguagem PRISM.
    - Variáveis para comportamentos opcionais e alternativos são anexadas ao modelo antes dos módulos de tarefas-folhas correspondentes.
    - Cada chamada de criação de módulo de tarefa-folha retorna uma fórmula booleana para seu próprio sucesso.
    - Fórmulas de sucesso de objetivos e tarefas não-folhas são criadas concatenando as 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 próprios pares de tipos/valores de variáveis armazenados em um conjunto de coleções sem duplicações.
12. O arquivo de saída DTMC é aberto.
13. O cabeçalho do modelo DTMC é escrito.
14. Pares de variáveis de contexto são escritos como declarações de variáveis *unsigned*.
15. Os módulos de tarefas-folhas, fórmulas de sucesso de objetivos e variáveis *unsigned* alternativas/opcionais são escritos.
16. O arquivo de saída DTMC é fechado.

A geração automática de modelo DTMC em linguagens PRISM e PARAM a partir de um CRGM foi implementada utilizando a linguagem de programação Java. Segundo [13] e [4], o *software* orientado a objetos é composto de classes as quais encapsulam uma série de métodos, em geral de baixa complexidade, que cooperam entre si na implementação de dada funcionalidade.

O paradigma de programação orientada a objetos possui um conjunto de construções poderosas, que apresentam riscos de erros (*fault hazard*) e problemas de teste [9]. Isso se deve à inevitável necessidade de encapsulamento de métodos e atributos dentro de uma classe, da variedade de modos como um subsistema pode ser composto e da possibilidade

de, em poucas linhas de código, dar um comportamento ao sistema que só será definido em tempo de execução [9]. Além disso, cada nível em uma hierarquia de heranças dá um novo contexto para a características herdadas; o comportamento correto nos níveis mais elevados não é garantido nos níveis mais baixos [9].

Garantir a correta transforação de modelos no GODA torna-se, então, elementar para aumentar a confiabilidade deste *framework*. No próximo capítulo será apresentada a metodologia utilizada para o desenvolvimento de testes que abrangem deficiências do GODA sob a perspectiva de corretude da modelagem.

2.3 Testes Funcionais

Técnicas e critérios de teste fornecem ao projetista de *software* uma abordagem sistemática e teoricamente fundamentada para a condução da atividade de teste [9]. Além disso, constituem um mecanismo que pode auxiliar na garantia da qualidade dos testes e na maior probabilidade em revelar defeitos no *software*. Várias técnicas podem ser adotadas para se conduzir e avaliar a qualidade da atividade de teste.

O teste funcional é uma técnica utilizada para projetar casos de teste na qual o programa ou sistema é considerado uma caixa preta. Para testá-lo, são fornecidas entradas e então avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados [9]. Assim, o código fonte do programa não é necessário, sendo obrigatório somente a especificação do produto.

Em princípio, o teste funcional pode ser denominado como teste exaustivo [21], submetendo o programa ou sistema a todas as possíveis entradas. No entanto, o domínio de entrada pode ser infinito ou muito grande, tornando o tempo da atividade de teste inviável, e fazendo com que essa alternativa se torne impraticável [9]. Com essa limitação da atividade de teste, não é possível afirmar que o programa esteja correto. Logo, foram definidas técnicas de teste e diversos critérios pertencentes a cada uma.

Segundo [9], os critérios mais conhecidos da técnica de teste funcional são Particionamento de Equivalência, Análise do Valor Limite, Grafo Causa-Efeito e Error-Guessing. Além desses, existem outros como Teste Funcional Sistemático [17], *Syntax Testing*, *State Transition Testing* e *Graph Matrix* [14]. Todos os critérios da técnica funcional se baseiam apenas na especificação do sistema ou programa testado. Eles podem ser aplicados em toda as fases de teste e em produtos desenvolvidos com qualquer paradigma de programação [9], pois não levam em consideração os detalhes de implementação [6]. Os critérios utilizados neste trabalho são definidos a seguir.

2.3.1 Particionamento de Equivalência

O critério de Particionamento de Equivalência foi criado como uma solução para o teste exaustivo. Este último acontece quando o teste funcional é utilizado submetendo o programa ou sistema à todas as entradas possíveis. O domínio de todas as possíveis entradas de um programa pode ser infinito ou muito grande, o que inviabiliza a atividade de teste. O objetivo do teste de partição, então, é dividir o domínimo de entrada do programa tal que, quando o testador selecionar casos de teste dos subdomínios (subconjuntos ou partições), o conjunto de testes resultante será uma boa representação do domínio in-

teiro. Assim, a quantidade de dados de entrada se torna finita e passível de ser tratado durante a atividade de teste.

O Particionamento de Equivalência divide o domíno de entrada em classes de equivalência. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para as condições de entrada. Assim, uma vez definidas as classes de equivalência, pode-se assumir, com alguma segurança, que qualquer elemento da classe pode ser considerado um representante desta, pois todos eles devem se comportar de forma similar [9].

Assim que as classes de equivalência são identificadas, pode-se determinar os casos de teste. Um elemento de cada classe deve ser escolhido, de forma que cada novo caso de teste cubra o maior número possível de classes válidas. Para as classes inválidas, é necessário gerar casos de teste exclusivos, já que um elemento de uma classe inválida pode mascarar a validação do elemento de outra classe inválida [6].

A vantagem deste critério é a possibilidade de reduzir o domínio de entrada e de criar casos de teste baseados apenas na especificação.

2.3.2 Análise do Valor Limite

Este critério é usado em conjunto com o Particionamento de Equivalência, mas no lugar de escolher dados de teste aleatoriamente, eles são escolhido de forma que o limitante de cada classe de equivalência seja explorado. Segundo Myers [21], casos de teste que exploram condições limites têm maior probabilidade de encontrar defeitos. Essas condições correspondem a valores exatamente sobre ou imediatamente acima ou abaixo das classes de equivalência [9].

As vantagens do critério Análise do Valor Limite são similares às do critério Particionamento de Equivalência, sendo que existem diretrizes para que os dados de teste sejam estabelecidos, uma vez que estes devem explorar especificamente os limites das classes identificadas [9].

2.3.3 Teste Funcional Sistemático

O critério de Teste Funcional Sistemático combina os critérios Particionamento de Equivalência e Análise do Valor Limite. Particionados os domínios de entrada e de saída, este critério requer ao menos dois casos de teste de partição para cada classe de equivalência, minimizando, assim, o problema de que defeitos coincidentes mascarem falhas (assim como ocorre no Particionamento de Equivalência). Da mesma maneira que é feito na Análise do Valor Limite, o critério Teste Funcional Sistemático requer avaliação no limite de cada subdomínio e subsequente a ele [17].

As vantagens do Teste Funcional Sistemático são as mesmas dos dois critérios citados anteriormente. Além disso, este critério enfatiza mais fortemente a seleção de mais de um caso de teste por partição e/ou limite, aumentando assim a chance de revelar os defeitos sensíveis a dados e também a probabilidade de obter uma cobertura de código do produto que está sendo testado [9].

Capítulo 3

Metodologia Proposta

A transformação de modelos no GODA pode apresentar deficiências tanto funcionais quanto estruturais. É importante, então, testar os elementos do modelo CRGM e, assim, verificar que a geração do modelo DTMC esteja ocorrendo corretamente.

Os elementos do CRGM são:

- Construções básicas;
- Anotações já usadas e novas;
- Transformações nas linguagens PRISM e PARAM.

As construções básicas representam a estrutura interna do programa. É necessário o seu conhecimento, sendo os aspectos de implementação fundamentais para a geração dos casos de testes associados.

As anotações (novas e antigas) utilizadas no modelo são obtidas a partir da especificação do GODA. Anotações antigas se resumem em *Goals* (objetivos), *Tasks* (tarefas) e relacionamentos já utilizadas pelo TROPOS. Anotações novas são as *Runtime Annotation* e *Context Annotation*, que são especificações comportamentais e contextuais do modelo, respectivamente.

As transformações nas linguagens PRISM e PARAM representam o arquivo de saída criado pelo GODA no final do processo de geração de código contendo o modelo DTMC nas linguagens citadas.

Este capítulo apresentará o planejamento, execução e documentação de testes com foco nos problemas de anotações que podem ocorrer no modelo CRGM. A princípio, serão apresentadas as características de nomenclatura das anotações utilizadas na modelagem. Em seguida, será feito um planejamento da atividade de teste com base na técnica funcional. Logo após, serão definidos os casos de testes específicos. Por fim, será apresentada a implementação dos testes com entradas pré-definidas.

3.1 Nomenclatura das anotações

Com base na especificação do GODA, as anotações utilizadas na transformação de modelos CRGM em DTMC são de objetivos, tarefas, anotação de tempo de execução

(*runtime annotation*) e anotação de contexto (*context annotation*). A seguir são apresentadas as características de nomenclatura que definem os estados válidos e inválidos de cada uma destas anotações.

1. Objetivo válido
   (a) Objetivos válidos devem ser escritos da forma **G#: *description* [*runtime annotation*]**, sendo G# o identificador (o prefixo G seguida por um número inteiro único #), *description* uma descrição textual do objetivo, e *runtime annotation* a anotação de tempo de execução (opcional em alguns casos).
   (b) Objetivo com nenhum nó filho não deve conter anotação de tempo de execução em sua escrita.
   (c) Objetivo com 1 (um) único nó filho não precisa conter anotação de tempo de execução. Caso contenha, as únicas anotações possíveis são **opt(E)**, **E+n**, **E#n** e **E@n**, em que E representa o identificador do nó filho e n é um número inteiro $n > 0$.
   (d) Objetivo com 2 (dois) ou mais nós filhos deve conter anotação de tempo de execução em sua escrita.
2. Objetivo inválido
   (a) Objetivos que não possuem nós filhos e que não são escritos da forma **G#: *description*** são considerados inválidos.
   (b) Objetivos com apenas um nó fillho que não são escritos da forma **G#: *description*** ou **G#: *description* [*runtime annotation*]** são considerados inválidos.
   (c) Objetivos com mais de um nó filho que não são escritos da forma **G#: *description* [*runtime annotation*]** são considerados inválidos.
   (d) Objetivo com nenhum nó filho que contenha anotação de tempo de execução é considerado inválido.
   (e) Objetivo com 1 (um) único nó filho que contenha anotação de tempo de execução é considerado inválido (com exceção das anotações **opt(E)**, **E+n**, **E#n** e **E@n**).
   (f) Objetivo com 2 (dois) ou mais nós filhos que não possua anotação de tempo de execução é considerado inválido.
   (g) Objetivos (2 ou mais) que possuam o mesmo identificador G# são considerados inválidos.
3. Tarefa válida
   (a) Tarefas válidas devem ser escritas da forma **T#: *description* [*runtime annotation*]**, sendo T# o identificador da tarefa, *description* uma descrição textual da tarefa, e *runtime annotation* a anotação de tempo de execução (opcional em alguns casos).
   (b) Nas tarefas de nível de profundidade 1 (um), o identificador é formado pelo prefixo T seguido por um número inteiro único #.

(c) Nas tarefas de nível de profundidade 2 (dois), o identificador é formado pelo identificador da tarefa pai T# seguida por um ponto "."e por um número inteiro único #. Por exemplo, as tarefas T1.1 e T1.2 são filhas da tarefa T1.

(d) Nas tarefas de nível de profundidade 3 (três) ou mais, o identificador é formado pelo identificador da tarefa pai T#.# seguida por um número inteiro único #. Por exemplo, as tarefas T2.11, T2.12 e T2.13 são filhas da tarefa T2.1.

(e) Tarefa com nenhum nó filho não deve conter anotação de tempo de execução em sua escrita.

(f) Tarefa com 1 (um) único nó filho não precisa conter anotação de tempo de execução. Caso contenha, as únicas anotações possíveis são **opt(E)**, **E+n**, **E#n** e **E@n**, em que E representa o identificador do filho e n é um número inteiro $n > 0$.

(g) Tarefa com 2 (dois) ou mais nós filhos deve conter anotação de tempo de execução em sua escrita.

(h) Dentro de um mesmo nível de profundidade, tarefas irmãs possuem identificadores únicos; tarefas que não são irmãs, no entanto, podem possuir o mesmo identificador. Por exemplo, G1 possui as tarefas T1 e T2, enquanto que G2 possui diferentes tarefas T1, T2 e T3.

4. Tarefa inválida

(a) Tarefas que não são escritas da forma **T#: *description* [*runtime annotation*]** são consideradas inválidas.

(b) Tarefa de nível de profundidade 2 (dois) cujo identificador não é formado pelo identificador da tarefa pai seguido de um ponto "."e um número inteiro único é considerada inválida. Por exemplo, a tarefa T11 ser filha da tarefa T1.

(c) Tarefa de nível de profundidade 3 (três) cujo identificador não é formado pelo identificador da tarefa pai seguido por um número inteiro único, é considerada inválida. Por exemplo, as tarefas T1.21 e T1.31 serem filhas da tarefa T1.1.

(d) Tarefa com nenhum nó filho que contenha anotação de tempo de execução é considerada inválida.

(e) Tarefa com 1 (um) único nó filho que contenha anotação de tempo de execução é considerada inválida (com exceção das anotações **opt(E)**, **E+n**, **E#n** e **E@n**).

(f) Tarefa com 2 (dois) ou mais nós filhos que não possua anotação de tempo de execução é considerada inválida.

(g) Tarefas irmãs (2 ou mais) que tenham o mesmo identificador são consideradas inválidas. Por exemplo, T1 possui as diferentes tarefas filhas T1.1 e T1.1.

5. Anotação de tempo de execução válida

(a) Toda anotação de tempo de execução válida deve ser escrita dentro de colchetes.

(b) Anotação de tempo de execução só é considerada válida se estiver após a descrição do objetivo ou da tarefa em questão.

(c) Anotação de tempo de execução válida deve obedecer à regras específicas, descritas na Tabela 2.1.

(d) Toda anotação de tempo de execução deve referenciar os filhos diretos da tarefa ou do objetivo em questão.

6. Anotação de tempo de execução inválida

   (a) Anotação de tempo de execução escrita fora de colchetes é considerada inválida.

   (b) Anotação de tempo de execução escrita antes da descrição do objetivo/tarefa é considerada inválida.

   (c) Anotação de tempo de execução que não obedece as regras descritas na Tabela 2.1 é considerada inválida.

   (d) Anotação de tempo de execução que referencia filhos não existentes do objetivo ou da tarefa em questão é considerada inválida.

7. Anotação de contexto válida

   (a) Anotação de contexto válida tem a forma ***operand operator value***. *Operands* são fatos de contexto e devem iniciar com caractere normal, não-dígito, e podem terminar com dígitos. *Operator* são operadores PRISM. *Value* é o valor a ser comparado, podendo ser booleano, inteiro ou *double*.

   (b) Anotação de contexto válida deve usar um operador PRISM válido para definir seus fatos de contexto. Operadores válidos são $<$, $\leq$, $>$, $\geq$, $=$, $!=$, &, |.

8. Anotação de contexto inválida

   (a) Anotações de contexto escritas fora da forma ***operand operator value***.

   (b) Anotação de contexto com operando escrito fora do formato pré-estabelecido, iniciando com dígitos.

   (c) Anotação de contexto com operador inválido.

   (d) Anotação de contexto cujo valor a ser comparado não é booleano, inteiro ou *double*.

3.2 Planejamento dos Testes

A abordagem escolhida para testar os problemas de anotações do modelo CRGM foi a de teste funcional, utilizando o critério Teste Funcional Sistemático. Como os detalhes de implementação não são considerados, o enfoque será na funcionalidade do GODA em transformar modelos CRGM em DTMC a partir de anotações utilizadas no ambiente de modelagem TAOM4E.

Os testes serão desenvolvidos utilizando a linguagem de programação Java. Para automatizar o conjunto de testes será utilizada a ferramenta JUnit, que viabiliza a documentação e a execução automática de casos de teste [9] na linguagem Java. Ele fornece relatórios sobre o comportamento dos casos de teste em relação ao que foi especificado. A

ideia deste *framework* é implementar algumas classes específicas que armazenem informações sobre os dados de entrada e a respectiva saída esperada para cada caso de teste [9]. Após a execução dos testes, a saída obtida é comparada com a saída esperada, e então é fornecido um relatório sobre quaisquer discrepâncias encontradas.

3.3 Testes Abstratos

Nesta seção serão apresentados os casos de teste funcionais utilizando a abordagem de Teste Funcional Sistemático. Também serão definidas as entradas utilizadas e saídas esperadas ao executar os testes concretos. Como mencionado anteriormente, o teste funcional estará focado nas anotações do modelo CRGM utilizadas no GODA.

Casos de teste funcional são baseados na especificação do sistema. No caso do GODA, a documentação de especificação se encontra no repositório do GitHub[1] e na dissertação de mestrado [18]. Para a criação do teste funcional abstrato, serão utilizadas as características que validam ou invalidam as anotações utilizadas pelo GODA na transformação de modelos. Primeiramente, serão definidas as classes de equivalência com base nessas características. Por fim, serão apresentados os casos de teste para cada classe de equivalência, seguindo o critério Teste Funcional Sistemático.

3.3.1 Classes de Equivalência

As classes de equivalência para a atividade de teste serão definidas a partir das características das anotações apresentadas no Seção 3.1. A metodologia utilizada para o particionamento foi, num primeiro instante, dividir classes válidas e classes inválidas. As classes válidas são aquelas cujas entradas são valores esperados pelo programa. As classes inválidas são aquelas cujas entradas também são inválidas, ou seja, inesperadas pelo programa. Então, houve um refinamento das classes de entrada. Isso significa que, os elementos de uma mesma classe tratados de forma diferente pelo programa foram subdivididos em classes menores.

As classes de equivalência definidas foram:

1. Classe válida

   Classe de entrada válida que verifica o comportamento do GODA tendo como entrada um modelo cujas anotações estejam totalmente de acordo com a especificação do *framework*.

2. Objetivos e/ou tarefas e/ou anotações de contexto escritos fora do padrão

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas e/ou anotações de contexto escritos de forma incorreta no ambiente de modelagem, segundo a especificação do *framework*.

3. Objetivos e/ou tarefas que não contenham nós filhos

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas que não possuam nós filhos mas, ainda assim, contenham anotação de tempo de execução em sua escrita.

[1] O repositório do GODA pode ser acessado em https://github.com/lesunb/CRGMToPRISM

4. Objetivos e/ou tarefas que contenham apenas um nó filho

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas que possuam apenas um nó filho, e que contenham anotação de tempo de execução diferente das permitidas na sua escrita. Neste caso as anotações permitidas são opt(E), E+n, E#n, E@n.

5. Objetivos e/ou tarefas que contenham dois ou mais nós filhos

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas que contenham dois ou mais nós filhos mas não possuam anotação de tempo de execução em sua escrita.

6. Objetivos (dois ou mais) que contenham o mesmo identificador

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada dois ou mais objetivos que contenham o mesmo identificador G# em sua escrita.

7. Tarefas irmãs (duas ou mais) que contenham o mesmo identificador

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada duas ou mais tarefas irmãs que contenham o mesmo identificador T# em sua escrita.

8. Tarefas de nível de profundidade dois ou mais com identificadores fora do padrão

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada tarefas de nível de profundidade dois (ou mais) em que a notação dos identificadores não obedecem às regras da especificação.

9. Anotação de tempo de execução não identificável

   Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas cujas anotações de tempo de execução não sejam identificadas durante o processo de análise e identificação dos *tokens* do ambiente de modelagem. Isto é, anotação escrita fora de colchetes ou anotação escrita antes da descrição do objetivo/tarefa.

10. Anotação de tempo de execução referenciando nós não filhos

    Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas cujas anotações de tempo de execução estejam referenciando nós que não são filhos do objetivo/tarefa em questão.

11. Anotação de tempo de execução escrita fora do padrão

    Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada objetivos e/ou tarefas cujas anotações de tempo de execução estejam escritas fora do padrão detalhado na Tabela 2.1.

12. Anotação de contexto com operando e/ou operador PRISM inválidos

    Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada anotações de contexto contendo operandos e/ou operadores PRISM inválidos, segundo a especificação do *framework*.

13. Anotação de contexto com valores inválidos

    Classe de entrada inválida que verifica o comportamento do GODA tendo como entrada anotações de contexto contendo valores que não sejam inteiro, booleano ou *double*.

A Tabela 3.1 relaciona as classes de equivalência definidas com as regras de nomenclatura descritas na Seção 3.1.

Tabela 3.1: Classes de equivalência e regras de nomenclatura das anotações do modelo CRGM

Classe de Equivalência	**Regra de Nomenclatura**
Classe de Equivalência 1	Regras 1a, 1b, 1c, 1d, 3a, 3b, 3c, 3d, 3e, 3f, 3g, 3h, 5a, 5b, 5c, 5d, 7a e 7b
Classe de Equivalência 2	Regras 2a, 2b, 2c, 4a e 8a
Classe de Equivalência 3	Regras 2d e 4d
Classe de Equivalência 4	Regras 2e e 4e
Classe de Equivalência 5	Regras 2f e 4f
Classe de Equivalência 6	Regra 2g
Classe de Equivalência 7	Regra 4g
Classe de Equivalência 8	Regras 4b e 4c
Classe de Equivalência 9	Regras 6a e 6b
Classe de Equivalência 10	Regra 6d
Classe de Equivalência 11	Regra 6c
Classe de Equivalência 12	Regras 8a e 8b
Classe de Equivalência 13	Regra 8d

3.3.2 Casos de Teste

Uma vez identificadas as classes de equivalência, é necessário determinar os casos de teste. O critério Teste Funcional Sistemático requer *ao menos dois casos de teste de cada*

partição. Assim, problemas de que defeitos coincidentes mascarem falhas são minimizados. Além disso, este critério também requer a avaliação de valores limites e subsequentes de cada partição [17]. No caso do GODA, os dados de entrada são strings, e, como os comprimentos destas strings não influenciam no comportamento do programa, será explorada a validação dos caracteres que a compõem. A string nula não será considerada para os casos de teste porque o ambiente de modelagem TAOM4E não permite tal caso (é utilizado um valor *default* caso o usuário force a inserção da string nula). Desta maneira, os casos de teste definidos com base nas classes de equivalência descritas acima foram:

1. Classe válida

   (a) Caso de teste 1: modelo correto

   - Entrada (Figura 3.1): Utilização de vários objetivos, tarefas e anotações de tempo de execução diferentes. Todos obedecendo as regras de anotação descritas na especificação.

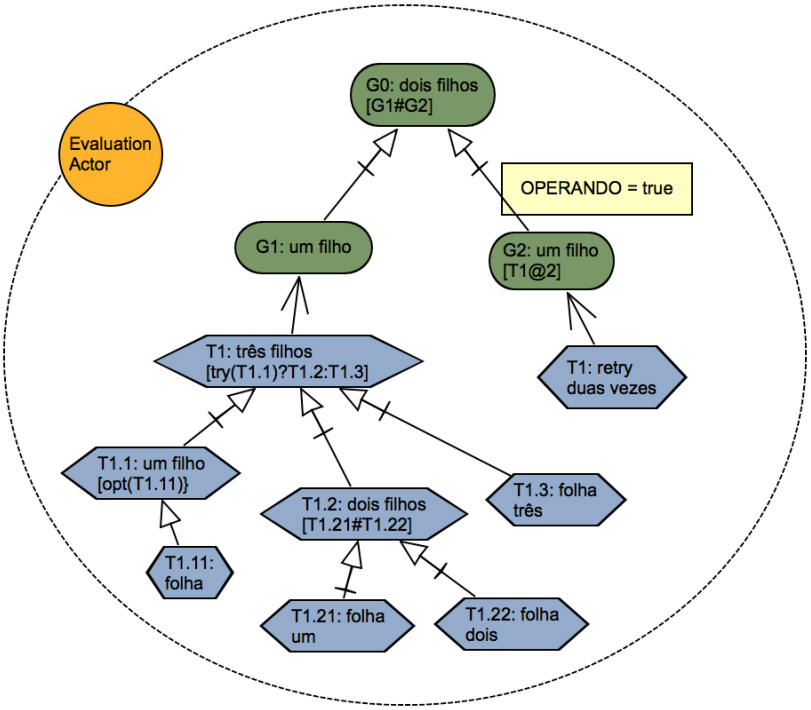

Figura 3.1: Ilustração de modelo para o caso de teste 1

   - Saída esperada: Execução finalizada com nenhuma exceção Java lançada pelo GODA.

   (b) Caso de teste 2: modelo correto

   - Entrada (Figura 3.2): Utilização de vários objetivos, tarefas, anotações de tempo de execução e anotações de contexto diferentes. Todos obedecendo as regras de anotação descritas na especificação.
   - Saída esperada: Execução finalizada com nenhuma exceção Java lançada pelo GODA.

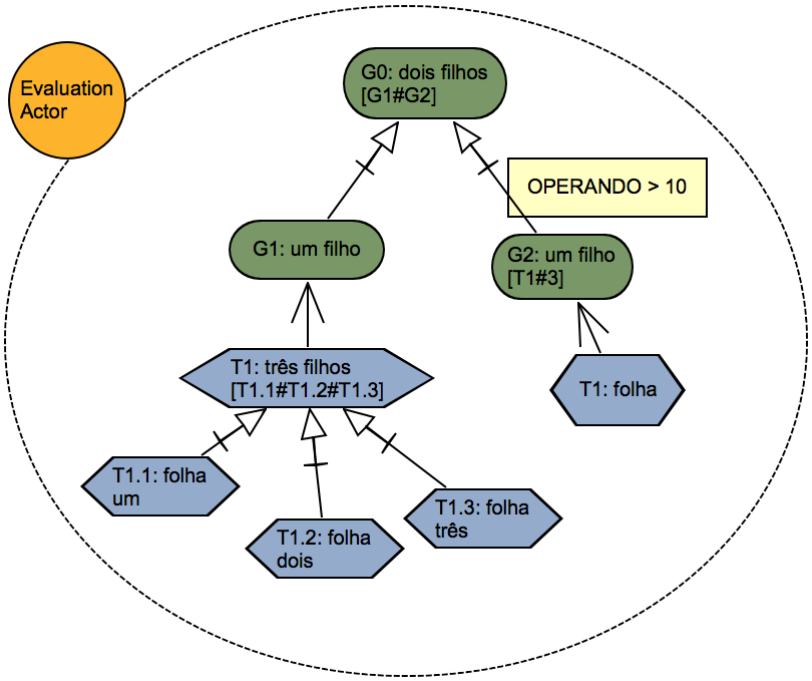

Figura 3.2: Ilustração de modelo para o caso de teste 2

2. Objetivos e/ou tarefas e/ou anotações de contexto escritos fora do padrão

   (a) Caso de teste 3: Objetivo escrito fora da *label* padrão

   - Entrada (Figura 3.3): Objetivo contendo dois nós filhos (G1 e G2 nós-folha), com a seguinte *label*: "Objetivo principal: testar erro de padrão [G1;G2]".
   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

   (b) Caso de teste 4: Tarefa escrita fora da *label* padrão

   - Entrada (Figura 3.4): Tarefa-folha (filha única do objetivo de *label* "G1: erro na tarefa"), com a seguinte *label*: "Tarefa única: folha".
   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

   (c) Caso de teste 5: Tarefa escrita fora da *label* padrão

   - Entrada (Figura 3.5): Tarefa-folha (filha única do objetivo de *label* "G1: erro na tarefa"), com a seguinte *label*: "T1.1: folha".
   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

   (d) Caso de teste 6: Anotação de contexto escrita fora da *label* padrão

   - Entrada (Figura 3.6): Anotação de contexto de *label*: "> Operand 3".
   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

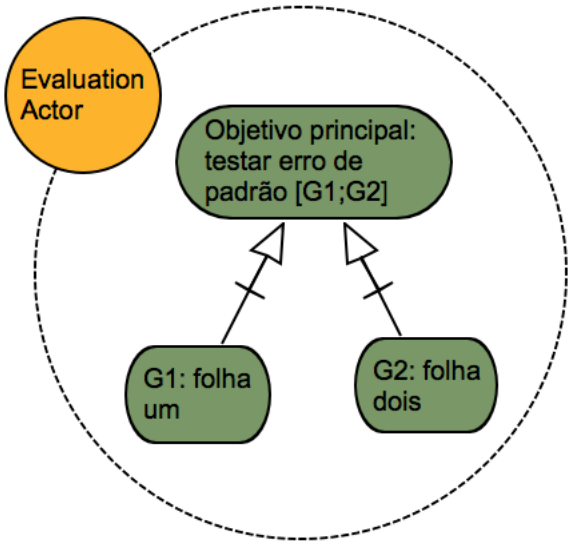

Figura 3.3: Ilustração de modelo para o caso de teste 3

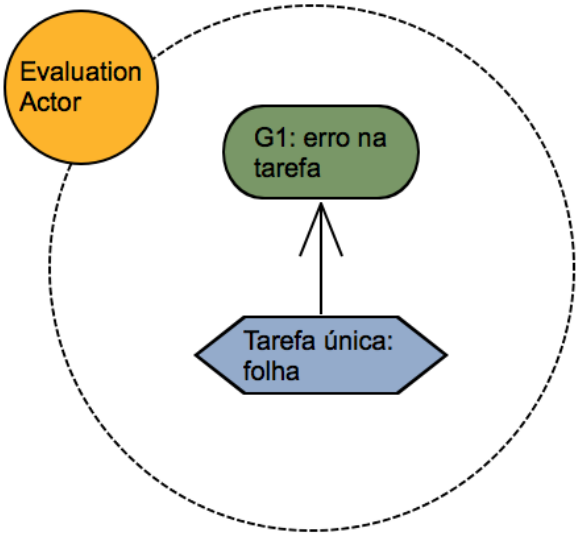

Figura 3.4: Ilustração de modelo para o caso de teste 4

3. Objetivos e/ou tarefas que não contenham nós filhos

   (a) Caso de teste 7: Objetivo que não possui nó filho mas contém anotação de tempo de execução

   - Entrada (Figura 3.7): Objetivo-folha com a seguinte *label*: "G1: nenhum filho [T1@2]".
   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

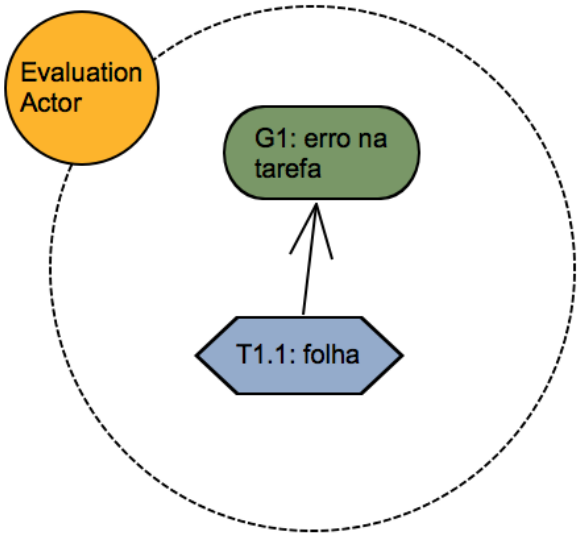

Figura 3.5: Ilustração de modelo para o caso de teste 5

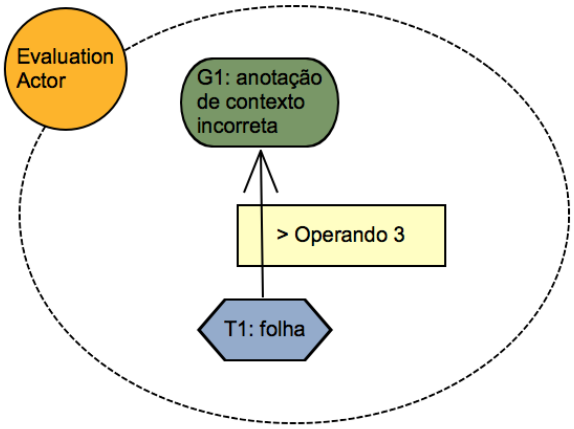

Figura 3.6: Ilustração de modelo para o caso de teste 6

(b) Caso de teste 8: Tarefa que não possui nó filho mas contém anotação de tempo de execução

- Entrada (Figura 3.8): Tarefa-folha (filha única do objetivo de *label* "G1: erro na tarefa"), com a seguinte *label*: "T1: folha [T1.1@2]".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

4. Objetivos e/ou tarefas que contenham apenas um nó filho

(a) Caso de teste 9: Objetivo que possui apenas um nó filho e apresenta anotação de tempo de execução inválida.

- Entrada (Figura 3.9): Objetivo contendo um nó filho (T1 nó-folha), com a seguinte *label*: "G1: um filho [try(T1)?skip:T2]".

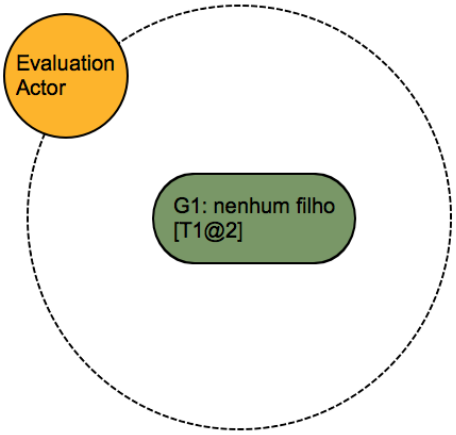

Figura 3.7: Ilustração de modelo para o caso de teste 7

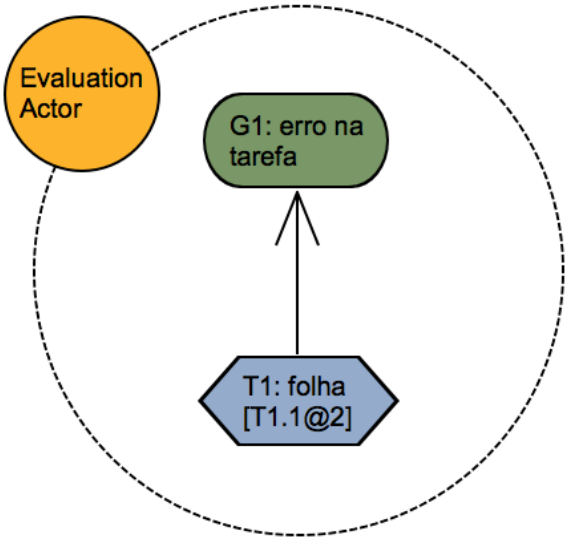

Figura 3.8: Ilustração de modelo para o caso de teste 8

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 10: Tarefa que possui apenas um nó filho e apresenta anotação de tempo de execução inválida.

- Entrada (Figura 3.10): Tarefa-folha (filha única do objetivo de *label* "G1: erro na tarefa") que possui um filho (T1.1 nó folha), com a seguinte *label*: "T1: um filho [T1.1#T1.2]".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

5. Objetivos e/ou tarefas que contenham dois ou mais nós filhos

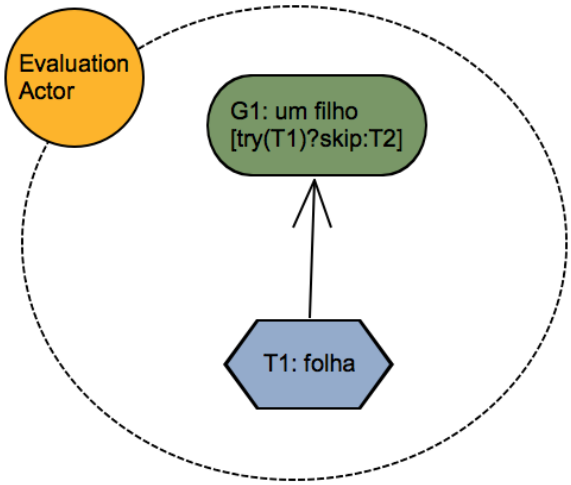

Figura 3.9: Ilustração de modelo para o caso de teste 9

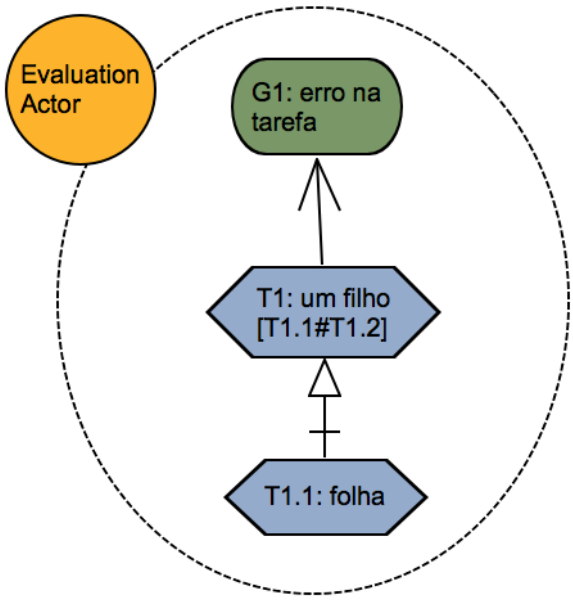

Figura 3.10: Ilustração de modelo para o caso de teste 10

(a) Caso de teste 11: Objetivo que possui dois nós filhos mas não apresenta anotação de tempo de execução.

- Entrada (Figura 3.11): Objetivo contendo dois nós filhos (G2 e G3 nós-folha), com a seguinte *label*: "G1: dois filhos".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 12: Objetivo que possui quatro nós filhos mas não apresenta anotação de tempo de execução.

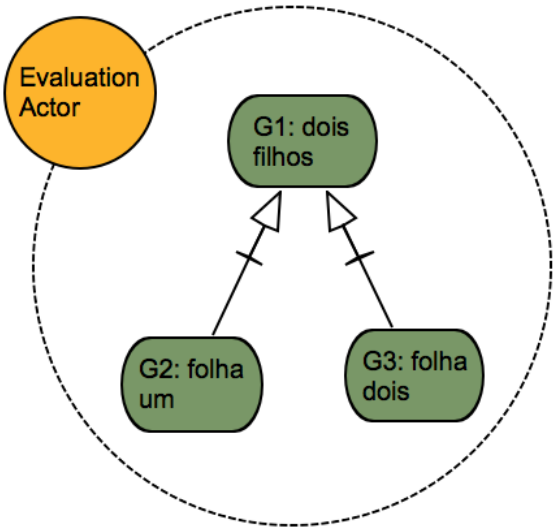

Figura 3.11: Ilustração de modelo para o caso de teste 11

- Entrada (Figura 3.12): Objetivo contendo quatro nós filhos (G1, G2, G3 e G4 nós-folha), com a seguinte *label*: "G0: quatro filhos".

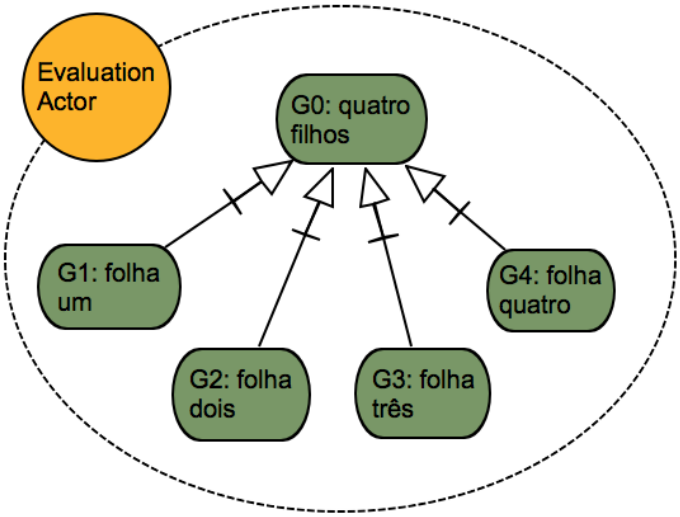

Figura 3.12: Ilustração de modelo para o caso de teste 12

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(c) Caso de teste 13: Tarefa que possui dois nós filhos mas não apresenta anotação de tempo de execução.

- Entrada (Figura 3.13): Tarefa (filha única do objetivo de *label* "G1: um filho") contendo dois filhos (T1.1 e T1.2 nós-folha), com a seguinte *label*: "T1: dois filhos".

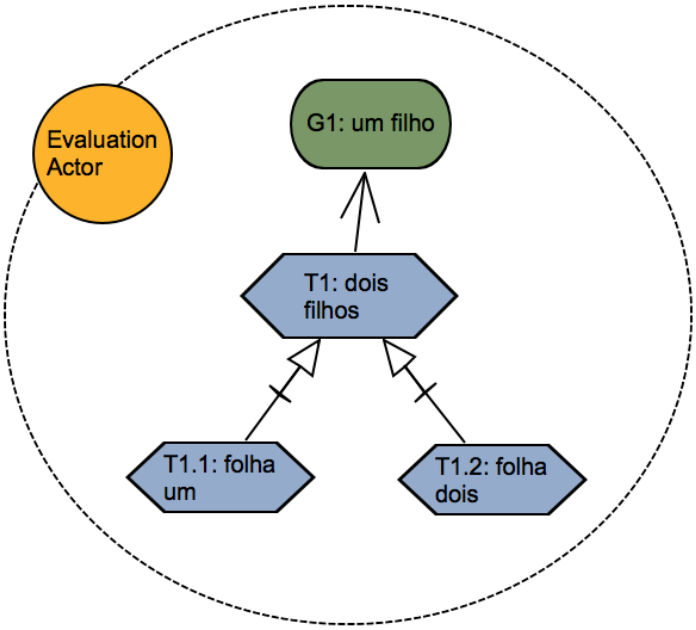

Figura 3.13: Ilustração de modelo para o caso de teste 13

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(d) Caso de teste 14: Tarefa que possui quatro nós filhos mas não apresenta anotação de tempo de execução.

- Entrada (Figura 3.14): Tarefa (filha única do objetivo de *label* "G1: um filho") contendo quatro filhos (T1.1, T1.2, T1.3 e T1.4 nós-folha), com a seguinte *label*: "T1: quatro filhos".

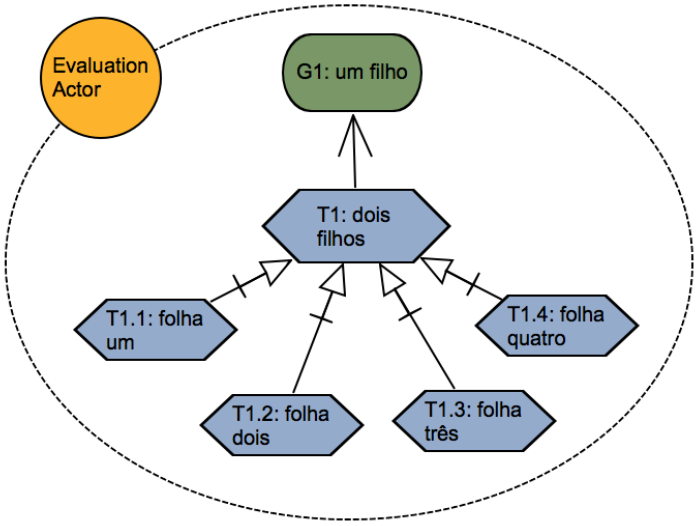

Figura 3.14: Ilustração de modelo para o caso de teste 14

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

6. Objetivos (dois ou mais) que contenham o mesmo identificador

   O TAOM4E previne que o mesmo identificador seja posto em dois ou mais objetivos caso as descrições também sejam as mesmas; caso contrário, ele permite. Logo, o caso teste utilizará descrições diferentes.

   (a) Caso de teste 15: Dois objetivos com o mesmo identificador

   - Entrada (Figura 3.15): Dois objetivos sem filhos com *labels* "G1: primeiro"e "G1: segundo".

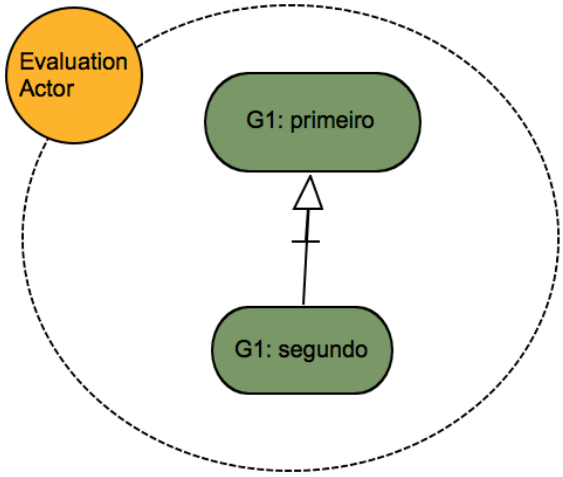

Figura 3.15: Ilustração de modelo para o caso de teste 15

   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

   (b) Caso de teste 16: Quatro objetivos com o mesmo identificador

   - Entrada (Figura 3.16): Quatro objetivos sem filhos com *labels* "G1: primeiro", "G1: segundo", "G1: terceiro"e "G1: quarto".
   - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

7. Tarefas irmãs (duas ou mais) que contenham o mesmo identificador

   O TAOM4E previne que o mesmo identificador seja posto em duas ou mais tarefas caso as descrições também sejam as mesmas; caso contrário, ele permite. Logo, o caso teste utilizará descrições diferentes.

   (a) Caso de teste 17: Duas tarefas irmãs de nível de profundidade três com o mesmo identificador

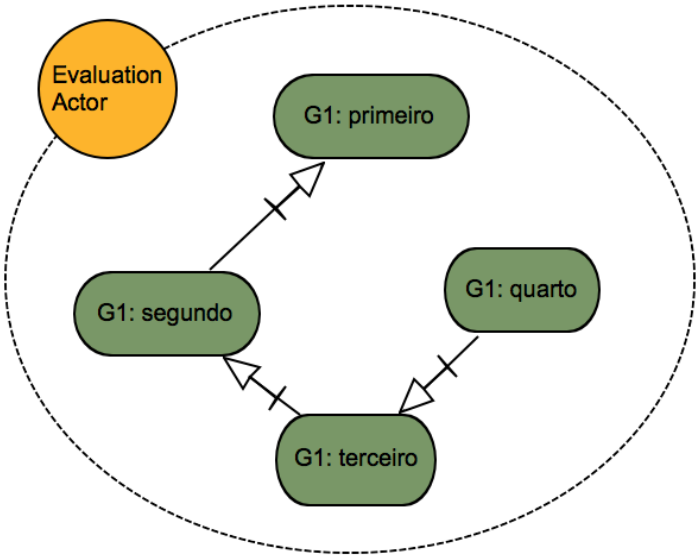

Figura 3.16: Ilustração de modelo para o caso de teste 16

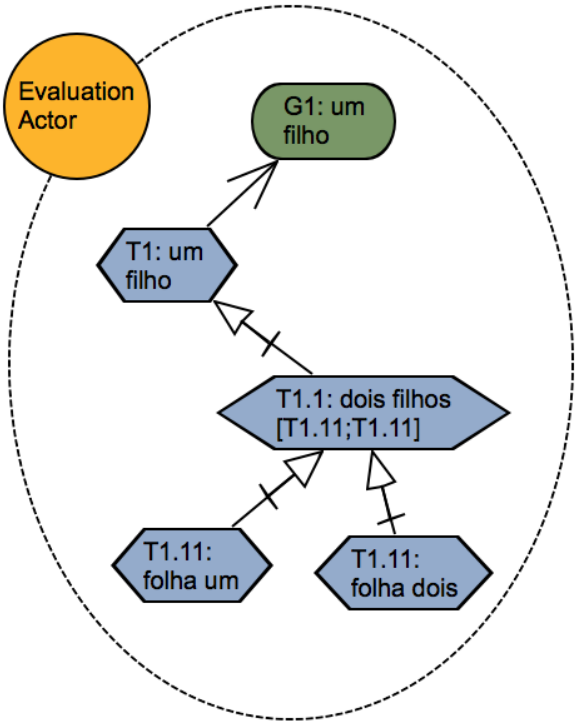

Figura 3.17: Ilustração de modelo para o caso de teste 17

- Entrada (Figura 3.17): Duas tarefas irmãs (filhas da tarefa de *label* "T1.1: dois filhos [T1.11;T1.11]") com *labels* "T1.11: folha um"e "T1.11: folha dois".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 18: Três tarefas irmãs de nível de profundidade dois com o mesmo identificador

- Entrada (Figura 3.18): Três tarefas irmãs (filhas da tarefa de *label* "T1: três filhos [T1.1;T1.1;T1.1]") com *labels* "T1.1: folha um", "T1.1: folha dois"e "T1.1: folha três".

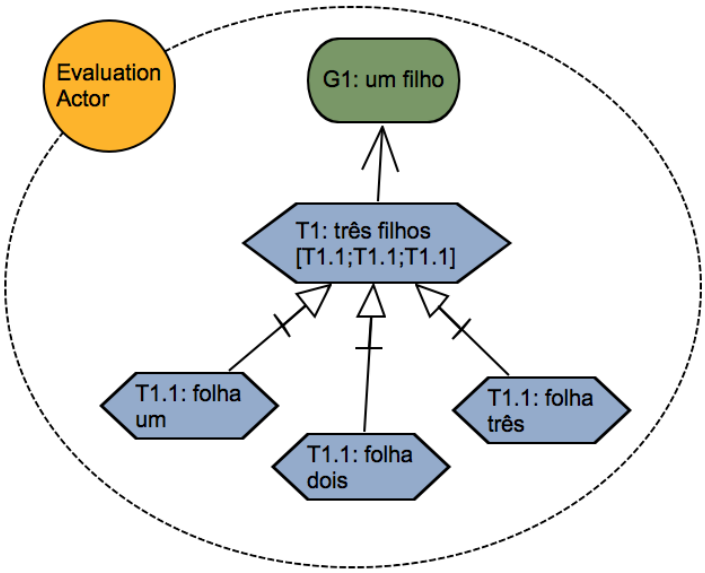

Figura 3.18: Ilustração de modelo para o caso de teste 18

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

8. Tarefas de nível de profundidade dois ou mais com identificadores fora do padrão

(a) Caso de teste 19: Tarefa de nível de profundidade dois com identificador escrito fora do padrão

- Entrada (Figura 3.19): Tarefa-folha de nível de profundidade dois (filha da tarefa de *label* "T1: um filho de identificador incorreto") com *label* "T11: folha".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 20: Tarefa de nível de profundidade três com identificador escrito fora do padrão

- Entrada (Figura 3.20): Tarefa-folha de nível de profundidade três (filha da tarefa de *label* "T1.1: uma filho de identificador incorreto") com *label* "T1.21: folha".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

9. Anotação de tempo de execução não identificável

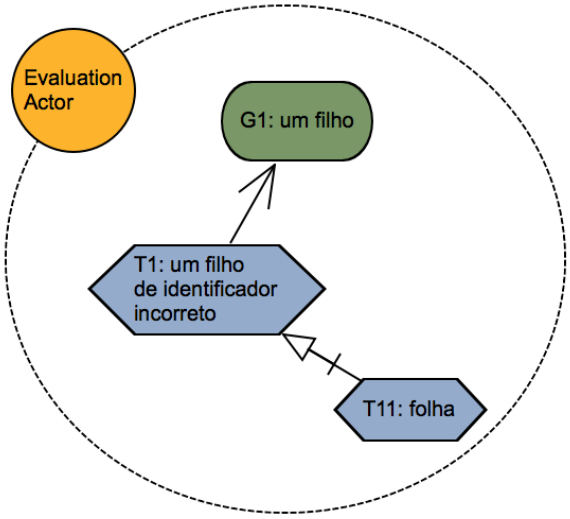

Figura 3.19: Ilustração de modelo para o caso de teste 19

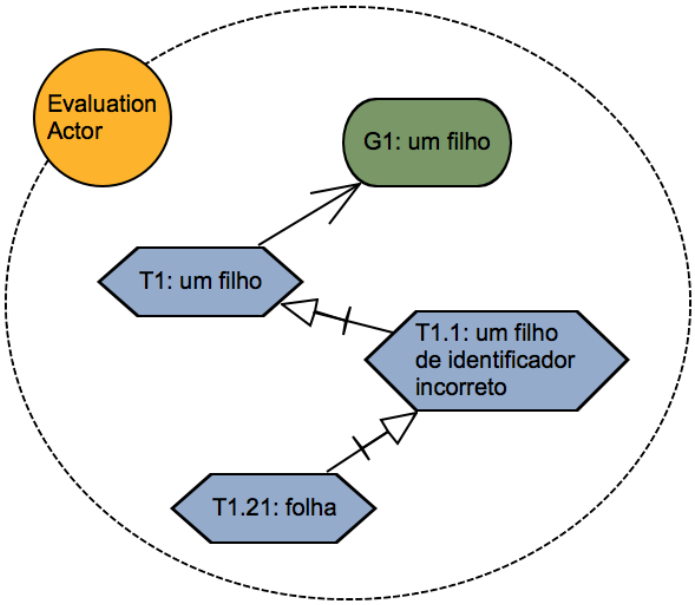

Figura 3.20: Ilustração de modelo para o caso de teste 20

(a) Caso de teste 21: Anotação de tempo de execução fora de colchetes

- Entrada (Figura 3.21): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: sem colchetes G1;G2".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 22: Anotação de tempo de execução escrito antes da descrição do objetivo/tarefa

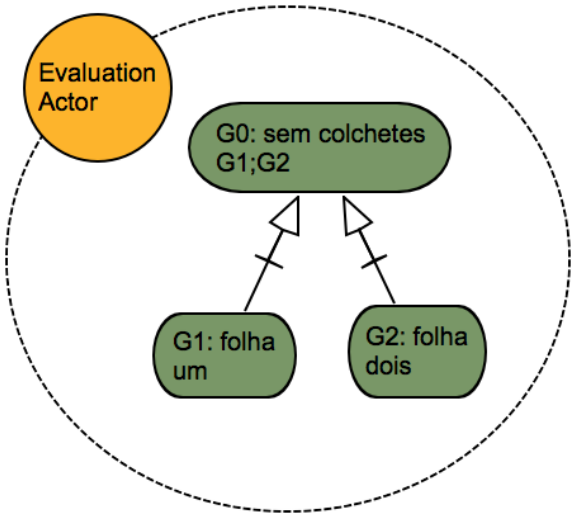

Figura 3.21: Ilustração de modelo para o caso de teste 21

- Entrada (Figura 3.22): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: [G1;G2] lugar errado".

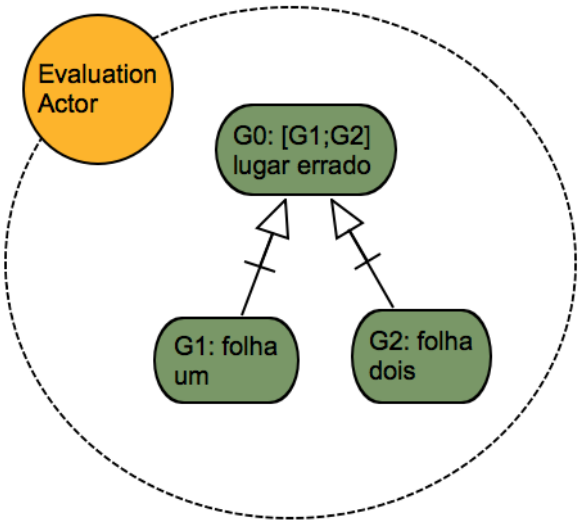

Figura 3.22: Ilustração de modelo para o caso de teste 22

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

10. Anotação de tempo de execução referenciando nós não filhos

    (a) Caso de teste 23: Anotação de tempo de execução com referência à um nó não filho

    - Entrada (Figura 3.23): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: dois filhos [G3;G1]".

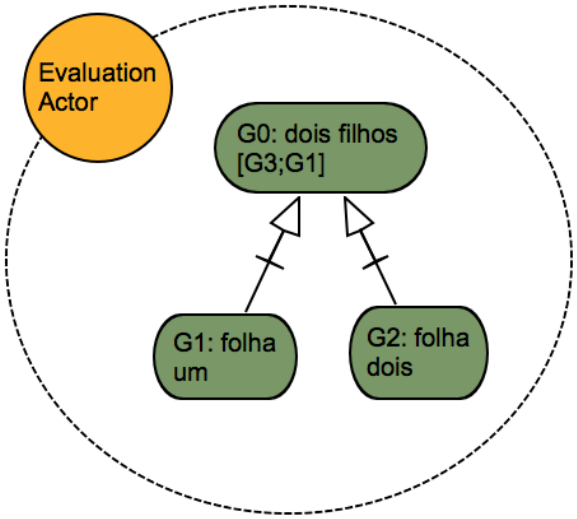

Figura 3.23: Ilustração de modelo para o caso de teste 23

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 24: Anotação de tempo de execução com referência a mais de um nó não filho

- Entrada (Figura 3.24): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: dois filhos [G3;G4]".

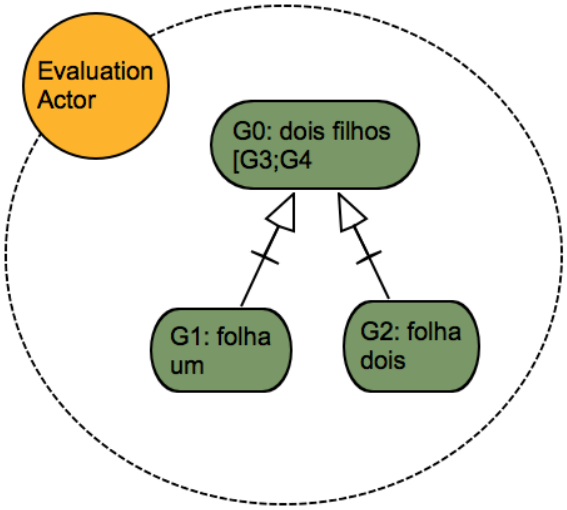

Figura 3.24: Ilustração de modelo para o caso de teste 24

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

11. Anotação de tempo de execução escrita fora do padrão

(a) Caso de teste 25: Expressão [E1;E2] fora do padrão

- Entrada (Figura 3.25): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: dois filhos [G1;]".

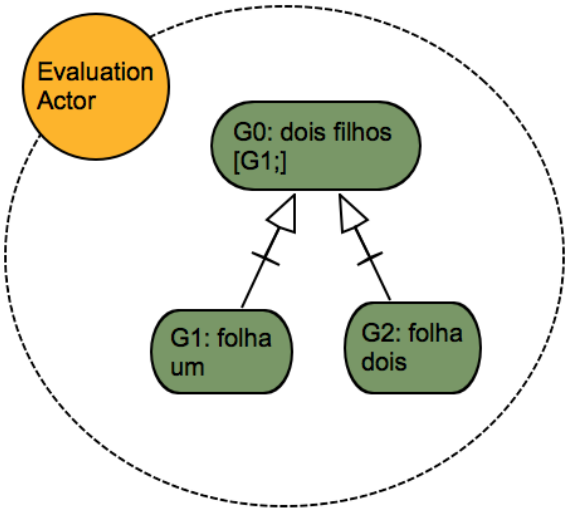

Figura 3.25: Ilustração de modelo para o caso de teste 25

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 26: Expressões [E1#E2] e [E#n] fora do padrão

- Entrada (Figura 3.26): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: dois filhos [G1#]".

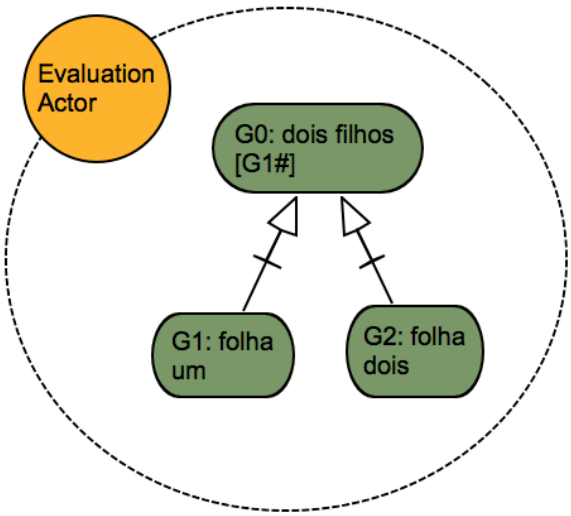

Figura 3.26: Ilustração de modelo para o caso de teste 26

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(c) Caso de teste 27: Expressão [E+n] fora do padrão

- Entrada (Figura 3.27): Objetivo com um nó filho (T1 nó-folha) com *label* "G1: um filho [T1+]".

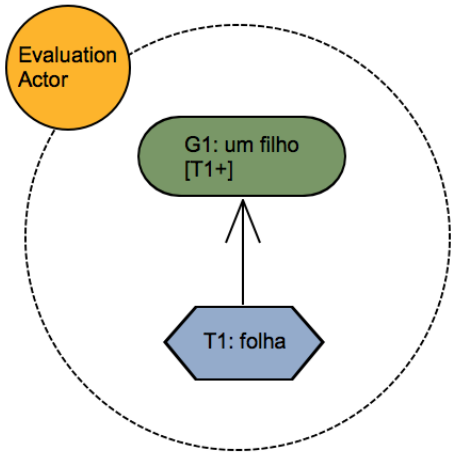

Figura 3.27: Ilustração de modelo para o caso de teste 27

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(d) Caso de teste 28: Expressão [E@n] fora do padrão

- Entrada (Figura 3.28): Objetivo com um nó filho (T1 nó-folha) com *label* "G1: um filho [T1@]".

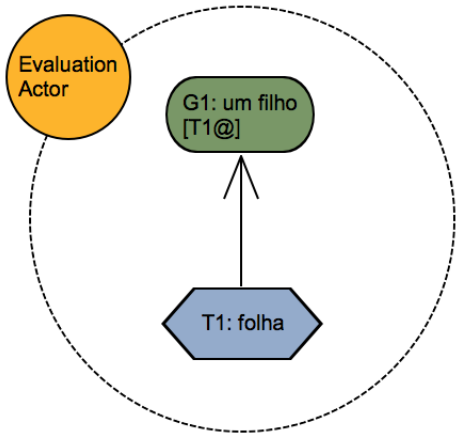

Figura 3.28: Ilustração de modelo para o caso de teste 28

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(e) Caso de teste 29: Expressão [E1|E2] fora do padrão

- Entrada (Figura 3.29): Objetivo com dois nós filhos (G1 e G2 nós-folha) com *label* "G0: dois filhos [G1|]".

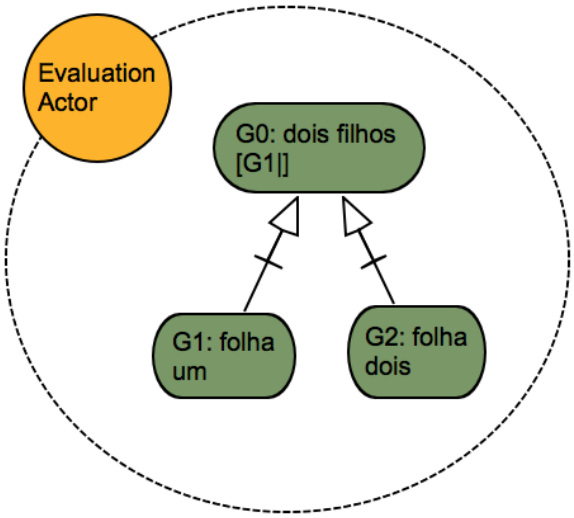

Figura 3.29: Ilustração de modelo para o caso de teste 29

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(f) Caso de teste 30: Expressão [opt(E)] fora do padrão

- Entrada (Figura 3.30): Objetivo com um nó filho (T1 nó-folha) com *label* "G1: um filho [opt()]".

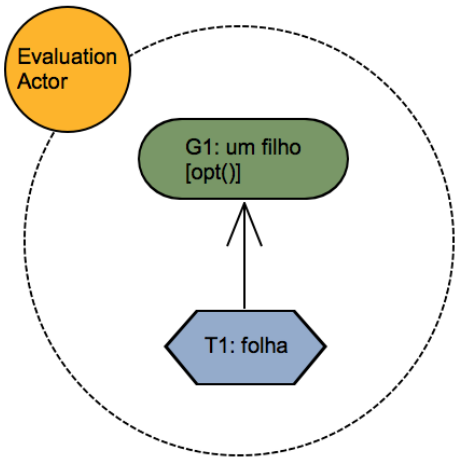

Figura 3.30: Ilustração de modelo para o caso de teste 30

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(g) Caso de teste 31: Expressão [try(E)?E1:E2] fora do padrão

- Entrada (Figura 3.31): Tarefa com três nós filhos (T1.1, T1.2 e T1.3 nós-folha) com *label* "T1: três filhos [try(T1.1)?T1.2]".
- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

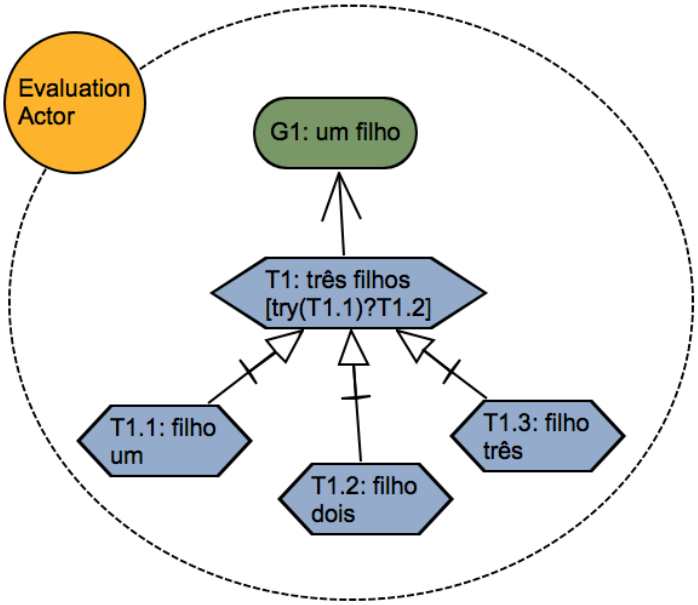

Figura 3.31: Ilustração de modelo para o caso de teste 31

12. Anotação de contexto com operando e/ou operador PRISM inválidos

    (a) Caso de teste 32: Operando de anotação de contexto iniciando com dígito.

    - Entrada (Figura 3.32): Anotação de contexto de *label*: "2Operando = 7".

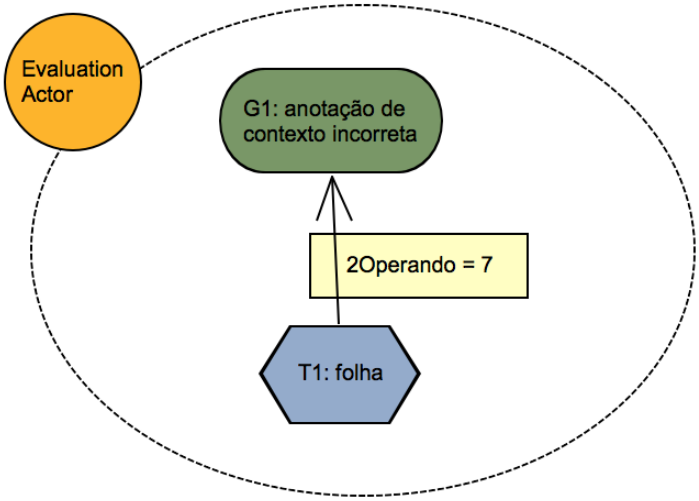

Figura 3.32: Ilustração de modelo para o caso de teste 32

    - Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

    (b) Caso de teste 33: Operador PRISM de anotação de contexto inválido

- Entrada (Figura 3.33): Anotação de contexto de *label*: "Operando + 7".

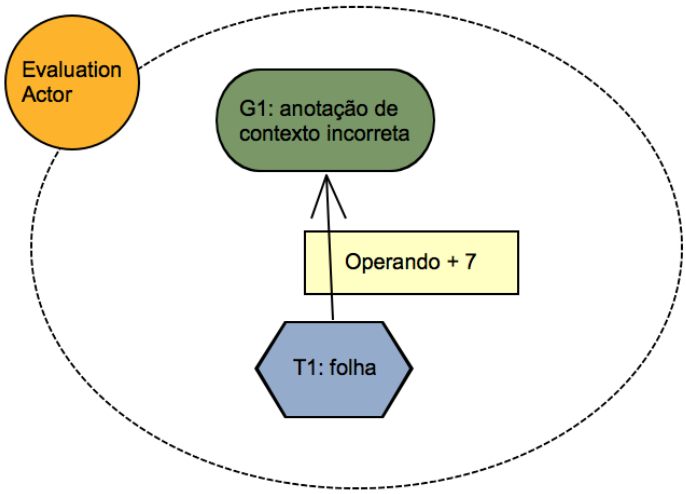

Figura 3.33: Ilustração de modelo para o caso de teste 33

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

13. Anotação de contexto com valores inválidos

(a) Caso de teste 34: Anotação de contexto com valor de comparação do tipo *string*

- Entrada (Figura 3.34): Anotação de contexto de *label*: "Operando != valor".

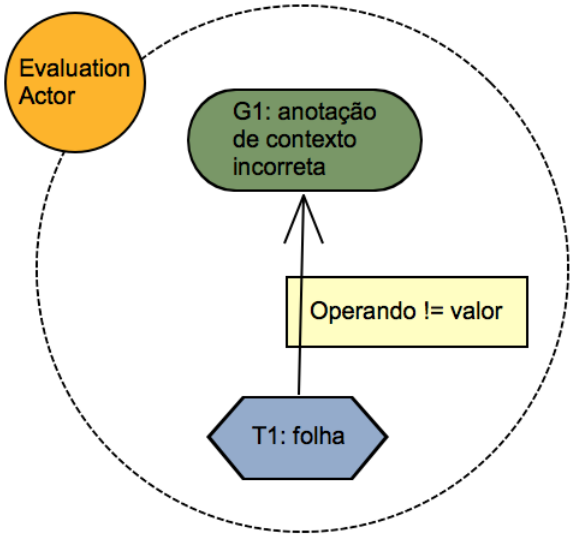

Figura 3.34: Ilustração de modelo para o caso de teste 34

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

(b) Caso de teste 35: Anotação de contexto com valor de comparação do tipo *char*

- Entrada (Figura 3.35): Anotação de contexto de *label*: "Operando >= A".

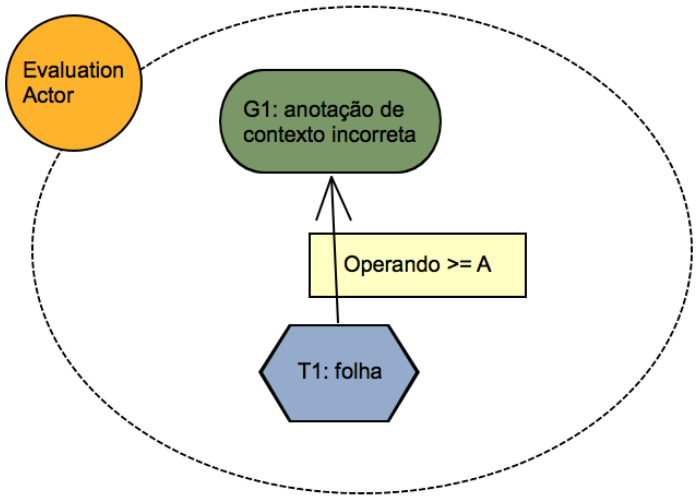

Figura 3.35: Ilustração de modelo para o caso de teste 35

- Saída esperada: Lançamento de exceção Java, impedindo a finalização da execução do GODA.

3.4 Testes Concretos

Os testes concretos são a implementação e execução dos testes abstratos, utilizando as entradas e saídas definidas para cada caso de teste. A implementação da suíte de testes proposta neste trabalho foi feita utilizando a linguagem de programação Java. Para automatizar a execução de tais testes, foi utilizada a ferramenta JUnit, já discutida neste capítulo. A suíte de testes está integrada à implementação do GODA, e pode ser encontrada no repositório do GitHub[2].

Para cada classe de equivalência identificada anteriormente, foi criada uma *JUnit Test Case* no pacote *br.unb.cic.functional*, que se encontra no diretório *src/test/java* do GODA. Cada *JUnit Test Case* contém um conjunto de casos de teste, assim como foi definido na fase de testes abstratos. A Tabela 3.2 mostra todas as classes criadas, e os casos de teste implementados em cada uma.

A seguir serão apresentas as etapas de desenvolvidos dos casos de teste concretos. Primeiro serão apresentados os padrões de implementação utilizados nos testes válidos e inválidos. Em seguida, será apresentada a preparação do modelo CRGM para ser utilizado na execução do GODA. Por fim, será explicado como é feita a chamada para a transformação de modelos no GODA.

[2]O repositório do GODA pode ser acessado em https://github.com/lesunb/CRGMToPRISM

Tabela 3.2: Classes Java e casos de teste

JUnit Test Case	**Casos de Teste**
CorrectModelTest	1 e 2
LabelPatternTest	3, 4, 5 e 6
NoChildTest	7 e 8
OneChildTest	9 e 10
MultipleChildrenTest	11, 12, 13 e 14
SameIdentifierGoalTest	15 e 16
SameIdentifierTaskTest	17 e 18
MultipleLevelTaskIdentifierTest	19 e 20
NoRuntimeAnnotationTest	21 e 22
RuntimeAnnotationReferenceTest	23 e 24
RuntimeAnnotationPatternTest	25, 26, 27, 28, 29, 30 e 31
OperandOperatorTest	32 e 33
ContextAnnotationValueTest	34 e 35

3.4.1 Modelo de Implementação

A saída esperada de cada caso de teste é a exeistência (ou não) de uma *Java Exception*. Desta maneira, os testes foram implementados utilizando blocos *Try/Catch*.

Em casos de teste válidos, a captura de uma exceção indica que o teste falhou, ou seja, podem haver problemas de implementação no programa testado; o teste finaliza com sucesso caso toda a execução do programa aconteça sem lançamentos de exceções. A Figura 3.36 exemplifica o modelo de implementação para casos de teste válidos.

Em casos de teste inválidos, o esperado da execução do programa é a captura de uma exceção, ou seja, são nesses casos que os testes finalizam com sucesso. Caso a execução do programa aconteça de forma correta, o teste falha indicando que nenhuma exceção foi encontrada. A Figura 3.37 exemplifica o modelo de implementação para casos de teste inválidos.

```
private AgentDefinition agentDefinition;

@Test
public void TestCase1() {
    String goalName = "G0:_dois_filhos_[G1#G2]";
    String contextAnnotation = "assertion condition OPERANDO = true\n";

    //id, branch, depth
    String[] elementsName = {"G1:_um_filho", "1", "1",
            "T1:_três_filhos_[try(T1.1)?T1.2:T1.3]", "1", "2",
            "T1.1:_um_filho_[opt(T1.11)]", "1", "3",
            "T1.11:_folha", "1", "4",
            "T1.2:_dois_filhos_[T1.21#T1.22]", "2", "3",
            "T1.21:_folha_um", "1", "4",
            "T1.22:_folha_dois", "2", "4",
            "T1.3:_folha", "3", "3",
            "G2:_um_filho_[T1@2]", "2", "1",
            "T1:_retry_duas_vezes", "1", "2"};

    //Add elements to new register
    InformationRegister[] info = new InformationRegister[elementsName.length/3];
    createRegister(elementsName, info);

    try{

        //Create CRGM java model
        CRGMTestProducer producer = new CRGMTestProducer(5, 2, 3, "EvaluationActor");
        agentDefinition = producer.generateCRGM(goalName, info, contextAnnotation);

        //Run Goda
        producer.run(agentDefinition);
    }catch(Exception e){
        fail("Exception found: " + e);
    }
}
```

Figura 3.36: Implementação de um caso de teste válido

3.4.2 Preparação de Modelos CRGM

No GODA, os modelos são construídos no ambiente de modelagem TAOM4E. Iniciada a execução do *framework*, o *Tropos Navigator* é instanciado com o arquivo de entrada do TROPOS, e acessa o modelo de negócios do *plugin* TAOM4E , recuperando os atores do no arquivo. A partir daí, os elementos do modelo são extraídos e se inicia uma chamada recursiva do método *addGoal()* para a criação da árvore que representa o modelo, e para a análise e escrita do modelo em linguagens PRISM e PARAM.

Para a atividade de teste, modelos são construídos programaticamente. A classe *CRGMTestProducer* contém métodos que auxiliam a geração de modelos. Em um primeiro momento, um objeto *CRGMTestProducer* é instanciado, passando como parâmetros a profundidade do modelo, a profundidade dos objetivos do modelo, a quantidade máxima de ramos existentes e o nome do ator (ou agente). Então, o modelo é gerado e atribuído à uma variável do tipo *AgentDefinition*, que mantém os elementos do modelo para um agente específico. O método *generateCRGM()* é responsável pela criação do modelo.

```
private AgentDefinition agentDefinition;

@Test
public void TestCase3() throws IOException {

    String goalName = "Objetivo_principal:_testar_erro_de_padrão_[G1;G2]";

    //id, branch, depth
    String[] elementsName = {"G1:_folha_um", "1", "1",
            "G2:_folha_dois", "2", "1"};

    //Add elements to new register
    InformationRegister[] info = new InformationRegister[elementsName.length/3];
    createRegister(elementsName, info);

    try{

        //Create CRGM java model
        CRGMTestProducer producer = new CRGMTestProducer(2, 2, 2, "EvaluationActor");
        agentDefinition = producer.generateCRGM(goalName, info, null);

        //Run Goda
        producer.run(agentDefinition);

        fail("No exception found.");
    }catch(Exception e){
        assertNotNull(e);
    }
}
```

Figura 3.37: Implementação de um caso de teste inválido

Os casos de teste definidos neste trabalhos possuem modelos CRGM de entrada únicos, logo, em todo início de caso de teste concreto um modelo específico é criado.

3.4.3 Execução do GODA

O método *run()* é utilizado para chamar a execução do GODA a partir de um *CRGMTestProducer*. Ele foi implementado com base no próprio método que normalmente executa a transformação de modelos. A diferença, no entanto, é que para a atividade de teste, o novo modelo DTMC não é escrito. Como o foco do trabalho está nos problemas de anotações do modelo CRGM, apenas a etapa de análise do modelo é realizada. É nessa etapa que os elementos do modelo são verificados. Logo, todos os elementos (objetivos, tarefas, anotações de contexto e anotações de tempo de execução) são analisados nos métodos *addGoal()* e *addPlan()*. Esses dois últimos métodos também foram implementados com base nos métodos de mesmo nome utilizados pelo GODA, deixando de lado código de criação da árvore que representa o modelo CRGM.

Capítulo 4

Resultados Obtidos

Este capítulo apresentará os resultados obtidos com a execução de cada caso de teste, junto com uma discussão relacionada a cada um dos resultados. Também serão apresentadas as lições aprendidas com o desenvolvimento do trabalho e as dificuldades encontradas no processo.

A opção *Run As* → *JUnit Test* do Eclipse foi utilizada para executar a suíte de testes. A Tabela 4.1 cita cada *JUnit Test Case* criada, e lista quais dos casos de teste implementados obtiveram sucesso e quais falharam.

CorrectModelTest

O caso de teste 1 (um) obteve sucesso (Figura 4.1), enquanto que o caso de teste 2 (dois) falhou (Figura 4.2). Ambos apresentam modelos CRGM de entrada válidos (de acordo com a especificação do GODA), com objetivos e tarefas diversos, anotação de contexto e anotações de tempo de execução. O sucesso no primeiro caso de teste indica que o processo de transformação de modelos CRGM em DTMC obedece todas as regras utilizadas neste modelo. A falha no segundo caso de teste indica alguma validação que ocorre de maneira equivocada. No caso, a anotação de tempo de execução $T1\#3$ é considerada inválida, apesar de a especificação constatá-la como uma anotação válida (vide Tabela 2.1).

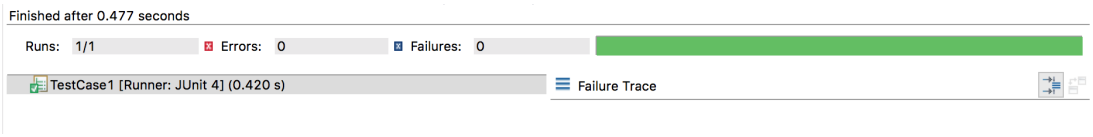

Figura 4.1: Execução do caso de teste 1

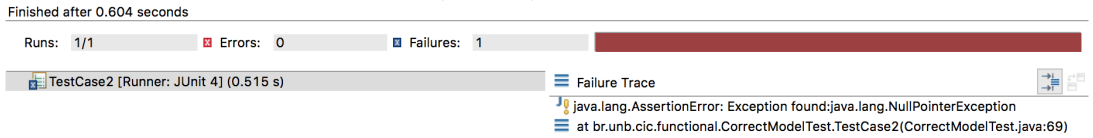

Figura 4.2: Execução do caso de teste 2

Tabela 4.1: Resultado da execução dos casos de teste

JUnit Test Case	**Sucesso**	**Falha**
CorrectModelTest	1	2
LabelPatternTest	-	3, 4, 5 e 6
NoChildTest	-	7 e 8
OneChildTest	-	9 e 10
MultipleChildrenTest	-	11, 12, 13 e 14
SameIdentifierGoalTest	-	15 e 16
SameIdentifierTaskTest	-	17 e 18
MultipleLevelTaskIdentifierTest	-	19 e 20
NoRuntimeAnnotationTest	-	21 e 22
RuntimeAnnotationReferenceTest	-	23 e 24
RuntimeAnnotationPatternTest	25, 26, 27, 28, 29, 30 e 31	-
OperandOperatorTest	-	32 e 33
ContextAnnotationValueTest	-	34 e 35

LabelPatternTest

Todos os casos de teste falharam (Figura 4.3, Figura 4.4, Figura 4.5 e Figura 4.6). Todos apresentam modelos CRGM de entrada inválidos, com objetivos, tarefas e anotações de contexto contendo *labels* inválidas, que não obedecem a especificação do *framework*. As falhas indiciam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta das *labels* dos elementos do modelo CRGM.

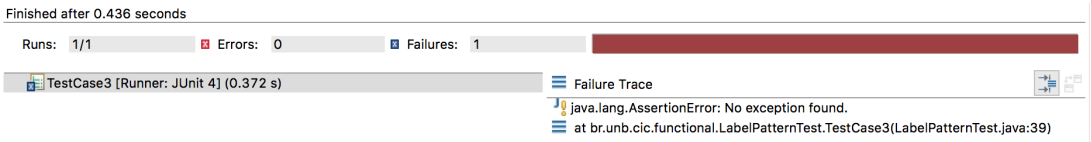

Figura 4.3: Execução do caso de teste 3

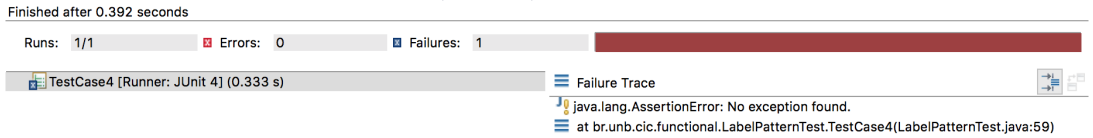

Figura 4.4: Execução do caso de teste 4

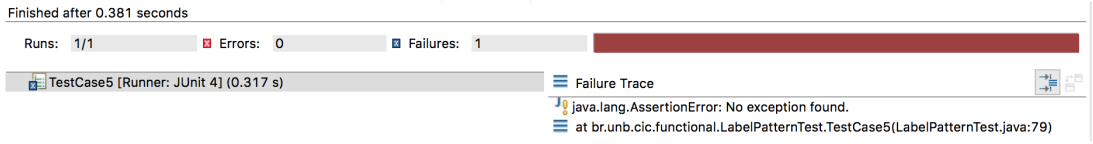

Figura 4.5: Execução do caso de teste 5

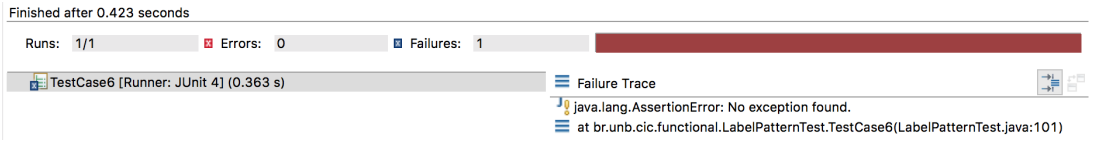

Figura 4.6: Execução do caso de teste 6

NoChildTest

Todos os casos de teste falharam (Figura 4.7 e Figura 4.8). Ambos apresentam modelos CRGM de entrada inválidos, com objetivos e/ou tarefas que não possuam nós filhos mas que apresentam alguma anotação de tempo de execução. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta da anotação de tempo de execução nesse caso específico.

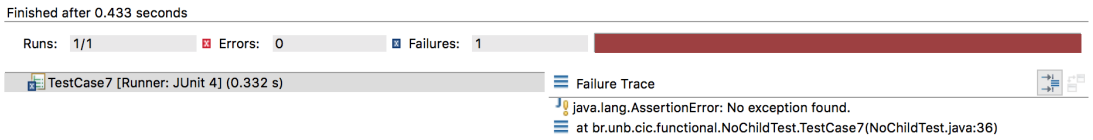

Figura 4.7: Execução do caso de teste 7

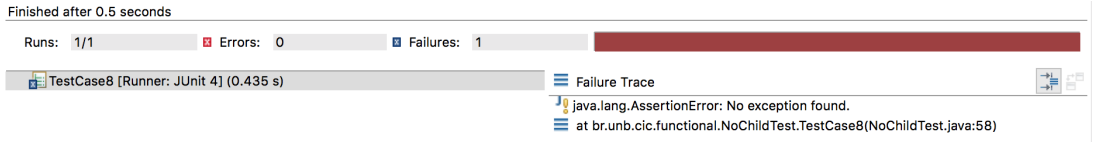

Figura 4.8: Execução do caso de teste 8

OneChildTest

Todos os casos de teste falharam (Figura 4.9 e Figura 4.10). Ambos apresentam modelos CRGM de entrada inválidos, com objetivos e/ou tarefas que contenham apenas um nó filho e cujas anotações de tempo de execução sejam incorretas. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta da anotação de tempo de execução nesse caso específico.

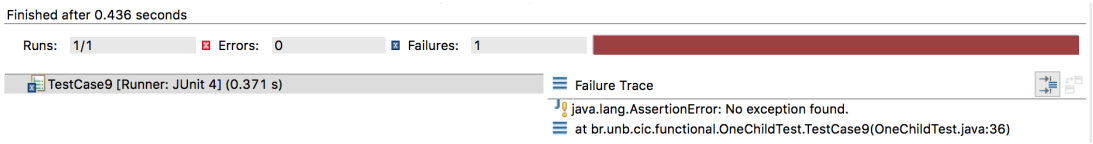

Figura 4.9: Execução do caso de teste 9

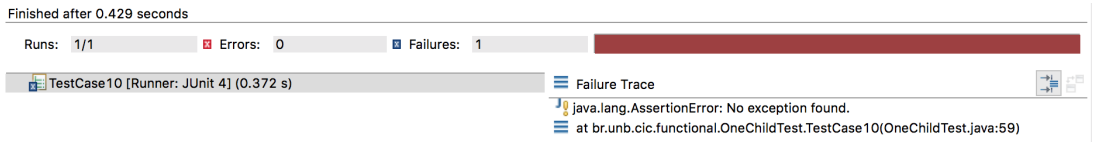

Figura 4.10: Execução do caso de teste 10

MultipleChildrenTest

Todos os casos de teste falharam (Figura 4.11, Figura 4.12, Figura 4.13 e Figura 4.14). Todos apresentam modelos CRGM de entrada inválidos, com objetivos e/ou tarefas que contenham apenas dois ou mais nós filhos mas que não possuam anotação de tempo de execução. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta da anotação de tempo de execução nesse caso específico.

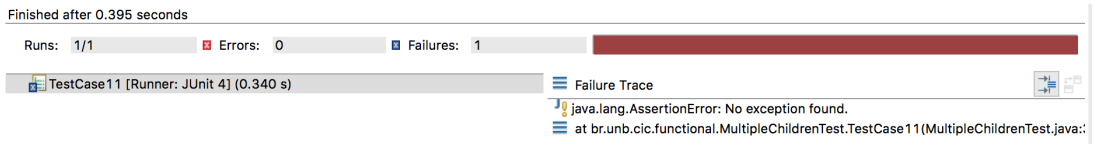

Figura 4.11: Execução do caso de teste 11

SameIdentifierGoalTest

Todos os casos de teste falharam (Figura 4.15 e Figura 4.16). Ambos apresentam modelos CRGM de entrada inválidos, com dois ou mais objetivos contendo o mesmo identificador. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta do identificador dos objetivos nesse caso específico.

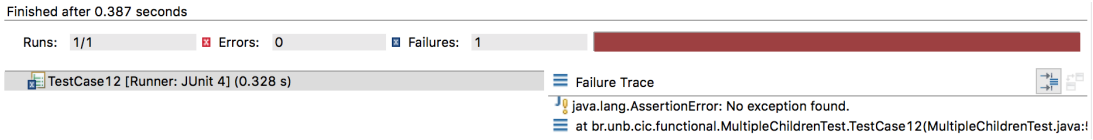

Figura 4.12: Execução do caso de teste 12

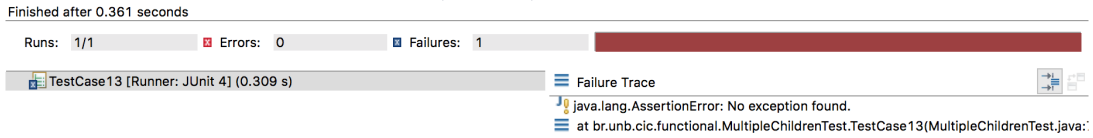

Figura 4.13: Execução do caso de teste 13

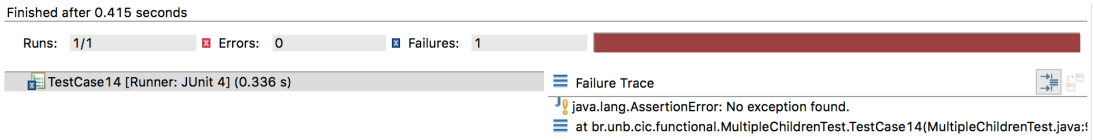

Figura 4.14: Execução do caso de teste 14

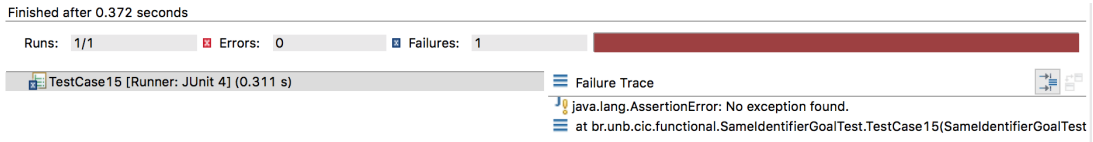

Figura 4.15: Execução do caso de teste 15

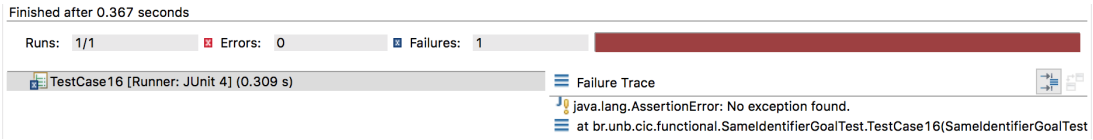

Figura 4.16: Execução do caso de teste 16

SameIdentifierTaskTest

Todos os casos de teste falharam (Figura 4.17 e Figura 4.18). Ambos apresentam modelos CRGM de entrada inválidos, com duas ou mais tarefas-irmãs com o mesmo identificador. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta do identificador das tarefas nesse caso específico.

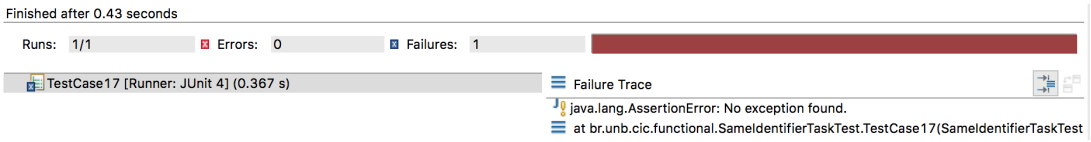

Figura 4.17: Execução do caso de teste 17

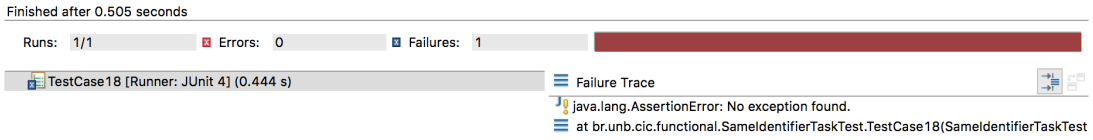

Figura 4.18: Execução do caso de teste 18

MultipleLevelTaskIdentifierTest

Todos os casos de teste falharam (Figura 4.19 e Figura 4.20). Ambos apresentam modelos CRGM de entrada inválidos, com tarefas de nível dois ou mais que contenham identificadores incorretos. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta do identificador das tarefas nesse caso específico.

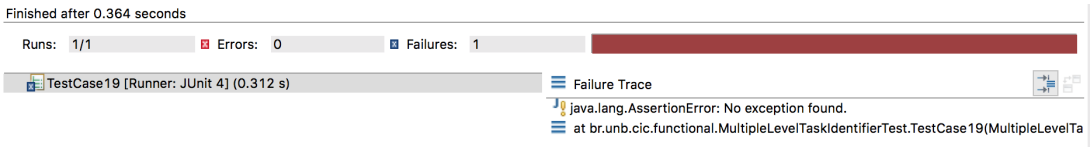

Figura 4.19: Execução do caso de teste 19

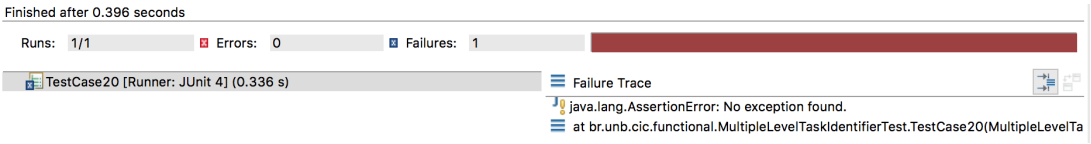

Figura 4.20: Execução do caso de teste 20

NoRuntimeAnnotationTest

Ambos os casos de teste falharam (Figura 4.21 e Figura 4.22). Todos apresentam modelos CRGM de entrada inválidos, com objetivos contendo dois nós filhos e cujas anotações de tempo de execução estejam escritas de forma não identificável (fora de colchetes, fora de ordem). As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta da anotações de tempo de execução nesse caso específico.

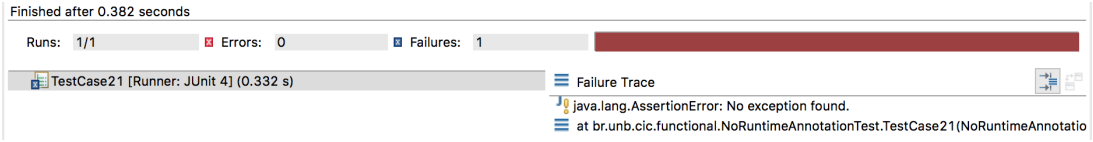

Figura 4.21: Execução do caso de teste 21

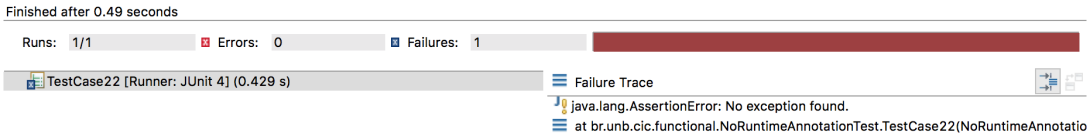

Figura 4.22: Execução do caso de teste 22

RuntimeAnnotationReferenceTest

Todos os casos de teste falharam (Figura 4.23 e Figura 4.24). Todos apresentam modelos CRGM de entrada inválidos, com objetivos contendo dois nós filhos e cujas anotações de tempo de execução estejam referenciando algum nó que não seja filho do objetivo em questão. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta da anotações de tempo de execução nesse caso específico.

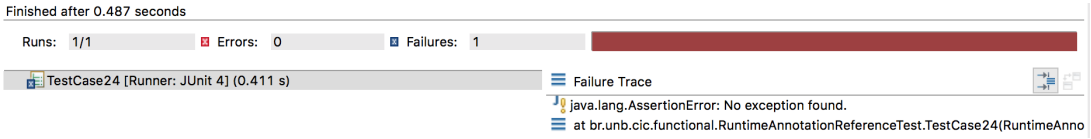

Figura 4.23: Execução do caso de teste 23

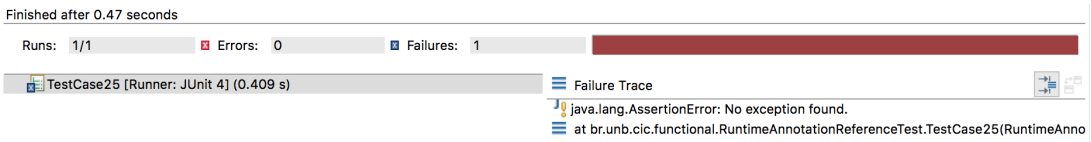

Figura 4.24: Execução do caso de teste 24

RuntimeAnnotationPatternTest

Todos os casos de teste obtiveram sucesso (Figura 4.25, Figura 4.26, Figura 4.27, Figura 4.28, Figura 4.29, Figura 4.30 e Figura 4.31). Todos apresentam modelos CRGM de entrada inválidos, com objetivos e/ou tarefas cujas anotações de tempo de execução estejam escrita fora das regras estabelecidas na Tabela 2.1. O sucesso indica que durante o processo de transformação de modelos CRGM em DTMC, a validação do cumprimento das regras de anotações de tempo de execução é realizada apropriadamente.

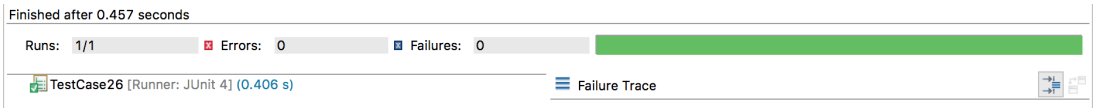

Figura 4.25: Execução do caso de teste 25

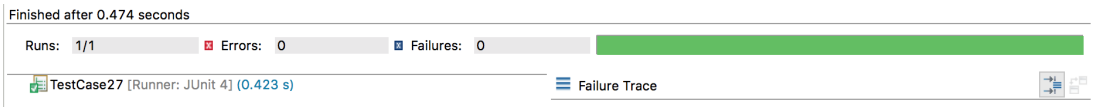

Figura 4.26: Execução do caso de teste 26

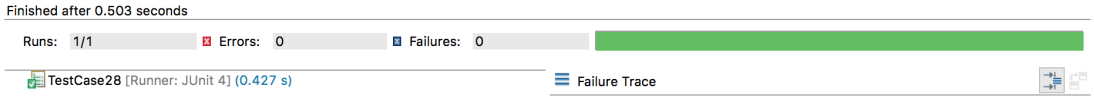

Figura 4.27: Execução do caso de teste 27

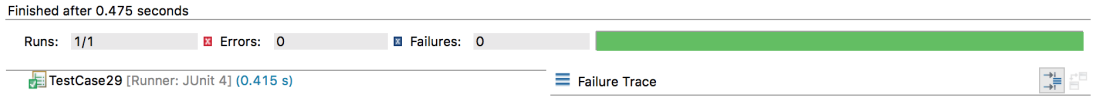

Figura 4.28: Execução do caso de teste 28

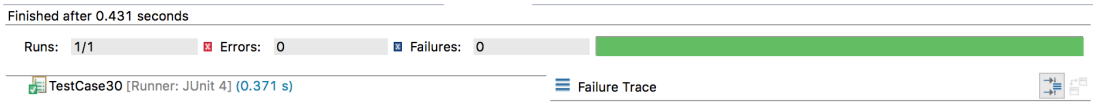

Figura 4.29: Execução do caso de teste 29

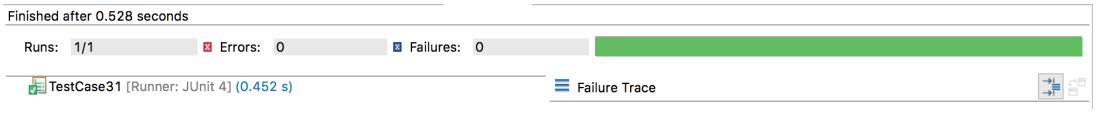

Figura 4.30: Execução do caso de teste 30

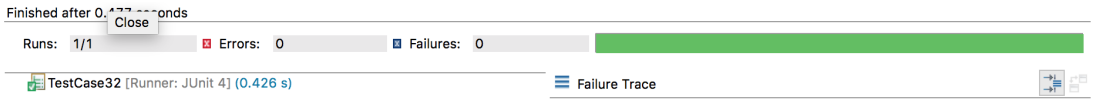

Figura 4.31: Execução do caso de teste 31

OperandOperatorTest

Todos os casos de teste falharam (Figura 4.32 e Figura 4.33). Ambos apresentam modelos CRGM de entrada inválidos, com anotações de contexto contendo operandos

e/ou operadores inválidos, de acordo com a especificação do GODA. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta dos operandos e operadores utilizadas em anotações de contexto.

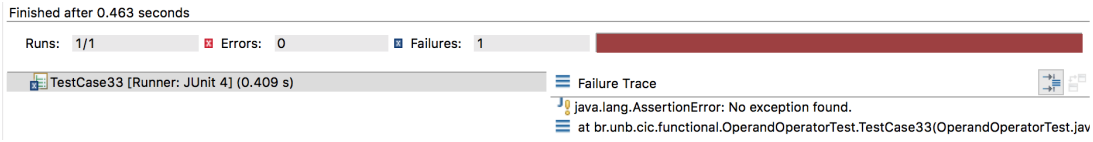

Figura 4.32: Execução do caso de teste 32

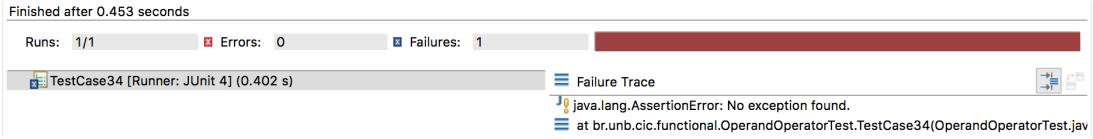

Figura 4.33: Execução do caso de teste 33

ContextAnnotationValueTest

Todos os casos de teste falharam (Figura 4.34 e Figura 4.35). Ambos apresentam modelos CRGM de entrada inválidos, com anotações de contexto contendo valores inválidos, de acordo com a especificação do GODA. As falhas indicam que durante o processo de transformação de modelos CRGM em DTMC, não é feita a validação correta dos valores utilizadas em anotações de contexto.

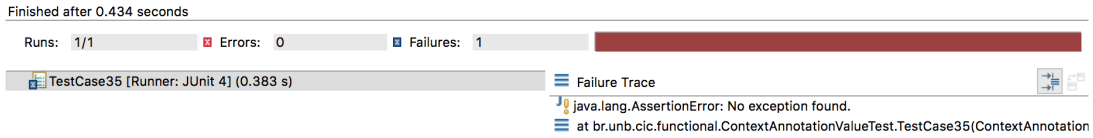

Figura 4.34: Execução do caso de teste 34

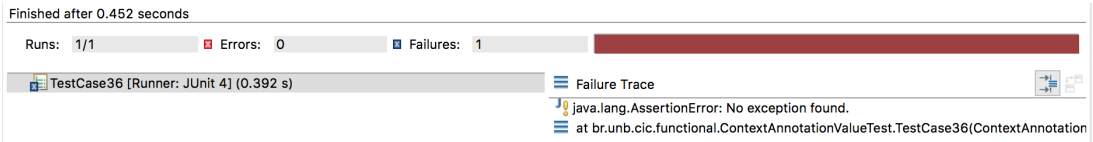

Figura 4.35: Execução do caso de teste 35

4.1 Lições Aprendidas e Dificuldades Encontradas

O final de todo trabalho implica em um aprendizado obtido com o estudo realizado e uma consciência das dificuldades encontradas no processo de desenvolvimento.

Este trabalho mostrou a importância que a atividade de teste tem para verificar o correto funcionamento de programas. A Engenharia de *Software* propõe que seja feito um planejamento antes do desenvolvimento dos testes. O planejamento ajuda a entender as abordagens para implementação de testes existentes e os critérios utilizados por cada uma. Esse conhecimento permite uma decisão consciente de qual metodologia seguir para atingir o objetivo principal do trabalho, com base nos problemas encontrados.

Outra lição aprendida foi o conceito de Teste Funcional. Essa abordagem de teste envolve diferentes critérios, com vantagens e desvantagens específicas que são apropriadas para diversos tipos de programas. Um ponto importante para a abordagem de Teste Funcional é a necessidade de uma boa documentação do programa, para conseguir testes concisos e corretos.

As dificuldades encontradas no decorrer deste trabalho consistem, principalmente, no entendimento da implementação e funcionamento do GODA. Este *framework* contém um código complexo que utiliza um modelo de negócios do TAOM4E para o tratamento do modelo CRGM, análise de seus elementos e escrita dos novos modelos PRISM e PARAM. Para o desenvolvimento dos testes automatizados, os modelos CRGM foram construídos programaticamente, logo não foi possível obter as infomações dos modelos de negócio enviados pelo TAOM4E. Por este motivo, houve dificuldade para gerar os modelos seguindo o formato utilizado pelo GODA e, também, para a utilização do código do GODA que analisa os modelos CRGM. A análise dos modelos nos testes teve que ser readaptada.

Apesar das dificuldades ao longo do processo de desenvolvimento, o resultado final mostra que o aprendizado foi significante para a conclusão com êxito do trabalho proposto.

Capítulo 5

Conclusão

O objetivo principal deste trabalho foi desenvolver uma suíte de testes para analisar anotações dos elementos dos modelos CRGM utilizados no *framework* GODA. Para cumprir com essa proposta, foi utilizado teste do tipo caixa preta, ou seja, uma abordagem de teste funcional. Dentre os critérios do teste funcional, o mais apropriado para a análise de modelos foi o de Teste Funcional Sistemático. Logo, foram definidas classes de equivalência a partir da especificação do GODA, e, em seguida, foram definidos os casos de teste. Cada classe de equivalência possui, no mínimo, dois casos de teste.

Os testes foram implementados utilizando a linguagem de programação Java, e foram automatizados utilizando a ferramenta JUnit. Os resultados foram satisfátorios e importantes. Os testes implementados mostram falhas de validação que ocorrem no processo de transformação de modelos CRGM em DTMC, no GODA. Eles conseguem especificar os casos em que a verificação não ocorre, ajudando os desenvolvedores do *framework* a encontrar as falhas.

O *framework* GODA foi desenvolvido unindo ferramentas diversas - como PRISM, PARAM, TAOM4E - o que aumenta a probabilidade de ocorrência de erros. Logo, a implementação de testes funcionais consegue verificar a corretude do funcionamento do programa para os tipos e classes mais relevantes do GODA.

Este trabalho analisa os elementos do CRGM durante a transformação e modelos que ocorre no GODA. Para trabalhos futuros, é importante o planejamento e desenvolvimento de outros tipos de teste, com abordagens diferentes, para garantir que o *framework* contenha uma estrutura interna correta, e que a transformação de modelos resulte num modelo DTMC válido e preciso.

Referências

[1] R. Ali, F. Dalpiaz, and P. Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15:439–458, 2010-11. 1, 5, 7

[2] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy od dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. 1

[3] C. Baier and P. J. Katoen. Principles of model checking (representation and mind series). *The MIT Press*, 2008. 5

[4] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, volume 1. Addison Wesley Longman, 1999. 12

[5] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, pages 203–236, 2004. 4

[6] L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, 2004. 13, 14

[7] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos. Runtime goal models: Keynote. *Research Challenges in Information Science (RCIS)*, pages 1–11, 2013. viii, 7, 8

[8] A. Dardenne and A. van Lamsweerde. Goal-directed requirements acquisition. *Science of Computer Programming*, pages 3–50, 1993. 4

[9] M. E. Delamaro, J. C. Maldonado, and M. Jino. *Introdução ao Teste de Software*. Elsevier Editora Ltda, 2007. 12, 13, 14, 18, 19

[10] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *Proceedings of RE 1995*, page 140, Washington, DC, USA, 1995. IEEE Computer Society. 1

[11] C. Ghezzi and A. M. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Inf. Softw. Technol.*, 55(3):508–524, 2013. 4, 6

[12] E. M. Hahn, H. Hermmans, B. Watcher, and L. Zhang. Param: A model checker for parametric markov models. *CAV*, pages 660–664, 2010. 6

[13] P. C. Jorgensen and C. Erickson. Object oriented integration testing. *Communications of the ACM*, 37(9):30–38, 1994. 12

[14] E. Kit. *Software testing in the real world: improving the process*. Addison-Wesley, 1995. 13

[15] M. Kwiatkowska and D. Parker. Advances in probabilistic model checking. *IOS Press*, 2012. 5, 6

[16] O. U. C. Laboratory. Prism web site. http://www.prismmodelchecker.org/. 6

[17] S. Linkman, A. M. R. Vincenzi, and J. Maldonado. An evaluation of systematic functional testing using mutation testing. In *7th International Conference on Empirical Assessment in Software Engineering - EASE*, Keele, UK, 2003. IEEE. 13, 14, 22

[18] D. F. Mendonca. Dependability verification for contextual/runtime goal modelling. Master's thesis, Universidade de Brasilia, 2015. vi, 5, 6, 9, 10, 11, 19

[19] D. F. Mendonca, G. N. Rodrigues, R. Ali, V. Alves, and L. Baresi. Goda: A goal-oriented requirements engineering framework for runtime dependability analysis. Preprint submitted to Information of Software Technology, 2016. vi, 1, 4, 5, 6, 7, 9

[20] M. Morandini, D. C. Nguyen, A. Perini, A. Siena, and A. Susi. Tool-supported development with tropos: The conference management system case study. *Proceedings of the 8th International Conference on Agent-oriented Software Engineering VIII*, 2008. 2, 8

[21] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing*. John Wiley & Sons, 2 edition, 2004. 13, 14

[22] J. Mylopoulos, L. Chung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Commun. ACM*, 42(1):31–37, jan 1999. 1

[23] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: A combined approach to self-management. In *Proceeding of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 1–8, New York, NY, USA, 2008. ACM. 1

[24] E. S. K. Yu. *Modelling strategic relationships for process reengineering*. PhD thesis, University of Toronto, 1996. 4