



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

**Desenvolvimento de um Analisador de Corretude do
Modelo Orientado a Objetivos do GODA.**

Gabriel Nunes Ribeiro Silva

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

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2019



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Desenvolvimento de um Analisador de Corretude do Modelo Orientado a Objetivos do GODA.

Gabriel Nunes Ribeiro Silva

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. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 12 de julho de 2019

Dedicatória

Dedico este trabalho a Deus porque Dele, e por Ele, e para Ele são todas as coisas.

Agradecimentos

Agradeço primeiramente a Deus, pela força, coragem e capacitação que Ele me concedeu durante a minha vida, e em particular durante este árduo processo de formação.

Agradeço à minha orientadora, Prof.a Dr.a Genaína Nunes Rodrigues, que esteve à disposição, me orientou e me guiou durante esse processo com empolgação e bom ânimo. À Gabriela Félix Solano, por ter se disponibilizado e oferecido para me prover uma ajuda que foi imprescindível para a realização deste trabalho.

Agradeço aos meus pais, Régia e Carlos, que sempre me apoiaram em todos momentos e me incentivaram para que eu pudesse alcançar os meus objetivos. Ao meu irmão, Vinícius, pelo exemplo de excelência acadêmica e profissional que sempre me passou. Aos meus avós, Maria, Cláudio e Nilda, que foram essenciais para o meu crescimento pessoal e minha educação.

Agradeço à minha namorada Ana Beatriz, por ter sido fundamental para que eu tivesse forças para concluir essa longa jornada, por me acompanhar durante toda essa caminhada me ajudando e me motivando para alcançar os meus objetivos, estando sempre presente para dividir as alegrias e frustrações da vida acadêmica.

Agradeço aos meus tios e tias, que ao longo desses anos contribuiram para o meu crescimento pessoal e me deram apoio durante esta jornada. Ao meu primo, Prof. MSc. Daniel Saad, por toda ajuda que pacientemente me ofereceu ao longo dessa caminhada, que foi fundamental para o meu desenvolvimento acadêmico.

Agradeço, por fim, ao Departamento de Ciência da Computação e aos docentes com quem tive o prazer de aprender durante esta jornada.

Resumo

O framework GODA (Goal-Oriented Dependability Analysis) realiza análises de dependabilidade em modelos orientados a objetivos. Para a análise probabilística de dependabilidade, o GODA transforma um modelo CRGM (Contextual and Runtime Model) em um modelo de verificação DTMC (Discrete-Time Markov Chains). Em seguida, propriedades de dependabilidade são renderizadas em fórmulas e enfim a verificação é iniciada. No entanto, para que a corretude da análise de dependabilidade seja assegurada, é importante certificar que o modelo CRGM gerado esteja correto, isto é, que o modelo cumpre todas as regras de runtime pré-determinadas para não propagar erros para a análise propriamente dita. A atual versão do GODA já foi testada em trabalhos que antecedem a realização deste, e foi verificada uma precariedade nas verificações do modelo CRGM, constatando que em diversos casos estados inválidos não estão sendo reconhecidos como tal. O problema disso é que acaba comprometendo os resultados gerados e acarretando em uma inconsistência do framework no desempenho de corretude de sua funcionalidade. Assim, esse trabalho almejou prover suporte a essa análise de corretude e consistência ao GODA através da implementação de um analisador de tipos, responsável por fazer a identificação dos estados válidos e inválidos do modelo CRGM, impedindo a propagação de erros para o modelo DTMC.

Palavras-chave: GODA, modelo orientado a objetivos, corretude de modelo orientado a objetivos, dependabilidade

Abstract

GODA framework performs dependability analysis on goal-oriented models. In order to do that, GODA automatically transforms a CRGM (Contextual and Runtime Goal Model) into a DTMC (Discrete-Time Markov Chains). After that, dependability properties are rendered as formulas and the verification is executed. However, in order to ensure the reliability of the process, it is important to make sure that the generated CRGM model is correct, that is, that the model fulfills all predetermined runtime rules so it won't propagate errors into the analysis itself. The most recent version of GODA has already been tested in previous works, and it has been confirmed a precariousness regarding CRGM correctness verifications. This implies that invalid states are not being recognized as such, compromising all results generated by the framework, causing an inconsistency in its analysis. Thus, this work aimed to provide this reliability and consistency to GODA through the implementation of a correctness checker, responsible for identifying the valid and invalid states of the CRGM model and preventing the propagation of errors to DTMC model.

Keywords: GODA, goal models, correctness of a goal model, dependability

Sumário

1	Introdução	1
1.1	Problema	2
1.2	Objetivos	3
1.2.1	Objetivo Geral	3
1.2.2	Objetivos Específicos	3
1.2.3	Contribuições Esperadas	3
1.2.4	Organização do Trabalho	4
2	Referencial Teórico	5
2.1	Engenharia de Requisitos Orientada a Objetivos (GORE)	5
2.2	TROPOS	5
2.2.1	Conceitos Básicos	6
2.2.2	Relações	6
2.2.3	Modelo	7
2.3	Modelo Contextual Orientado a Objetivos (CGM)	8
2.4	Modelo Orientado a Objetivos em Tempo de Execução (RGM)	8
2.5	Cadeias de Markov em Tempo Discreto (DTMC)	9
2.5.1	Processo Estocástico	10
2.5.2	Propriedade de Markov	10
2.5.3	Definição Formal	11
2.6	Verificação Probabilística de Modelos (PMC)	11
2.6.1	Linguagem PRISM	13
2.6.2	Lógica Probabilística de Árvore Computacional (PCTL)	13
2.7	Análise de Dependabilidade Orientada a Objetivos (GODA)	14
2.7.1	Modelo Contextual Orientado a Objetivos em Tempo de Execução (CRGM)	15
2.8	ANTLR	17

3 Proposta	19
3.1 Regras de Corretude	19
3.2 Planejamento da Solução	21
3.3 Casos Contemplados	22
3.4 Modelo de Implementação	33
3.4.1 Mudança no Front-end	37
4 Resultados	41
4.1 Discussão	51
4.2 Histórico de Resultados do GODA	52
4.3 Lições Aprendidas e Dificuldades Encontradas	53
5 Conclusão	54
Referências	56
Anexo	57
I Testes Funcionais	58
II Testes Adicionais	72

Listas de Figuras

2.1	Exemplo de Modelo Orientado a Objetivo baseado em [1]	7
2.2	Exemplo de um DTMC [2].	12
2.3	Exemplo de sintaxe PRISM DTMC [3].	13
2.4	Processo do GODA [3].	14
2.5	CRGM referente ao modelo de objetivos da Figura 2.1	16
3.1	Exemplo dos conceitos de declaração e definição.	22
3.2	Ilustração para o caso 1a.	23
3.3	Ilustração para o caso 1b.	23
3.4	Ilustração para o caso 1c.	24
3.5	Ilustração para o caso 1d.	24
3.6	Ilustração para o caso 1e.	24
3.7	Ilustração para o caso 1f.	25
3.8	Ilustração para o caso 1g.	25
3.9	Ilustração para o caso 1h.	26
3.10	Ilustração para o caso 1i.	26
3.11	Ilustração para o caso 1j.	26
3.12	Ilustração para o caso 1k.	27
3.13	Ilustração para o caso 2a.	27
3.14	Ilustração para o caso 2b.	28
3.15	Ilustração para o caso 2c: chave foi utilizada no lugar do colchete em T1. .	28
3.16	Ilustração para o caso 2d.	29
3.17	Ilustração para o caso 2e.	29
3.18	Ilustração para o caso 2f.	30
3.19	Ilustração para o caso 2g.	30
3.20	Ilustração para o caso 2h.	31
3.21	Ilustração para o caso 2i.	31
3.22	Ilustração para o caso 2j.	32
3.23	Ilustração para o caso 2k.	32
3.24	Ilustração para o caso 2l.	33

3.25 Parte do menu do piStarGODA [4].	37
3.26 A label "G10: Line Locations Tracked" está inserida em uma Tarefa [3].	38
3.27 Ilustrando diferenciação por cores.	38
3.28 Mensagem de erro fixa.	39
3.29 Mensagem de erro informativa. [3].	39
4.1 Resultado do caso de teste 1.	41
4.2 Errata: erro de digitação na tarefa T1.1 na realidade tornaria o modelo incorreto [5].	43
4.3 Resultado do caso de teste 2.	44
4.4 Resultado do caso de teste 3.	44
4.5 Resultado do caso de teste 4.	44
4.6 Resultado do caso de teste 5.	44
4.7 Resultado do caso de teste 6.	44
4.8 Resultado do caso de teste 7.	44
4.9 Resultado do caso de teste 8.	44
4.10 Resultado do caso de teste 9.	45
4.11 Resultado do caso de teste 10.	45
4.12 Resultado do caso de teste 11.	45
4.13 Resultado do caso de teste 12.	45
4.14 Resultado do caso de teste 13.	45
4.15 Resultado do caso de teste 14.	45
4.16 Resultado do caso de teste 15.	45
4.17 Resultado do caso de teste 16.	46
4.18 Resultado do caso de teste 17.	46
4.19 Resultado do caso de teste 18.	46
4.20 Resultado do caso de teste 19.	46
4.21 Resultado do caso de teste 20.	46
4.22 Resultado do caso de teste 21.	46
4.23 Resultado do caso de teste 22.	46
4.24 Resultado do caso de teste 23.	47
4.25 Resultado do caso de teste 24.	47
4.26 Resultado do caso de teste 25.	47
4.27 Resultado do caso de teste 26.	47
4.28 Resultado do caso de teste 27.	47
4.29 Resultado do caso de teste 28.	47
4.30 Resultado do caso de teste 29.	47
4.31 Resultado do caso de teste 30.	48

4.32	Resultado do caso de teste 31.	48
4.33	Resultado do caso de teste 32.	48
4.34	Resultado do caso de teste 33.	48
4.35	Resultado do caso de teste 34.	48
4.36	Resultado do caso de teste 35.	48
4.37	Sumário dos resultados dos testes.	49
4.38	Resultados dos casos de teste adicionais via JUnit.	50
I.1	Ilustração para o caso de teste 1.	58
I.2	Ilustração para o caso de teste 2.	59
I.3	Ilustração para o caso de teste 3.	59
I.4	Ilustração para o caso de teste 4.	59
I.5	Ilustração para o caso de teste 5.	60
I.6	Ilustração para o caso de teste 6.	60
I.7	Ilustração para o caso de teste 7.	60
I.8	Ilustração para o caso de teste 8.	61
I.9	Ilustração para o caso de teste 9.	61
I.10	Ilustração para o caso de teste 10.	61
I.11	Ilustração para o caso de teste 11.	62
I.12	Ilustração para o caso de teste 12.	62
I.13	Ilustração para o caso de teste 13.	62
I.14	Ilustração para o caso de teste 14.	63
I.15	Ilustração para o caso de teste 15.	63
I.16	Ilustração para o caso de teste 16.	64
I.17	Ilustração para o caso de teste 17.	64
I.18	Ilustração para o caso de teste 18.	64
I.19	Ilustração para o caso de teste 19.	65
I.20	Ilustração para o caso de teste 20.	65
I.21	Ilustração para o caso de teste 21.	66
I.22	Ilustração para o caso de teste 22.	66
I.23	Ilustração para o caso de teste 23.	66
I.24	Ilustração para o caso de teste 24.	67
I.25	Ilustração para o caso de teste 25.	67
I.26	Ilustração para o caso de teste 26.	67
I.27	Ilustração para o caso de teste 27.	68
I.28	Ilustração para o caso de teste 28.	68
I.29	Ilustração para o caso de teste 29.	68
I.30	Ilustração para o caso de teste 30.	69

I.31 Ilustração para o caso de teste 31.	69
I.32 Ilustração para o caso de teste 32.	69
I.33 Ilustração para o caso de teste 33.	70
I.34 Ilustração para o caso de teste 34.	70
I.35 Ilustração para o caso de teste 35.	71
II.1 Ilustração para o caso de teste da regra de corretude 1b.	72
II.2 Ilustração para o caso de teste da regra de corretude 1c.	73
II.3 Ilustração para o caso de teste da regra de corretude 1e.	73
II.4 Ilustração para o caso de teste da regra de corretude 1g.	73
II.5 Ilustração para o caso de teste da regra de corretude 1j.	74
II.6 Ilustração para o caso de teste da regra de corretude 1k.	74
II.7 Ilustração para o caso de teste da regra de corretude 2e.	75
II.8 Ilustração para o caso de teste da regra de corretude 2g.	75

Lista de Tabelas

2.1 Sintaxe de expressões em um modelo orientado a objetivos, onde E é uma fórmula que representa um objetivo [6].	9
2.2 Regras RGM aplicadas no GODA, onde n, n1, n2 representam objetivos ou tarefas [3].	16
4.1 Classes de equivalência e respectivos casos de teste [4],[5]	42
4.2 Resultado dos testes.	43
4.3 Resultados dos casos de teste adicionais.	50
4.4 Versões do GODA e seus desempenhos nos testes funcionais.	52

Capítulo 1

Introdução

Requisitos de um sistema de software podem ser analisados por uma metodologia orientada a objetivos que visa explicar o quê, o porquê e o como de um sistema [7]. Fazendo uso dessa metodologia nasce o GORE (*Goal-Oriented Requirements Engineering*), que está diretamente associado à análise da utilização de objetivos como forma de elaborar, validar, modificar requisitos e com base nisso avaliar a forma ideal de desenvolvimento de um sistema de software [8].

Um modelo orientado a objetivos tende a ser descrito de forma hierárquica e o desafio passa a ser desenvolver maneiras distintas de satisfazê-los. No entanto, o contexto do sistema de software influencia diretamente os stakeholders do sistema, uma vez que promove mudanças na viabilidade e performance de cada solução que satisfaça um dado objetivo. Assim, o desempenho de um sistema pode ser medido pela sua capacidade de se adaptar a mudanças de contexto [9].

Além da adaptabilidade, outra característica importante para um sistema de software é a dependabilidade, que diz respeito à habilidade de fornecer um serviço cuja confiança pode ser justificada [10]. Nesse escopo, integrando adaptabilidade, dependabilidade e GORE, surge o framework GODA (*Goal-Oriented Dependability Analysis*), que propõe um meio de argumentar sobre a dependabilidade de um sistema em contextos variáveis [3]. Para isso, GODA agrupa um modelo de verificação probabilística com os conceitos do RGM (*Runtime Goal Model*) e do CGM (*Contextual Goal Model*), permitindo a avaliação das probabilidades de satisfação de diferentes objetivos, instigando uma correlação direta entre o modelo de objetivos do sistema e a verificação de dependabilidade [7].

Para a análise probabilística, o GODA transforma o modelo CRGM (*Context and Runtime Goal Model*) em um modelo de verificação DTMC (*Discrete-Time Markov Chains*) a partir da linguagem PRISM. No entanto, para que a corretude da análise de dependabilidade seja assegurada, é importante certificar que o modelo CRGM gerado esteja correto. Isto é, é necessário verificar se o CRGM gerado satisfez todas as regras de tempo

de execução estabelecidas pelo RGM, além de verificar se as notações pré-determinadas foram utilizadas corretamente.

A última versão do GODA se vale apenas da utilização de uma gramática e do ANTLR (*Another Tool for Language Recognition*) para prover essa verificação. No entanto, como será discutido no decorrer deste trabalho, apenas essa ferramenta não é suficiente para que a boa formação do modelo CRGM seja assegurada, pois não são realizadas verificações acerca da modelagem, no que diz respeito à coerência entre as declarações de objetivos e tarefas e suas posteriores definições.

As declarações ocorrem quando um nó define seu refinamento em outros subobjetivos e/ou subtarefas, declarando estes com seus nomes e identificadores em sua anotação de tempo de execução. A definição, por sua vez, é quando esse nó, que representa o refinamento do objetivo/tarefa pai, é de fato criado e inserido no modelo de objetivos. Portanto, a verificação de coerência é pautada na correspondência, ou não, do nome do objetivo/tarefa no momento de definição, e do nome que foi previamente declarado na anotação de tempo de execução de seu nó superior.

Dessa forma, a motivação deste trabalho é promover essa consistência ao GODA, estabelecendo meios de garantir a boa formação do modelo no que diz respeito ao cumprimento das regras de tempo de execução e de notação, complementando o ANTLR e assegurando maior confiabilidade à transformação do CRGM em DTMC.

As definições acerca do GODA serão mais profundamente abordadas no próximo capítulo.

1.1 Problema

O problema observado é que não existe uma ferramenta de verificação implementada para garantir que o modelo CRGM instanciado esteja bem formado. Isto é, que esteja respeitando os estados válidos e corretamente identificando os inválidos, conforme as regras de tempo de execução do *framework* GODA.

Assim, mesmo que as *labels* dos elementos modelados estejam bem formadas em relação à gramática (verificado pelo ANTLR, detalhado na Seção 2.6), não existem verificações quanto à corretude do conteúdo dela no contexto da modelagem sendo realizada, podendo ocorrer declarações incorretas das variáveis (que representam fórmulas instanciadas ou elementos do modelo de objetivos).

Características de nomenclatura e de declarações que definem estados válidos e inválidos, bem como resultados dos casos de teste que a atual versão do GODA foi submetido foram devidamente descritos por Solano *et al.* (2016) e Bergmann *et al.* (2018).

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo central deste trabalho é o desenvolvimento de um analisador de corretude do modelo orientado a objetivos do framework GODA, visando assegurar a boa formação do modelo CRGM, no que diz respeito à coerência entre as declarações das variáveis do modelo e suas posteriores definições. Além disso, objetiva o cumprimento de regras de tempo de execução não verificadas pelo ANTLR (Seção 2.9).

1.2.2 Objetivos Específicos

O analisador de corretude do modelo orientado a objetivos do GODA proposto neste trabalho visa analisar a boa formação e as declarações de variáveis instanciadas no CRGM para aumentar a confiança dos processos de transformação que ocorrem a partir do CRGM. Para isso, foi mapeado, com o auxílio do estudo de Solano *et al.* (2016), diversas questões que infringem as regras do sistema para serem endereçadas na realização deste, elas são devidamente descritas na Seção 3.1.

Alguns testes sobre essas condições foram realizados sobre a primeira versão do GODA e relatados no estudo de Solano *et al.*, e reaplicados no trabalho de Bergmann *et al.*, e ambas revelam a necessidade de uma ferramenta como a proposta neste projeto.

Para alcançar êxito no desenvolvimento deste trabalho, foram traçadas algumas atividades a serem contempladas visando a boa organização do processo de implementação. A primeira é a coleta de atuais discrepâncias do sistema em relação à identificação de estados inválidos, com auxílio dos trabalhos anteriores [5], [4]. Após a coleta, filtrar os erros atribuídos à falta de um analisador de corretude do modelo orientado a objetivos. Posteriormente, tratar os casos levantados através da implementação de métodos na linguagem Java, seguido de experimentação com testes funcionais desenvolvidos por Solano *et al.* e, por fim, o relato dos resultados obtidos.

1.2.3 Contribuições Esperadas

A contribuição efetiva deste trabalho está no acréscimo de um analisador de corretude do modelo orientado a objetivos do framework GODA, proporcionando uma garantia maior em relação à corretude do modelo CRGM. Isso promove um aumento na confiança do processo de transformação do CRGM em modelo DTMC (abordado no capítulo seguinte), que por sua vez irá resultar em uma maior corretude no processo de renderização de fórmulas para a análise de dependabilidade. Assim, a realização deste trabalho irá viabilizar

uma maior utilização do GODA, uma vez que os principais e mais frequentes focos de erros propagados para a análise de dependabilidade serão tratados.

1.2.4 Organização do Trabalho

Este trabalho foi estruturado em cinco (5) capítulos. O Capítulo 2 apresenta o Referencial Teórico, que define os conceitos que fazem parte do escopo deste trabalho. O Capítulo 3 aborda a proposta deste trabalho para o problema exposto, e descreve o embasamento e a implementação da solução. O Capítulo 4, por sua vez, foi organizado de forma a apresentar os resultados obtidos através da realização deste trabalho. O capítulo 5, por fim, discorre sobre as conclusões inferidas.

Capítulo 2

Referencial Teórico

2.1 Engenharia de Requisitos Orientada a Objetivos (GORE)

A engenharia de requisitos orientada a objetivos nasceu visando uma exploração do conceito de objetivo e da forma com que é possível modelar sistemas de software em cima dessa definição. Assim, GORE propõe uma forma de utilizar modelos orientados a objetivos para elaborar, estruturar, documentar e levantar requisitos de forma a argumentar o porquê de um sistema de software, justificando cada requisito do *system-to-be* [8]. Outro motivo da popularização de modelos de objetivos é que objetivos também podem ser utilizados para identificar tipos distintos de funcionalidades: as funcionais (atreladas às funcionalidades que o sistema tem de exercer), e as não-funcionais (relacionadas à qualidade, segurança, performance, precisão, entre outras). A Figura 2.1 exemplifica um modelo orientado a objetivo.

Dessa forma, dentro desse escopo de modelagem orientada a objetivos, existem algumas metodologias *agent-oriented* que foram definidas de forma a guiar o processo de desenvolvimento de software orientado a objetivos, uma delas é o TROPOS [11], que será abordado na Seção a seguir.

2.2 TROPOS

Tropos é uma metodologia de desenvolvimento de software orientada a agentes que contempla as mais diversas etapas de desenvolvimento, desde o processo de coleta inicial de requisitos até a fase de implementação. Para definir suas diretrizes, TROPOS adota a linguagem i* - linguagem para modelagem com enfoque na abordagem de questões acerca do *quê*, *quem* e *como*, bastante utilizada no escopo didático de orientação a objetivos -

e todos os conceitos vigentes nela e estende essas definições para contemplar diferentes fases do desenvolvimento [11].

2.2.1 Conceitos Básicos

Os conceitos básicos do TROPOS são, de acordo com [1]:

- Atores: são entidades ativas que possuem objetivos e decidem autonomamente a forma de satisfazê-los. Atores podem ser humanos, organizações, tecnologias.
- Objetivos: aquilo que os atores pretendem alcançar através de determinada ação. Nesse tópico existe a distinção entre *Hardgoal* e *Softgoal*: a primeira denominação é utilizada para fazer referência a objetivos que possuem indicadores exatos para medir se ele foi satisfeito ou não, enquanto o segundo diz respeito aos objetivos que não se pode medir a satisfação de uma forma pré-estabelecida, sendo um pouco mais subjetivo [12].
- Tarefas: é o meio pelo qual é possível satisfazer um dado objetivo, isto é, uma ação ou um conjunto de ações que levam ao alcanceamento dele.
- Recursos: uma informação ou um recurso físico que pode ser fornecido ou solicitado por atores.

2.2.2 Relações

Ainda de acordo com [1], algumas relações fundamentais em abordagens orientadas à objetivos são:

- Refinamento *OR*: refina um objetivo (ou uma tarefa) em subobjetivos (ou subtarefas) de forma que, para satisfazer o *goal* inicial, ao menos um dos subobjetivos (ou subtarefas) criados devem ser realizados/contemplados.
- Refinamento *AND*: refina um objetivo (ou uma tarefa) em subobjetivos (ou subtarefas) de forma que, para satisfazer o *goal* inicial, todos os subobjetivos (ou subtarefas) criados devem ser realizados/contemplados.

Portanto, o TROPOS, assim como o KAOS (*Knowledge Agent-Oriented System*) - outra metodologia orientada a agentes - fornece um meio de levantar os requisitos dos *stakeholders* e, através das relações, é capaz de prover combinações de tarefas que satisfazem um determinado objetivo.

2.2.3 Modelo

A seguir é exemplificado um modelo orientado a objetivos que ilustra os conceitos básicos e as relações do TROPOS explicitadas anteriormente. Observe que as formas (além das cores, neste exemplo) diferem os elementos: o ator está representado por um círculo, os objetivos pelo formato oval, e as tarefas pelo formato em polígono.

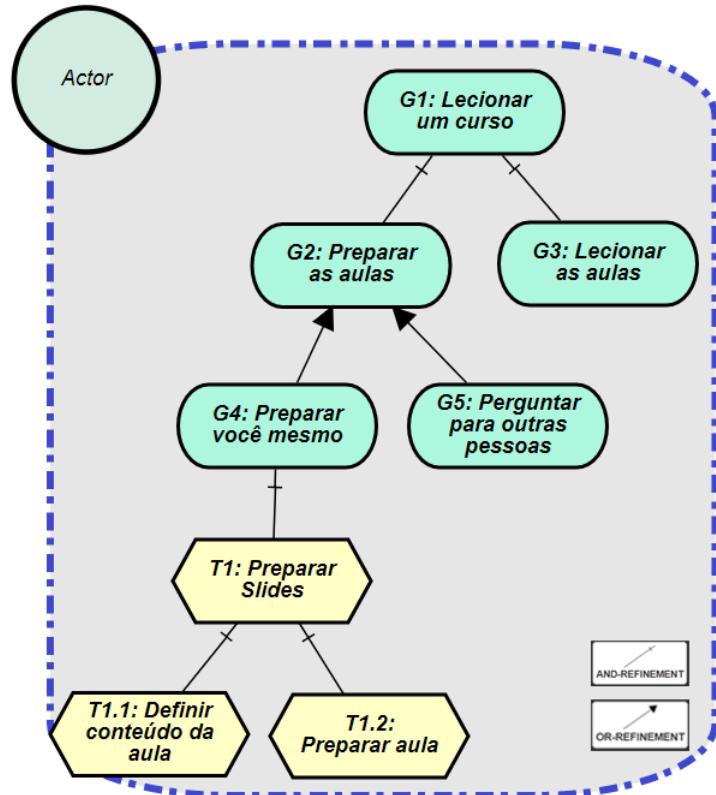


Figura 2.1: Exemplo de Modelo Orientado a Objetivo baseado em [1] .

A leitura do modelo é bastante simples: o objetivo principal é "Lecionar um Curso", e ele é satisfeito através da realização de ambos objetivos (refinamento *AND*) G2 e G3 ("Preparar as aulas" e "Lecionar as Aulas", respectivamente). No entanto, para que o objetivo de "Preparar as aulas" seja satisfeito, deve-se realizar um subobjetivo (refinamento *OR*): G4 ou G5 ("Preparar você mesmo" ou "Perguntar para outras pessoas", respectivamente). Da mesma forma, para o objetivo G4 ser alcançado, a tarefa T1 (Preparar Slides) deve ser contemplada. Porém, para isso, devem ser satisfeitas ambas subtarefas T1.1 e T1.2 ("Definir conteúdo da aula" e "Preparar aula").

A semântica da modelagem acima exemplifica como os preceitos do GORE (abordado na Seção 2.1) são colocados na prática, e como é natural utilizar o conceito de objetivo para arquitetar, estruturar e levantar requisitos de um sistemas de software.

2.3 Modelo Contextual Orientado a Objetivos (CGM)

O contexto de um sistema de software pode ser visto como o conjunto de fatores que compõe o ambiente de execução em que um sistema opera. Ele é um fator determinante para estabelecer o conjunto de requisitos de um sistema e as possíveis formas para satisfazê-los. Além disso, também é utilizado para viabilizar a escolha da melhor solução para cada objetivo considerando cada cenário específico desse sistema de software [9].

No entanto, apesar das vantagens, muitos processos de engenharia de requisitos ainda falham em se preocupar com o contexto de um sistema de software, resultando em baixa flexibilidade e baixo desempenho desses sistemas em particular.

O que deve ser compreendido é que o conjunto de fatores que compõe o ambiente de execução em que um sistema opera influencia diretamente os *stakeholders* de um sistema de software, uma vez que ações distintas que satisfazem um mesmo objetivo podem ser viáveis ou não, a depender do contexto de execução naquele momento. Assim, o contexto acaba sendo uma variável, e os sistemas devem ser planejados para se auto-adaptarem a essas oscilações [9].

No escopo do GODA, o CGM é aplicado de forma que cada contexto possua um identificador único, uma descrição e uma expressão booleana que será verificada na análise de dependabilidade [7]. Dessa forma, dependendo da satisfação ou não da expressão que representa cada contexto, é possível adotar uma solução alternativa mais viável naquele momento.

2.4 Modelo Orientado a Objetivos em Tempo de Execução (RGM)

O RGM propõe uma maneira de analisar a forma com que o sistema se comporta durante sua execução, verificando o cumprimento (ou não) dos requisitos estabelecidos pela RE (Requirements Engineering), provendo capacidade de monitoramento e diagnóstico para o sistema [6]. Mais especificamente, sua estruturação identifica: a quantidade de instâncias de um objetivo em algum momento específico, o comportamento de cada instância e o estado em que cada uma se encontra (sucesso, em andamento).

Para isso, o RGM propõe uma série de regras de sintaxe e notação para as expressões, como evidenciado na Tabela 2.1. A proposição do RGM adveio da alegação de Dalpiaz *et al.* [6] que modelos orientados a objetivos estavam sendo utilizados - errôneamente, para eles - para diagnóstico de tempo de execução de sistemas, quando na realidade o intuito dessa metodologia é de modelagem/design de sistemas de software [6]. Assim, RGM foi

proposto para criar um meio de substituir a utilização de um modelo inadequado para os fins explicitados anteriormente.

Expressão	Significado
skip	Sem ação.
$E_1 ; E_2$	Ocorrência sequencial.
$E_1 E_2$	Alternativa (escolha exclusiva).
opt(E)	E é opcional.
E^+	Uma ou mais ocorrência sequencial de E .
G^{succ}	Uma instância de G começa e termina em um estado de sucesso.
G^{fail}	Uma instância de G começa e termina em um estado de falha.
try(G)? $E_1 : E_2$	Se uma instância de G obtém sucesso, E_1 ; senão, E_2 .
$E_1 \# E_2$	Ocorrência intercalada de E_1 e E_2 .
$E^{\#}$	Uma ou mais instâncias de E ocorrendo concorrentemente.

Tabela 2.1: Sintaxe de expressões em um modelo orientado a objetivos, onde E é uma fórmula que representa um objetivo [6].

Suponha que um objetivo representado pela fórmula E é decomposto em E_1 e E_2 . No RGM, foram definidos dois sistemas fundamentais de comportamento: sequencial, expresso por ';;', e intercalado, expresso por '#'. Portanto, a anotação $(E_1;E_2)$ expressa que o objetivo representado por E_1 deve ser satisfeito antes do E_2 , enquanto a anotação $(E_1\#E_2)$ não impõe nenhuma ordem de satisfação dos objetivos envolvidos, apenas que ambos objetivos executem de forma concorrente.

A regra de alternativa $(E_1|E_2)$ requere a execução exclusiva de E_1 ou E_2 , expressando a capacidade do sistema de exibir comportamentos distintos, enquanto a anotação envolvendo opt(E) indica a opção de satisfazer o objetivo E ou não. Nessas linhas, a anotação *try*, que ilustra uma expressão similar ao *if-then-else*, demanda que o sistema tente atingir ao objetivo G , e dependendo do resultado (êxito ou falha), diferentes comportamentos são indicados [6].

As regras E^+ e $E^{\#}$ pressupõem a possibilidade de objetivos serem decompostos de forma que seja necessário mais de uma instância para os subobjetivos, de forma que E^+ representa a ocorrência de múltiplas instâncias de E de forma sequencial, enquanto $E^{\#}$ representa a ocorrência de múltiplas instâncias de E de forma concorrente.

Já as anotações G^{succ} e G^{fail} representam o sucesso ou falha, respectivamente, de determinada instância de um objetivo G .

2.5 Cadeias de Markov em Tempo Discreto (DTMC)

Informalmente, DTMC pode ser definido como um caso particular de processo estocástico que representa um sistema de transição de estados, estendido com a utilização de probabilidades [2]. Antes de definir formalmente o que é uma Cadeia de Markov em Tempo

Discreto, é necessário discorrer sobre processo estocástico e propriedade de Markov, conforme subseções a seguir.

2.5.1 Processo Estocástico

Um processo estocástico é uma família de variáveis aleatórias para modelar a evolução de um sistema dinâmico de valores com o tempo. De acordo com Clarke e Disney (1979), processos estocásticos são fenômenos que variam em algum grau, de forma imprevisível, à medida que o tempo passa [13].

Mais especificamente, dado um espaço amostral Ω , um processo estocástico é um mapeamento $X : Tx\Omega \rightarrow S$, onde T é o conjunto relativo ao parâmetro tempo (geralmente \mathbb{Z}_+), e S é o espaço de estados de X [14].

Assim, pode-se classificar o processo estocástico em relação ao conjunto T (parâmetro tempo):

- se T é tal que $T = \{0, 1, 2, \dots\}$, então o Processo Estocástico é dito de Tempo Discreto.
- Se T é tal que $T = \{t : 0 \leq t < \infty\}$, então o Processo Estocástico é dito de Tempo Contínuo.

Da mesma maneira, é possível classificar o processo estocástico em relação ao conjunto S (espaço de estados) [15]:

- Se $X(t)$ é definido sobre um conjunto enumerável ou finito, então o Processo Estocástico é dito de Estado Discreto.
- Caso contrário, o Processo Estocástico é dito de Tempo Contínuo.

2.5.2 Propriedade de Markov

Um processo estocástico possui a propriedade de Markov, e, consequentemente, é dito um Processo Markoviano se o estado futuro depende apenas do estado presente. Por conta disso, essa propriedade é chamada de "sem memória", uma vez que estados passados não influenciam o futuro [15]. Essa propriedade foi nomeada em homenagem a Andrey Markov, um matemático russo.

2.5.3 Definição Formal

A definição formal de Cadeia de Markov em Tempo Discreto (DTMC) é descrita a seguir [16].

Definição 1 Cadeia de Markov em Tempo Discreto é um Processo Estocástico de Estado Discreto e de Tempo Discreto em que a probabilidade condicional de qualquer evento futuro independe dos estados passados, mas apenas do estado presente, satisfazendo a seguinte condição:

$$P[X_{n+1} = j | X_n = i_n, \dots, X_0 = i_0] = P[X_{n+1} = j | X_n = i_n] = p_{ij}(n) \quad (2.1)$$

onde $p_{ij}(n)$ é a probabilidade de transição do estado i ao estado j no tempo n .

Assim, conforme a definição, um DTMC satisfaz as seguintes características [2]:

- Há um conjunto discreto de estados que representam as possíveis configurações do sistema sendo modelado;
- Transições entre estados ocorrem em intervalos de tempo discreto;
- Probabilidade de realizar transições entre estados são dados por distribuições de probabilidade discreta.

A Figura 2.2 ilustra um DTMC modelando um protocolo de comunicação [2]. A modelagem é bastante simples, podendo ser entendida da seguinte forma: após o primeiro passo (s_0, s_1) , o processo tenta enviar uma mensagem. Para este processo, existem três cenários possíveis: um deles é dado pela probabilidade de 0.01 de que o canal não esteja pronto, devendo aguardar mais um passo para tentar novamente. O segundo cenário é de que a mensagem é enviada com sucesso e o processo para. A probabilidade desse caso é de 0.98. E, por fim, o último cenário ilustra o caso em que o envio da mensagem falha e o processo é reiniciado (s_2, s_0) . A probabilidade disso acontecer é de 0.01.

2.6 Verificação Probabilística de Modelos (PMC)

Técnicas de verificação formal, e em particular checagem de modelos, oferecem uma abordagem poderosa para estabelecer a corretude de sistemas complexos. A verificação probabilística de modelos é uma generalização dessas técnicas, almejando a verificação de sistemas que possuem natureza estocástica [17].

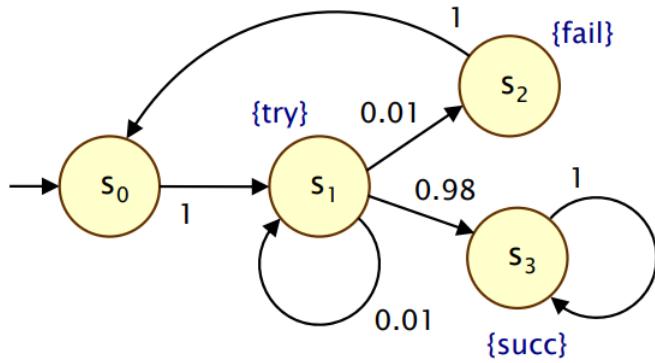


Figura 2.2: Exemplo de um DTMC [2].

Muitos sistemas são suscetíveis a fenômenos de natureza estocástica e desenvolvem um comportamento não determinístico [7]. A verificação probabilística de modelos proporciona um meio para que afirmações quantitativas sejam feitas à respeito da conduta desses sistemas, em termos de probabilidades ou expectativas, complementando o procedimento de checagem de modelo, que por sua vez provê declarações qualitativas acerca desses sistemas [18].

O PMC é baseado em construção e análise de modelos probabilísticos, e entre os mais utilizados estão aqueles baseados em cadeia de Markov, ou processo de Markov, como o DTMC (*Discrete-Time Markov Chain*) e o MDP (*Markov Decision Process*), por exemplo [17]. A técnica PMC possibilita, dessa forma, a previsão do desempenho e da dependabilidade de sistemas baseados em eventos probabilísticos descritos em modelos probabilísticos [7]. Para isso, é necessário [17]:

1. uma descrição do sistema a ser analisado, tipicamente em uma linguagem de modelagem alto nível, como a linguagem PRISM;
2. uma especificação formal das propriedades quantitativas do sistema a serem analisadas, geralmente expressas em variantes de lógica temporal.

A primeira necessidade exposta proporciona ao PMC a construção do modelo probabilístico que corresponde ao sistema a ser analisado. É uma variante probabilística de um sistema de transição de estados, onde cada estado representa uma configuração possível do sistema sendo modelado, e cada transição representa a possibilidade de evolução do sistema de uma configuração para outra ao longo do tempo [17].

O poder do PMC vêm do fato que esses modelos são construídos de forma exaustiva, explorando todos os estados que podem ocorrer. Uma vez construído o modelo, ele pode ser utilizado para analisar uma série de propriedades quantitativas do sistema original, como o desempenho e a dependabilidade [17].

2.6.1 Linguagem PRISM

O framework GODA aplica a técnica PMC e utiliza a linguagem de modelagem alto nível PRISM para descrever o modelo DTMC.

Nessa linguagem, módulos são as principais estruturas no modelo. Eles são compostos de variáveis e comandos. As variáveis descrevem os estados (finitos) em que um módulo pode estar, enquanto comandos expressam o comportamento de um módulo, isto é, as ações que podem resultar em transições de estados. Por fim, as *labels* são utilizadas para nomear comandos e para fins de sincronização [7].

```

const double <probability>; //model constant
module <module_name>
    state : [0..n] init 0; //local module
    variable

    [<action>] <state guard> ->
        <probability>:(<state update>); //command
endmodule

```

Figura 2.3: Exemplo de sintaxe PRISM DTMC [3].

Assim, conforme exemplo acima, um comando DTMC na linguagem PRISM é da seguinte forma [7] (A *label* é opcional):

$$[<action>] <guard> \rightarrow <probability> : <update>; \quad (2.2)$$

A semântica do comando é simples: uma vez que o predicado *guard* é satisfeito, o módulo faz a transição para o estado *update* com uma probabilidade *probability*, onde $0 \leq probability \leq 1$. Caso a probabilidade esteja omitida, é adotado como padrão o valor 1. A presença da *label action* nomeia um comando que deve ser sincronizado com outros comandos, desde que as condições *guard* sejam satisfeitas. Caso não haja *label*, o comando é executado de forma assíncrona [3].

2.6.2 Lógica Probabilística de Árvore Computacional (PCTL)

PCTL (*Probabilistic Computation Tree Logic*) é uma lógica probabilística que pode ser usada para especificar propriedades para modelos de tempo discreto como DTMC e MDP [19]. Essa lógica probabilística deriva da CTL (*Computation Tree Logic*).

A Lógica Probabilística de Árvore Computacional permite verificar a alcançabilidade de uma propriedade φ através da expressão $P =? [F(\varphi)]$, que computa a probabilidade que um sistema eventualmente alcance um estado que satisfaça φ [7].

Assim, a dependabilidade de objetivos no GODA pode ser obtida através da especificação de fórmulas PCTL para verificar propriedades de dependabilidade de um CRGM que foi mapeado em um modelo de verificação DTMC a partir da linguagem PRISM.

2.7 Análise de Dependabilidade Orientada a Objetivos (GODA)

O framework GODA foi proposto visando fornecer um meio de argumentar sobre a dependabilidade de sistemas que operam em contextos dinâmicos.

O processo do GODA é iniciado com o estabelecimento do modelo de objetivos, e, em seguida, é incorporado a ele informações de contexto e informações de tempo de execução, isto é, é integrado ao modelo de objetivos os modelos CGM e RGM, gerando o modelo CRGM (Contextual and Runtime Goal Model). Dessa forma, os objetivos presentes no modelo são cumpridos levando em consideração: especificações contextuais do sistema e cooperação entre os objetivos e tarefas instânciados em tempo de execução, de acordo com as regras RGM aplicadas (que foram estendidas daquelas expostas na Tabela 2.1).

Uma vez completo, é feita uma transformação (automática) do modelo CRGM para o modelo DTMC (*Discrete Time Markov Chain*), e então propriedades de dependabilidade são renderizadas em fórmulas PCTL (*Probabilistic Computation Tree Logic*) [20] para a verificação ser de fato iniciada [3]. O processo de análise de dependabilidade exercido pelo GODA pode ser ilustrado conforme a Figura 2.4.

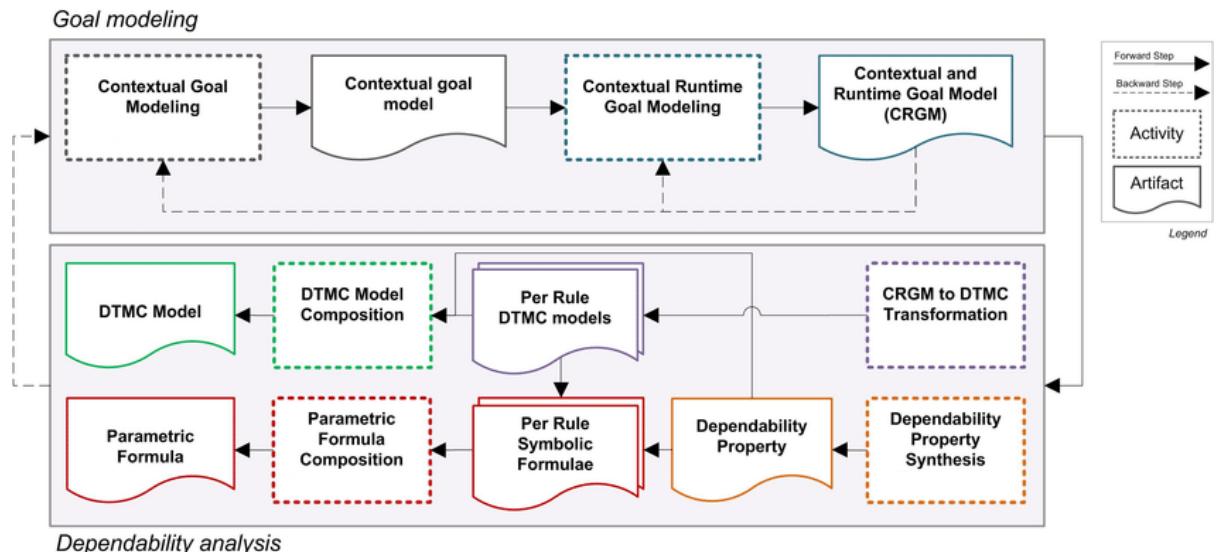


Figura 2.4: Processo do GODA [3].

2.7.1 Modelo Contextual Orientado a Objetivos em Tempo de Execução (CRGM)

O CRGM é a integração entre o Modelo Orientado a Objetivos em Tempo de Execução (RGM) e o Modelo Contextual Orientado a Objetivos (CGM). Essa integração promove a necessidade de considerar especificações de contexto e cooperações entre instâncias de objetivos e/ou tarefas em tempo de execução para a satisfação de cada elemento do modelo de objetivos.

A definição formal do CRGM, de acordo com Mendonça *et al.*, é apresentada a seguir [3].

Definição 2 Um CRGM é uma tupla $(M, rt_annot, ctx_annot, ID)$ onde:

- M é um modelo orientado a objetivos em tempo de design (DGM, por [6], é uma tupla (N, R) , onde N é um conjunto de objetivos e tarefas em um modelo, e R é o conjunto correspondente aos relacionamentos entre os elementos em N).
- rt_annot é uma função de anotação de tempo de execução que retorna uma anotação de tempo de execução associada a um nó $n \in N$ tal que $rt_annot(n)$ pode ser uma única regra de tempo de execução ou uma composição dessas regras.
- ctx_annot é uma função de anotação de contexto que retorna uma fórmula de contexto associada a um nó $n \in N$.
- é uma função índice que mapeia todo $n \in N$ a um identificador $ID(n) = prefix(n) + counter(n)$, onde $prefix(n)$ retorna 'G' ou 'T' para objetivos e tarefas, respectivamente, + é apenas um operador de concatenação de funções e a função $counter(n)$ retorna:
 - um inteiro que identifica únicamente os objetivos em uma ordem ascendente;
 - a constante 1 para tarefas means-end;
 - um vetor de inteiros correspondendo ao ramo e nível da tarefa, incrementado em ordem ascendente.

A Figura 2.5 ilustra um exemplo de um modelo CRGM. Note as diferenças em relação ao modelo de objetivos, exemplificando na Figura 2.1. Apenas no CRGM há preocupações em relação às anotações de tempo de execução, bem como anotações de contexto.

As regras de anotações de tempo de execução utilizados no GODA, ilustrado na Tabela 2.2, devem ser satisfeitas no CRGM modelado.

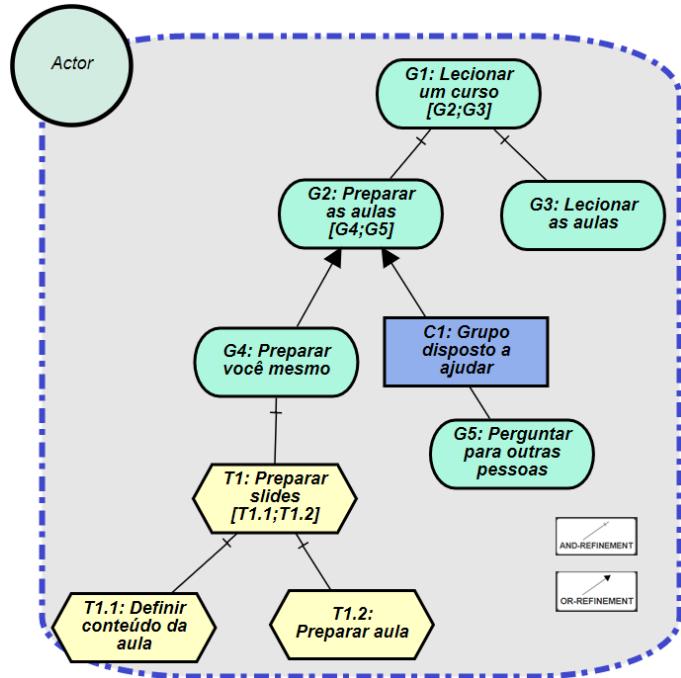


Figura 2.5: CRGM referente ao modelo de objetivos da Figura 2.1 .

As regras RGM utilizadas pelo GODA, bem como suas respectivas descrições e significados, estão expostas a seguir na Tabela 2.2. Essas regras foram estendidas das originalmente expostas por Dalpiaz *et al.*, apresentadas na Tabela 2.1. Note que os refinamentos AND/OR presentes nela remetem às relações previamente descritas na Seção 2.2.2, que explicita as relações fundamentais para abordagens orientadas a objetivo.

Expressão	Significado
AND ($n_1; n_2$)	Cumprimento sequencial de n_1 e n_2 .
AND ($n_1 \# n_2$)	Cumprimento em paralelo de n_1 e n_2 .
OR (n_1/n_2)	Cumprimento sequencial de n_1 ou n_2 ou ambos.
OR ($n_1 \# n_2$)	Cumprimento em paralelo de n_1 ou n_2 ou ambos.
$n+k$	n deve ser realizado k vezes, com $k > 0$.
$n\#k$	Cumprimento paralelo de k instâncias de n , com $k > 0$.
$n@k$	Máximo de $k - 1$ tentativas de cumprimento de n , com $k > 0$.
opt(n)	Cumprimento opcional de n .
try(n)? $n_1:n_2$	Se n foi cumprido, então n_1 deve ser cumprido também; senão, n_2 .
$n_1 n_2$	Cumprimento alternativo de n_1 ou n_2 , mas não ambos.
skip	Sem ação. Útil para expressões condicionais ternárias.

Tabela 2.2: Regras RGM aplicadas no GODA, onde n , n_1 , n_2 representam objetivos ou tarefas [3].

Suponha que n , n_1 , n_2 sejam objetivos e que n é decomposto em n_1 e n_2 .

A regra de $AND(n_1; n_2)$ difere de $AND(n_1 \# n_2)$ apenas em relação à ordem de execução: no primeiro caso o objetivo (ou tarefa) n_1 deve estar completo antes do n_2 , enquanto no segundo caso não há imposição de ordem, ambos são executados em paralelo [3].

A semântica é análoga para o caso do refinamento OR , mas nesse caso não é necessário o cumprimento de ambas, por definição. Assim, as expressões $OR(n_1; n_2)$ e $OR(n_1 \# n_2)$ fazem sentido apenas quando ambos objetivos (ou tarefas) n_1 e n_2 são executados (uma das condições de verdade do OU lógico), pois caso apenas um seja executado, o operador sequencial (ou paralelo) é irrelevante. Dessa forma, quando ambos forem executados, se aplica a mesma definição do caso do refinamento AND .

As anotações $n + k$, $n \# k$, $n @ k$ dizem respeito à forma com que o objetivo (ou tarefa) n deve ser executado: no primeiro caso, se n for repetido k vezes, k instâncias de n devem ser satisfeitas. No segundo caso, k instâncias de n devem ser cumpridas em paralelo. A última, por sua vez, indica que podem ser realizadas no máximo $k - 1$ tentativas de satisfazer n .

Uma anotação com opt indica que seu cumprimento é optativo, podendo ser executado ou ignorado. Já a expressão try , de natureza *if-then-else*, demanda que o sistema tente atingir ao objetivo n , e dependendo do resultado (êxito ou falha), diferentes comportamentos são indicados: executa n_1 no primeiro caso, n_2 no segundo.

Por fim, a expressão $n_1 | n_2$ sugere a execução exclusiva de n_1 ou n_2 .

2.8 ANTLR

ANTLR (Another Tool for Language Recognition) é um gerador de *parser* para realizar leitura, processamento, execução ou tradução de textos estruturados ou arquivos binários [21]. Através do estabelecimento de uma gramática para uma linguagem qualquer criada, ANTLR gera um *parser* para essa linguagem capaz de automaticamente percorrer estruturas de dados como árvores sintáticas, além de também ser capaz de prover acesso aos nós dessa árvore.

O *parser*, ao receber um conteúdo textual, transforma em uma estrutura organizada, como uma árvore sintática abstrata, expressando a representação daquele texto na ordem analisada. Assim, o principal objetivo para utilizar ANTLR é exportar o *parser* de uma linguagem *custom* para uma linguagem alto nível como JAVA, como foi realizado no próprio GODA e explicitado em [7].

A utilização do ANTLR no GODA, no entanto, não é suficiente para assegurar a corretude do modelo CRGM, uma vez que a análise relacionada ao ANTLR é individual a cada variável instanciada no modelo de objetivos. Assim, não é realizado nenhuma verificação em relação à corretude da modelagem, no que diz respeito à correspondência entre as

declarações de objetivos e tarefas e suas respectivas definições posteriores. Dessa forma, o Analisador de Corretude do Modelo Orientado a Objetivos do GODA proposto neste trabalho objetiva complementar a utilização do ANTLR no GODA, a fim de verificar a coerência das declarações e definições das variáveis instanciadas em relação à modelagem.

Na Seção 4.1 será retomada a discussão acerca da importância do ANTLR e as limitações do seu uso no contexto do GODA.

Capítulo 3

Proposta

O modelo CRGM não é suficientemente verificado em relação às regras de tempo de execução e às variáveis declaradas no modelo de objetivos. Como resultado, diversos estados inválidos não estão sendo reconhecidos como tal, fazendo com que esses erros, provenientes de má formação do modelo, sejam transmitidos para o modelo DTMC durante a transformação. O problema disso é que os erros acabam sendo propagados para a análise de dependabilidade, comprometendo os resultados obtidos e causando uma inconsistência do GODA no desempenho de sua função.

Assim, é necessário garantir que regras de runtime pré-determinadas sejam respeitadas em tempo de execução, lançando exceções Java e impedindo o fim da execução do GODA quando um estado inválido se revelar. Isso proporcionará uma redução na chance de erros relacionados à nomenclatura e/ou formação do modelo serem propagados para a análise de dependabilidade propriamente dita.

Este capítulo apresentará o planejamento da solução implementada, iniciando com a exposição das regras de corretude, seguida da forma com que os dados foram identificados para posterior verificação. Posteriormente, serão detalhados e ilustrados os casos inválidos que foram contemplados pelo analisador de corretude proposto neste trabalho, encerrando com a apresentação de modelos de implementação.

3.1 Regras de Corretude

As regras de corretude nada mais são que requisitos que devem ser satisfeitos durante a modelagem para manter o modelo em um estado consistente. Algumas dessas regras foram propostas por Solano *et al.* (2016) [5], e outras foram levantadas por este trabalho, baseando-se em aspectos semânticos e na coerência entre declarações e definições de objetivos/tarefas do modelo de objetivos.

1. Objetivos

- (a) Dois ou mais objetivos não podem ter o mesmo identificador **G#**, onde **#** representa um número inteiro.
- (b) Objetivos não podem se refinar em sub-objetivo e tarefa simultâneamente.
- (c) Objetivos não podem se refinar através de refinamento AND e através de refinamento OR ao mesmo tempo.
- (d) Objetivos não podem possuir anotação de tempo de execução caso não tenham nenhum nó filho.
- (e) O número de nós na anotação de tempo de execução de um objetivo deve ser igual à quantidade de nós filhos dele.
- (f) Todos os identificadores de nós filhos **G#**, onde **#** representa um número inteiro, devem estar presentes na anotação de tempo de execução.
- (g) Objetivos não podem se refinar em mais de uma tarefa.
- (h) Objetivos que possuem dois ou mais filhos devem possuir anotação de tempo de execução.
- (i) Objetivos devem possuir *label* da forma **G#:** **description** [**runtime annotation**], onde **#** representa um número inteiro, respeitando a ordem (deve existir uma descrição).
- (j) Caso a *label* de um objetivo tenha um colchete "[" (para começar a escrita da anotação de tempo de execução), é necessário que seu par "]" exista.
- (k) Caso a *label* de um objetivo tenha um colchete "]" (para terminar a escrita da anotação de tempo de execução), é necessário que seu par "[" exista.

2. Tarefas

- (a) Tarefas em nível um devem ser nomeadas como a seguir: **T#**, onde **#** representa um número inteiro.
- (b) Tarefas devem possuir *label* da forma **T#:** **description** [**runtime annotation**], onde **#** representa um número inteiro, respeitando a ordem (deve existir uma descrição).
- (c) Caso a *label* de uma tarefa tenha um colchete "[" (para começar a escrita da anotação de tempo de execução), é necessário que seu par "]" exista.
- (d) Caso a *label* de um objetivo tenha um colchete "]" (para terminar a escrita da anotação de tempo de execução), é necessário que seu par "[" exista.
- (e) Tarefas não podem se refinar através de refinamento AND e através de refinamento OR ao mesmo tempo.

- (f) Tarefas não podem possuir anotação de tempo de execução caso não tenham nenhum nó filho.
- (g) A quantidade de nós filhos da Tarefa deve ser igual ao número de nós presente na anotação de tempo de execução.
- (h) Duas ou mais tarefas irmãs não podem possuir o mesmo identificador.
- (i) Tarefas que possuem dois ou mais filhos devem possuir anotação de tempo de execução.
- (j) Tarefas a partir do nível 2 devem ser nomeadas como a seguir: $T\#.\#$, onde $\#$ representa um número inteiro.
- (k) Todos os identificadores de nós filhos ($T\#.\#$), onde $\#$ representa um número inteiro, devem estar presentes na anotação de tempo de execução.
- (l) Toda tarefa em nível 2 ou mais deve herdar o identificador do nó pai e ser nomeada a partir dela.

3.2 Planejamento da Solução

Uma vez levantados os tipos de erros que serão endereçados na solução, faz-se necessário também identificar em que estruturas do GODA as informações a respeito do modelo de objetivos podem ser encontradas, e de que forma elas estão organizadas. Essas informações são cruciais para arquitetar uma solução.

Assim, a abordagem primária escolhida foi de identificar e extrair as informações necessárias do modelo e mapear o fluxo de execução do GODA para constatar o melhor ponto para conduzir as verificações acerca do modelo CRGM.

Como o objetivo do analisador de corretude proposto é minimizar o máximo possível a possibilidade de erros serem propagados do modelo CRGM para o modelo DTMC, foi concluído que o ponto ideal para as verificações serem realizadas seria no momento que antecede a transformação para o modelo DTMC.

Posto que foi definido o momento ideal no fluxo de execução para as verificações ocorrerem, basta a aplicação de uma nova metodologia, agora de desenvolvimento, baseada em explorar as estruturas que a arquitetura interna do GODA utiliza para representar o modelo de objetivos, extraíndo as informações e implementando métodos para de fato verificar se o modelo em questão é válido, conforme casos levantados na Seção anterior.

A ideia para o desenvolvimento se pautou em, primeiramente, verificar a coerência entre as declarações de objetivos e tarefas e suas posteriores definições. Nesse sentido, as declarações ocorrem quando um nó define seu refinamento em outros subobjetivos e/ou subtarefas, declarando estes com seus nomes e identificadores em sua anotação de tempo

de execução. A definição, por sua vez, é quando esse nó, que representa o refinamento do objetivo/tarefa pai, é de fato criado e inserido no modelo de objetivos. Portanto, a verificação de coerência é pautada na correspondência, ou não, do nome do objetivo/tarefa no momento de definição, e do nome que foi previamente declarado na anotação de tempo de execução de seu nó superior.

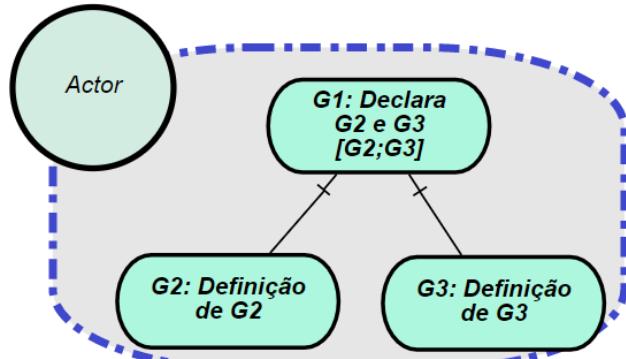


Figura 3.1: Exemplo dos conceitos de declaração e definição.

Assim, conforme a Figura 3.1, a declaração de G2 e G3 ocorre no nó G1, ao inserí-los em sua anotação de tempo de execução. A definição ocorre quando o nó G2 ou G3 é de fato criado e nomeado dessa forma. No entanto, pode ser que no momento da definição o usuário insira um nome diferente do que foi declarado em G1, o que geraria uma inconsistência. Por esse motivo, um dos objetivos do desenvolvimento do Analisador de Corretude é verificar a consistência entre as declarações e as definições na modelagem.

3.3 Casos Contemplados

O desenvolvimento buscou alcançar êxito no reconhecimento dos estados inválidos através da satisfação das vinte e três (23) regras de corretude na Seção 3.1, seguindo a mesma nomenclatura.

1. Objetivos

- (a) Caso em que dois ou mais objetivos possuem o mesmo identificador. (Figura 3.2)
 - Ação: lançamento de exceção com a seguinte mensagem: "*Goals IDs are not unique*".
- (b) Caso em que um objetivo está se decompondo em objetivos e tarefa simultaneamente. (Figura 3.3)

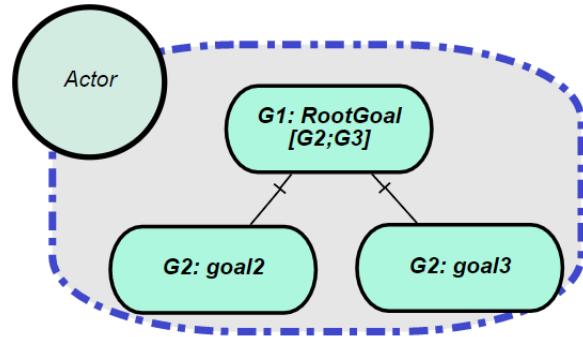


Figura 3.2: Ilustração para o caso 1a.

- Ação: lançamento de exceção com a seguinte mensagem: "*Goals cannot be refined into tasks and subgoals at the same time*".

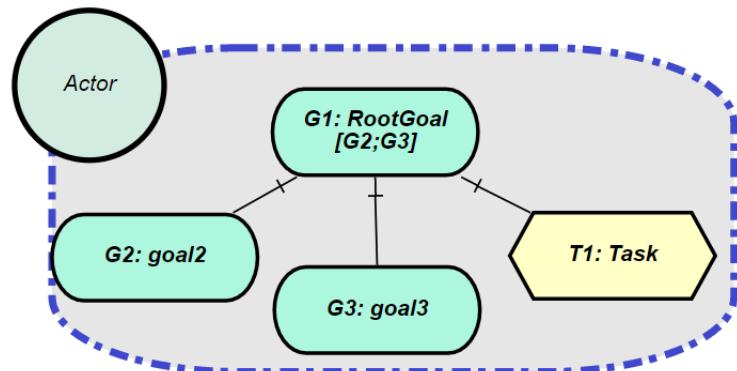


Figura 3.3: Ilustração para o caso 1b.

- (c) Caso em que um objetivo se refina através de refinamento AND e refinamento OR ao mesmo tempo. (Figura 3.4)
 - Ação: lançamento de exceção com a seguinte mensagem: "*Goal is being refined through AND and OR-decompositions at the same time*".
- (d) Caso em que um objetivo possui anotação de tempo de execução sem possuir nenhum filho. (Figura 3.5)
 - Ação: lançamento de exceção com a seguinte mensagem: "*Goal presents a runtime annotation but it doesn't have any children nodes*".
- (e) Caso em que um objetivo possui número de filhos diferente da quantidade de nós na anotação de tempo de execução (Figura 3.6)
 - Ação: lançamento de exceção com a seguinte mensagem: "*The amount of children does not match the amount of goals presented in the runtime annotation*".

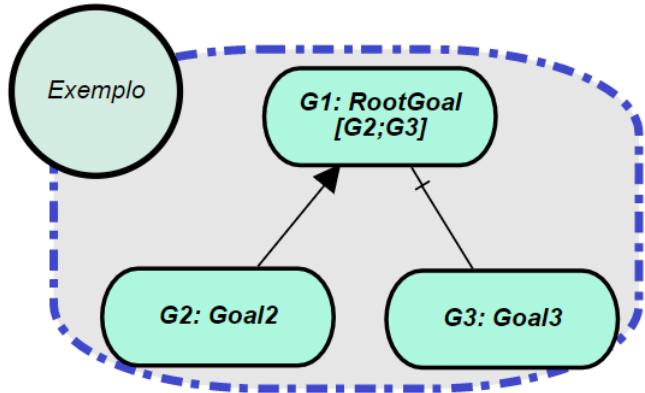


Figura 3.4: Ilustração para o caso 1c.

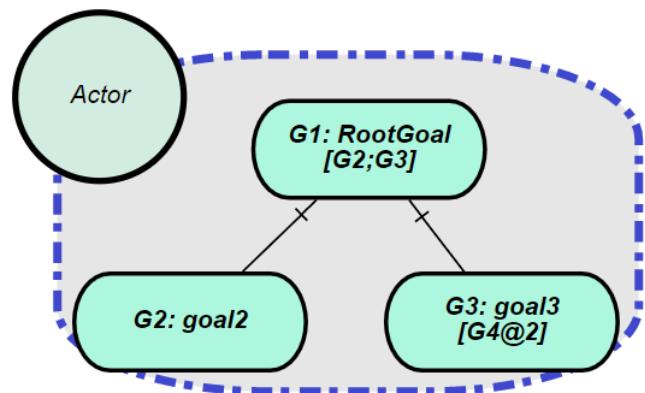


Figura 3.5: Ilustração para o caso 1d.

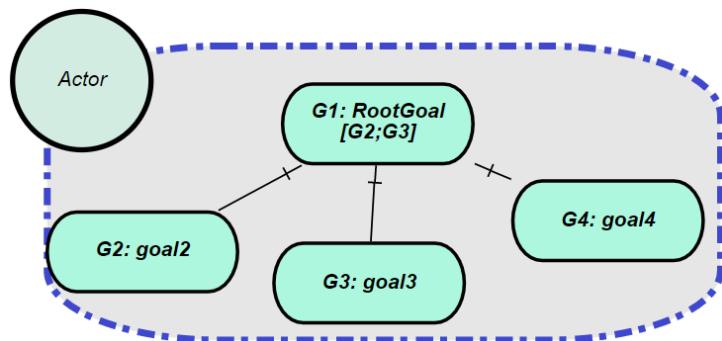


Figura 3.6: Ilustração para o caso 1e.

(f) Caso em que um objetivo possui ao menos um nó filho que não consta na anotação do tempo de execução (caso a anotação de tempo de execução não seja nula). (Figura 3.7)

- Ação: lançamento de exceção com a seguinte mensagem: "*There is at least one goal that is not in the runtime annotation*".

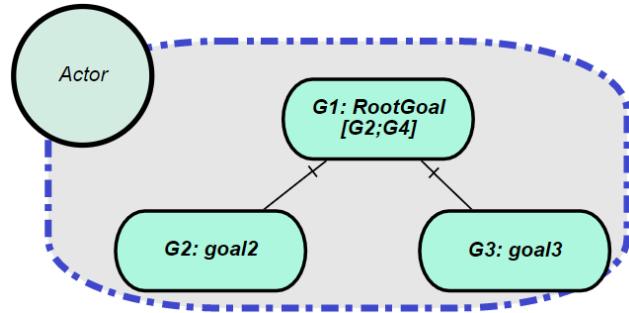


Figura 3.7: Ilustração para o caso 1f.

(g) Caso em que um objetivo se refina em mais de uma tarefa. (Figura 3.8)

- Ação: lançamento de exceção com a seguinte mensagem: *"Goals refine in just one task, not multiple"*.

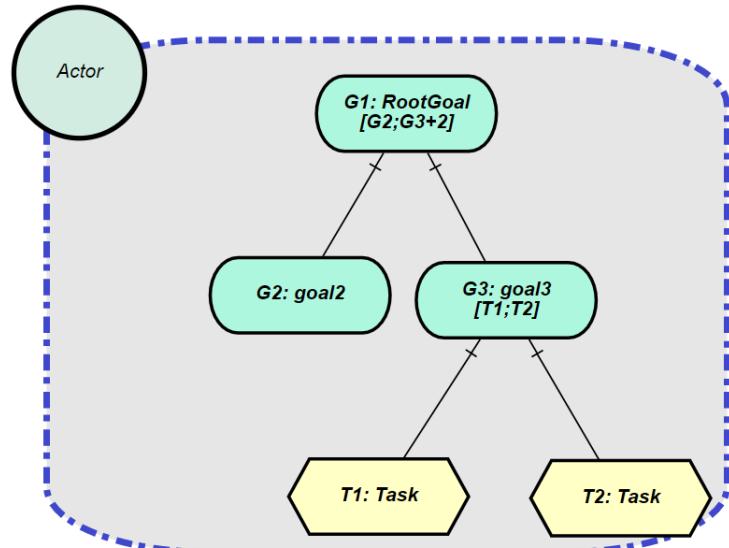


Figura 3.8: Ilustração para o caso 1g.

(h) Caso em que um objetivo possui dois ou mais filhos e não possui anotação de tempo de execução. (Figura 3.9)

- Ação: lançamento de exceção com a seguinte mensagem: *"Goals with 2 or more children must have runtime annotation"*.

(i) Caso em que a anotação de tempo de execução aparece antes da descrição. Isto é, a *label* está fora do padrão **G#:** *description [runtime annotation]*. (Figura 3.10)

- Ação: lançamento de exceção com a seguinte mensagem: *"Label must respect the following order: (G/T)#: description [runtime annotation]"*.

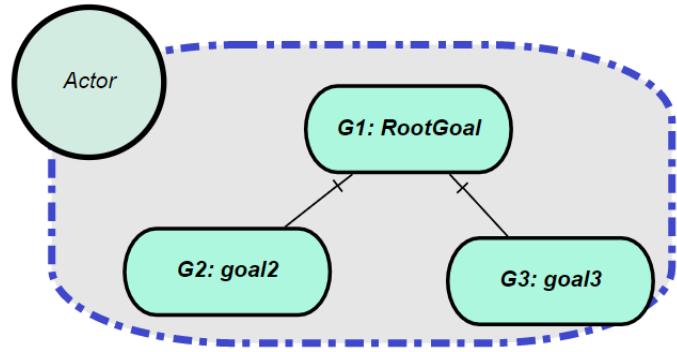


Figura 3.9: Ilustração para o caso 1h.

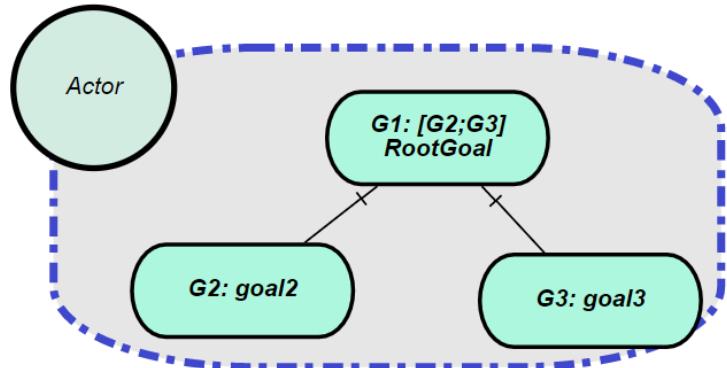


Figura 3.10: Ilustração para o caso 1i.

- (j) Caso em que existe um colchete para iniciar a anotação de tempo de execução, mas não há o fechamento dele. (Figura 3.11)

- Ação: lançamento de exceção com a seguinte mensagem: "*Closing bracket is missing*".

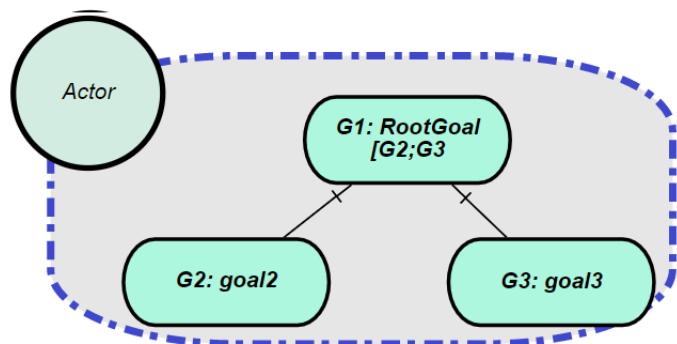


Figura 3.11: Ilustração para o caso 1j.

- (k) Caso em que existe um colchete para finalizar a anotação de tempo de execução, mas não há a abertura dele. (Figura 3.12)

- Ação: lançamento de exceção com a seguinte mensagem: "*Open bracket is missing*".

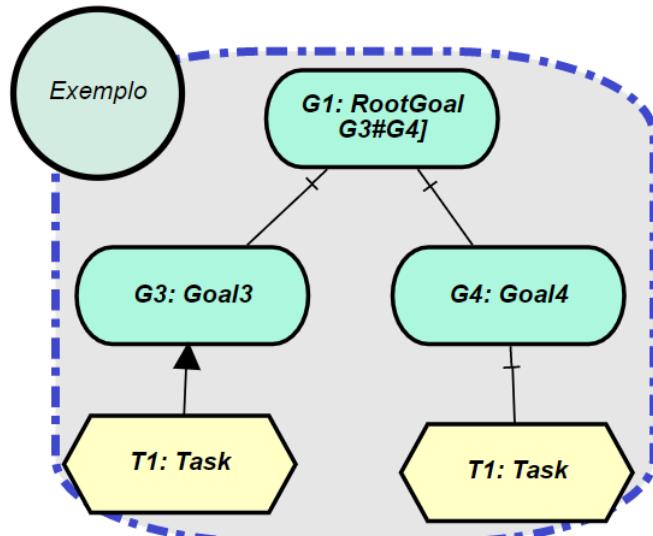


Figura 3.12: Ilustração para o caso 1k.

2. Tarefas

- Caso em que uma tarefa de nível 1 está com a nomenclatura fora da forma estipulada ($T\#$). (Figura 3.13)
- Ação: lançamento de exceção com a seguinte mensagem: "*Level 1 tasks must be named accordingly ($T\#$)*".

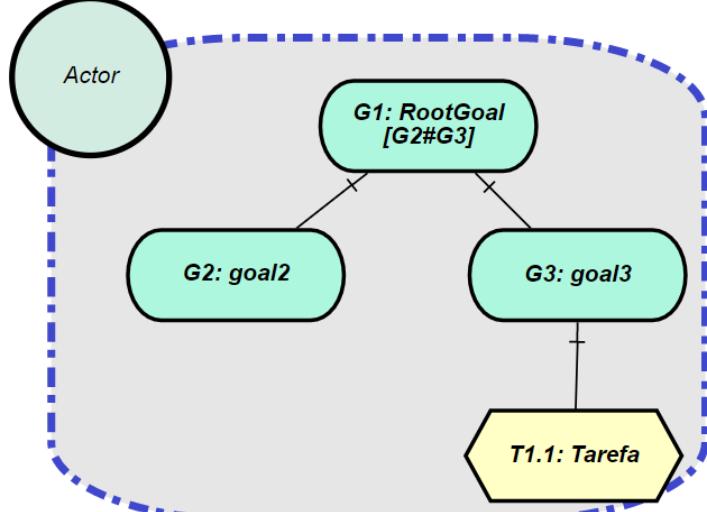


Figura 3.13: Ilustração para o caso 2a.

- (b) Caso em que a anotação de tempo de execução aparece antes da descrição. Isto é, a *label* está fora do padrão **T#:** **description** [runtime annotation]. (Figura 3.14)

- Ação: lançamento de exceção com a seguinte mensagem: "*Label must respect the following order: (G/T) #: description [runtime annotation]*".

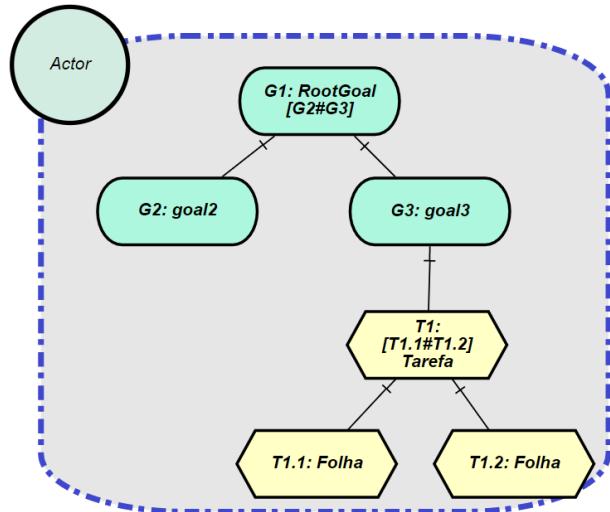


Figura 3.14: Ilustração para o caso 2b.

- (c) Caso em que existe um colchete para iniciar a anotação de tempo de execução mas não há o fechamento dele. (Figura 3.15)

- Ação: lançamento de exceção com a seguinte mensagem: "*Closing bracket is missing*".

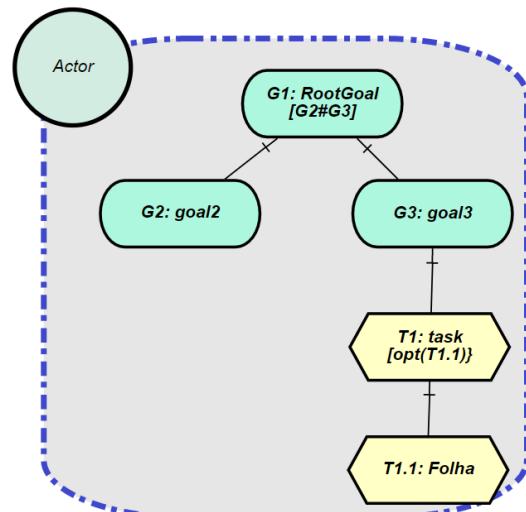


Figura 3.15: Ilustração para o caso 2c: chave foi utilizada no lugar do colchete em T1.

- (d) Caso em que existe um colchete para terminar a anotação de tempo de execução mas não há a abertura dele. (Figura 3.16)

- Ação: lançamento de exceção com a seguinte mensagem: "*Open bracket is missing*".

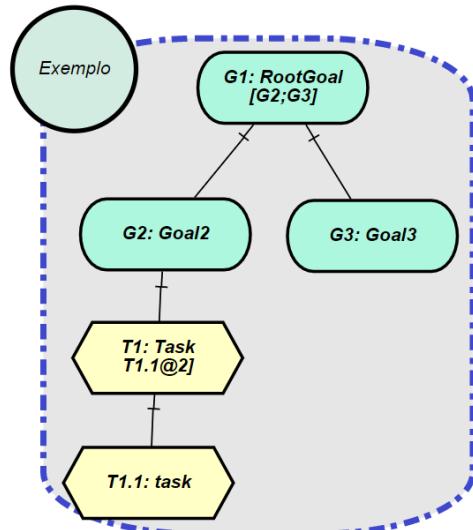


Figura 3.16: Ilustração para o caso 2d.

- (e) Caso em que uma tarefa se refina através de refinamento AND e refinamento OR ao mesmo tempo. (Figura 3.17)

- Ação: lançamento de exceção com a seguinte mensagem: "*Task is being refined through AND and OR-decompositions at the same time.*".

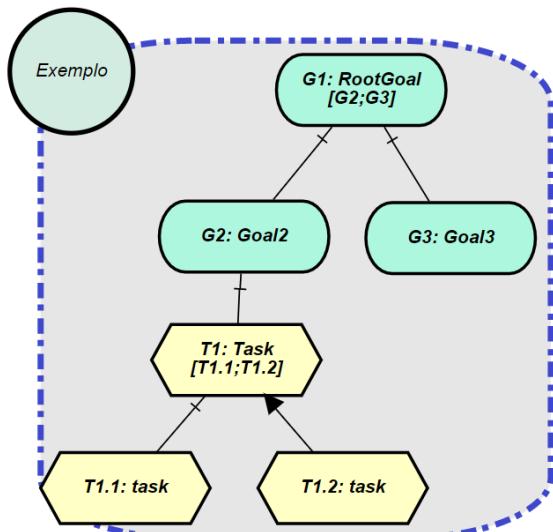


Figura 3.17: Ilustração para o caso 2e.

(f) Caso em que uma tarefa possui anotação de tempo de execução sem possuir nenhum nó filho. (Figura 3.18)

- Ação: lançamento de exceção com a seguinte mensagem: "*Task presents a runtime annotation but it doesn't have any children nodes*".

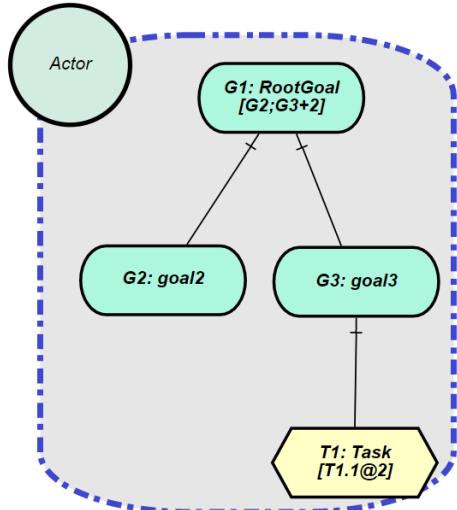


Figura 3.18: Ilustração para o caso 2f.

(g) Caso em que uma tarefa possui quantidade de filhos diferente do número de nós na anotação de tempo de execução. (Figura 3.19)

- Ação: lançamento de exceção com a seguinte mensagem: "*The amount of children does not match the amount of tasks presented in the runtime annotation*".

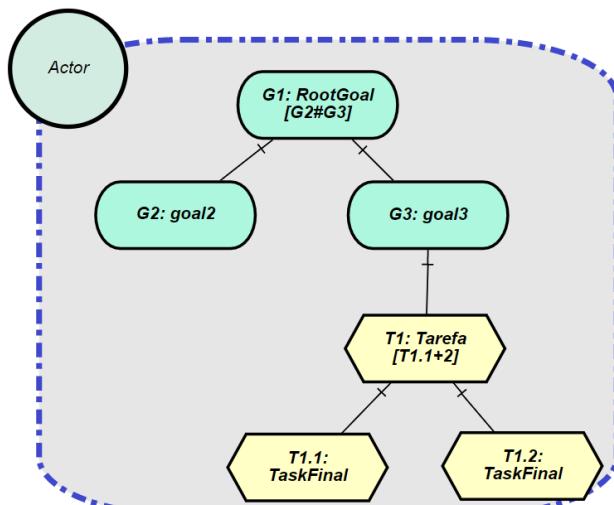


Figura 3.19: Ilustração para o caso 2g.

(h) Caso em que duas ou mais tarefas irmãs possuem o mesmo identificador. (Figura 3.20)

- Ação: lançamento de exceção com a seguinte mensagem: "*at least two subtasks have the same exactly name*".

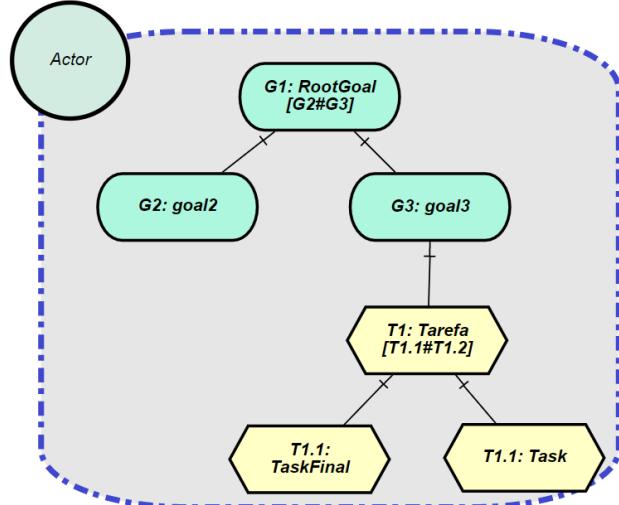


Figura 3.20: Ilustração para o caso 2h.

(i) Caso em que uma tarefa possui ao menos dois filhos e não possui anotação de tempo de execução. (Figura 3.21)

- Ação: lançamento de exceção com a seguinte mensagem: "*Tasks with 2 or more children must have runtime annotation*".

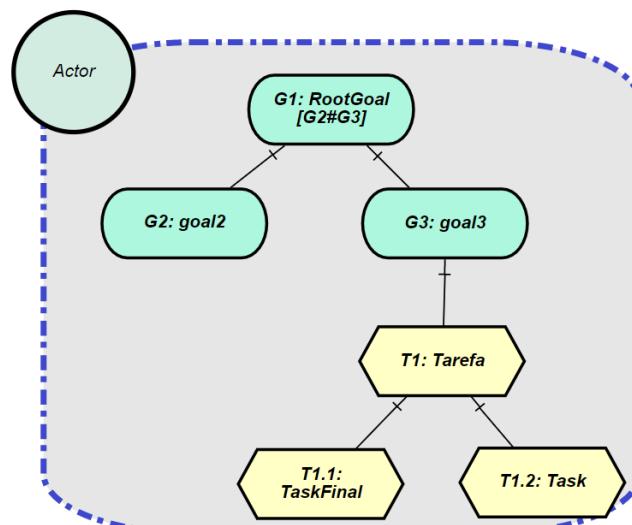


Figura 3.21: Ilustração para o caso 2i.

- (j) Caso em que uma tarefa de nível dois ou mais está com a nomenclatura fora da forma estipulada ($T\#\#\#$). (Figura 3.22)

- Ação: lançamento de exceção com a seguinte mensagem: "*Tasks that are not on level 1 must have $T\#\#\#$ form.*".

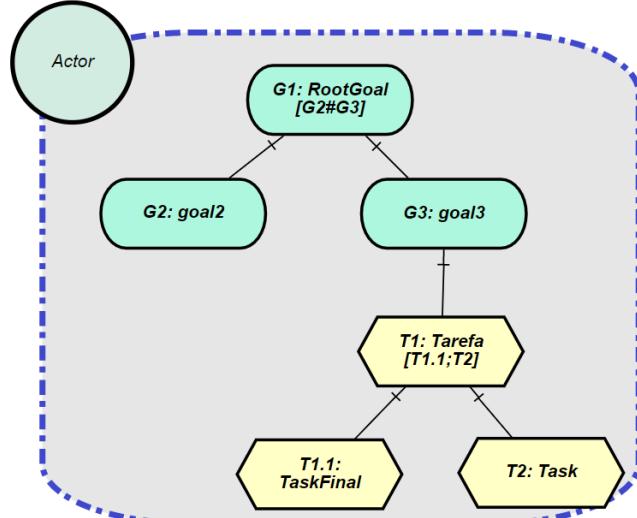


Figura 3.22: Ilustração para o caso 2j.

- (k) Caso em que uma tarefa possui ao menos um nó filho que não consta na anotação do tempo de execução. (Figura 3.23)

- Ação: lançamento de exceção com a seguinte mensagem: "*IDs in the run-time annotation dont not match subtasks IDs*".

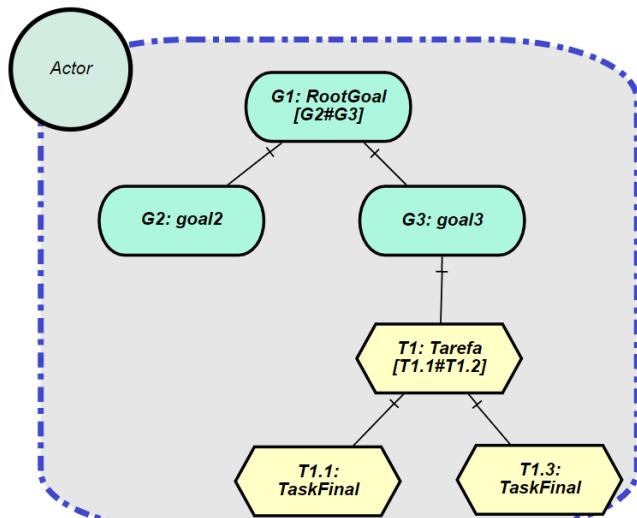


Figura 3.23: Ilustração para o caso 2k.

(l) Caso em que uma tarefa de nível 2 ou mais não herda a primeira parte do seu identificador do nó pai. (Figura 3.24)

- Ação: lançamento de exceção com a seguinte mensagem: "*subtask must inherit first part of its ID from its parent.*".

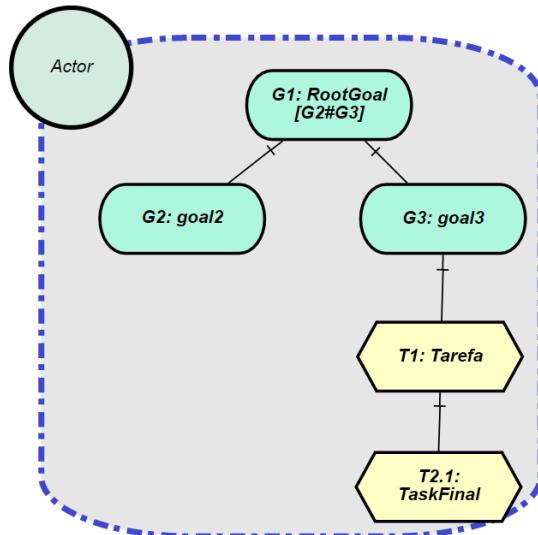


Figura 3.24: Ilustração para o caso 2l.

3.4 Modelo de Implementação

Para a verificação de todos os vinte e três (23) casos explicitados na Seção 3.1, foram implementados quatro (4) métodos utilizando a IDE Eclipse e a linguagem Java. Além disso, uma alteração no FrontEnd da aplicação foi realizada e será descrita na Seção 3.4.1.

O desenvolvimento se pautou, em geral, na obtenção da informação necessária para a verificação, e em seguida a realização de *parse* nas *strings* de anotação de tempo de execução, *labels*, e em listas encadeadas referentes às decomposições de cada nó.

A implementação do piStarGODA utiliza estruturas específicas - listas encadeadas - que mapeiam a modelagem realizada pelo usuário para o tratamento interno. Assim, a árvore de representação do modelo de objetivos na prática é um conjunto de listas encadeadas responsáveis por:

1. Armazenar a lista de refinamento de cada objetivo.
2. Armazenar a lista de refinamento de cada tarefa.

Essas listas encadeadas possuem como elementos objetos, e não apenas valores constantes. Assim, os atributos dos objetos permitem o acesso a informações importantes que

constam na modelagem, como o tipo do refinamento do objetivo, string que representa a anotação de tempo de execução, informação acerca de quem é o nó pai, entre outras. Assim, a implementação deste projeto se pautou na utilização dessas listas encadeadas já implementadas no GODA para realizar as verificações acerca da modelagem.

O Algoritmo 1 a seguir ilustra a forma com que foram conduzidas as implementações dos métodos.

Algorithm 1: Checking Goals Decompositions part1

```

1 let G be the set of goals in a goal model
2 for each  $g \in G$  do
3   S  $\leftarrow$  set of subgoals of g;
4   P  $\leftarrow$  set of subtasks of g;
5   R  $\leftarrow$  runtime annotation of g;
6   if  $S \neq \emptyset, P \neq \emptyset$  then
7     | Exception: goals cannot be refined into tasks and goals simultaneously;
8   else
9     | if  $R \neq \text{null}$  then
10    |   | if  $S = \emptyset, P = \emptyset$  then
11      |     | Exception: goal presents a runtime annotation but it does not have
12        |     | any children nodes;
13    |   | else
14    |     |   | for each char  $c \in R$  do
15    |     |     |   | if  $c = "T"$  or  $c = "G"$  then
16    |     |       |       |   | children  $\leftarrow$  children+1;
17    |     |   | if  $S = \emptyset$  then
18    |     |     |   | if  $|P| \geq 2$  or  $|P| \neq \text{amount of children in the runtime annotation}$ 
19    |     |       |       | then
20    |     |         |         |   | Exception: Goals refine in just one task, not multiple;
21    |     |       |       | else
22    |     |         |         |   | if  $|S| \neq \text{children in the runtime annotation}$  then
23    |     |           |           |   | Exception: The amount of subgoals does not match the
24    |     |             |             |   | amount of children presented in the runtime annotation;

```

Nesse trecho de código em particular, são tratados os seguintes casos:

1. Caso em que um objetivo está se decompondo em objetivos e tarefa simultaneamente (Figura 3.3);

2. Caso em que um objetivo possui anotação de tempo de execução sem possuir nenhum filho (Figura 3.4);
3. Caso em que um objetivo se refina em mais de uma tarefa (Figura 3.7);
4. Caso em que um objetivo possui número de filhos diferente da quantidade de nós na anotação de tempo de execução (Figura 3.5).

A verificação desses casos depende da obtenção de listas encadeadas que representam as decomposições de cada objetivo presente no modelo. Além disso, também é necessário a identificação da anotação de tempo de execução presente na *label* de cada objetivo.

Para o primeiro caso, foi necessário realizar uma verificação em relação ao conteúdo dos conjuntos que representam os possíveis subobjetivos e subtarefas. Caso ambos sejam não-vazios, isso indica que o objetivo se refina em ao menos um objetivo e uma tarefa simultaneamente. Portanto, nesse caso, como trata-se de um cenário inválido, de acordo com as regras de corretude na Seção 3.1, uma exceção é lançada, interrompendo a execução do framework.

A verificação do segundo caso é similar, mas depende de uma informação que o primeiro cenário não demanda: a anotação de tempo de execução do objetivo em questão. Caso o objetivo possua anotação de tempo de execução não nula, isso indica que ele deve possuir ao menos um nó filho. Essa característica, por sua vez, é mensurada a partir da *if clause* na linha 10 do pseudocódigo explicitado acima, em que é verificado se o objetivo é decomposto em algum nó, qualquer que seja ele. Caso não seja, então o objetivo de fato possui uma anotação de tempo de execução sem possuir nós filhos, o que representa um estado inválido. Da mesma forma, uma exceção é lançada.

Para o terceiro caso, foi necessário verificar a cardinalidade do conjunto P , que reúne as tarefas em que o objetivo em questão se refina. O cenário em que P possui mais de um elemento implica no fato de que o objetivo está se decompondo em mais de uma tarefa, o que é um erro, de acordo com as regras de corretude. Assim, lança-se uma exceção interrompendo o fluxo de execução do framework.

Por fim, o último caso engloba o cenário em que o conjunto S , que representa os subobjetivos em que o objetivo em questão se refina, é não nulo. Dessa forma, deve-se comparar a quantidade de filhos presente na anotação de tempo de execução com a quantidade de elementos desse conjunto. Para obter a primeira informação, é necessário realizar um *parse* na string da anotação de tempo de execução, representado nas linhas 13-15, incrementando as ocorrências de objetivos declarados. Assim, a comparação da cardinalidade do conjunto S com a quantidade de elementos na anotação de tempo de execução indica a corretude, ou não, do modelo. Caso não sejam iguais, uma exceção é lançada, caracterizando um estado inválido.

A continuação deste método possui um segundo trecho de código, responsável por outras verificações. O pseudo código deste trecho é descrito no Algoritmo 2 a seguir.

Algorithm 2: Checking Goals Decompositions part2

```

1 let G be the set of goals in a goal model
2 for each  $g \in G$  do
3    $S \leftarrow$  set of subgoals of  $g$ ;
4    $R \leftarrow$  runtime annotation of  $g$ ;
5   if  $g.decomposition = (AND \mid OR)$ ,  $|S| \geq 2$  then
6     if  $R = null$  then
7       Exception: Goals with 2 or more children must have a runtime
         annotation;
8     for each  $s$  in  $S$  do
9        $sId \leftarrow$  s identifier;
10       $found \leftarrow R.parse(sId)$ ;
11      if  $!found$  then
12        Exception: There is at least one subgoal that is not in the runtime
          annotation;

```

No Algoritmo 2, é tratado o caso em que um objetivo é decomposto utilizando relação *AND* ou *OR* e possui ao menos dois nós filhos. Nesse cenário, conforme condições apresentadas na Seção 3.1, são devidamente apurados:

1. A existência (ou não) de uma anotação de tempo de execução (caso não haja, uma exceção é lançada);
 - Método: a realização deste caso se pautou na verificação do valor contido no atributo *rtregex* do objetivo *gc* (instância da classe *Goal Container*) via método *get*. A checagem quanto ao conteúdo da anotação de tempo de execução foi representada na linha 6 do pseudo código acima.
2. Se o identificador de todos os nós filhos estão devidamente assinalados na anotação de tempo de execução (caso contrário, uma exceção é lançada).
 - Método: para essa verificação foi necessário realizar *parse* na string que representa a anotação de tempo de execução, realizando comparações de carácter a carácter com o identificador de cada nó filho (obtido através da utilização do comando *declist.get(i).getName().split("\\|:")[0]*). A representação desse processo no pseudo código está nas linhas 8-12.

Dessa forma, os métodos foram implementados conforme raciocínios expostos nos algoritmos em pseudo código explicitados anteriormente. Para verificar, na íntegra, o código, pode-se acessar o endereço eletrônico:

github.com/lesunb/pistargodaintegration/tree/crgm_correctness_checker.

Nele está contido o projeto do piStarGODA, já acrescido do Analisador de Tipo proposto neste trabalho. Os códigos foram inseridos nos seguintes arquivos: RTGoreProducer.java, AgentDefinition.java. O nome dos métodos são: *checkUniqueIdentifier*, *checkGoalDecomposition*, *checkPlanDecomposition*, *checkOrder*.

3.4.1 Mudança no Front-end

A atual versão do GODA foi fruto de uma integração entre o dito framework e a ferramenta Pi-Star, em um trabalho que almejou desacoplar o GODA do Eclipse e viabilizar a utilização do framework em uma plataforma web orientada a serviços [4]. No entanto, nessa transição entre o projeto original do GODA exposto em [3] e a versão atualizada, houve uma pequena mudança relativa à coloração do modelo de objetivos. A atual versão do GODA é monocromática, o que a princípio não parece ser uma questão importante, mas de certo modo aumenta a probabilidade de uma incorretude ocorrer: uma tarefa ser nomeada como se fosse um objetivo e/ou vice-versa, pois a única diferenciação entre eles é o formato de cada um, como exposto na Figura 3.25.



Figura 3.25: Parte do menu do piStarGODA [4].

Dessa forma, a diferenciação entre objeto e tarefa, no que diz respeito à forma de representação, se pauta no fato que o objeto é representado por um formato oval, enquanto uma tarefa é representada por um polígono. No entanto, apesar de existir uma diferenciação, isso pode não ser suficiente para impedir que erros sejam cometidos, ainda mais em modelos suficientemente grandes, onde a quantidade de objetivos/tarefas é muito grande para atentar para a forma de cada um ao renomear, criar, mover entidades. O modelo a seguir ilustra esse cenário:

Além de aumentar a probabilidade de erros como esse ocorrerem, o monocromatismo também dificulta a compreensão dos modelos, pois a princípio leva muito tempo para definir onde estão e quais são os objetivos e tarefas (inclusive para detectar erros como o exemplificado). Isso faz com que o processo de compreensão seja desgastante e desmotivador, a depender do tamanho do modelo.

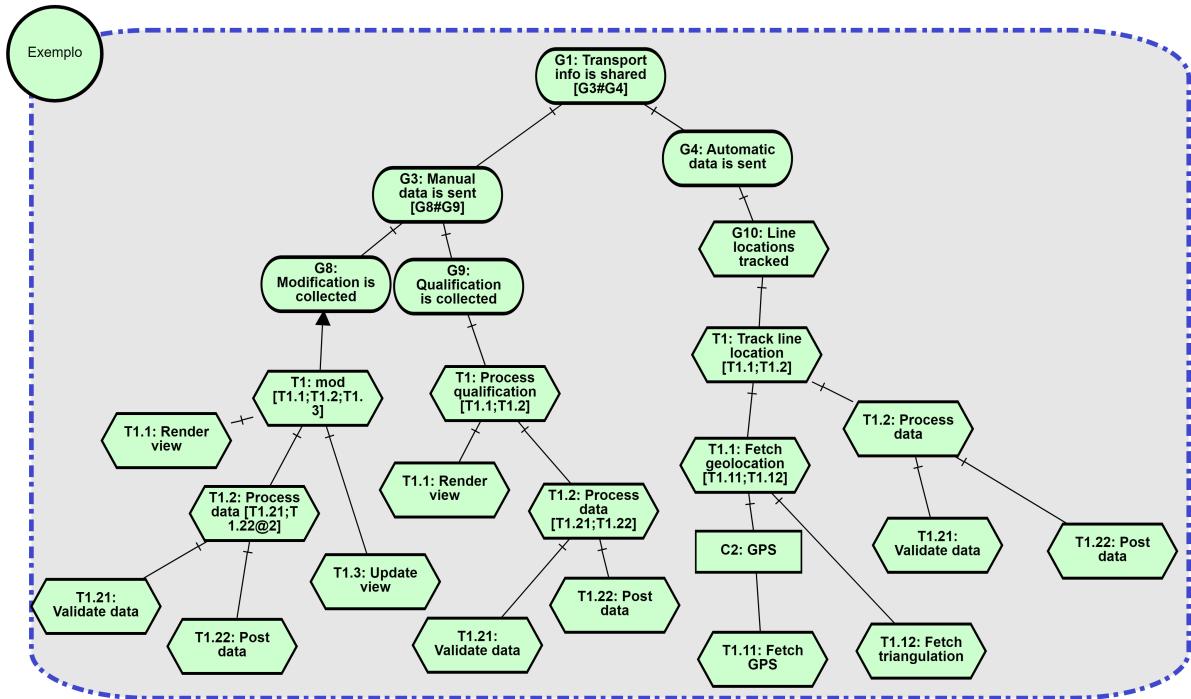


Figura 3.26: A label "G10: Line Locations Tracked" está inserida em uma Tarefa [3].

Tendo em vistas os problemas associados a essa abordagem, este trabalho propõe a utilização de cores para diferenciação das entidades a serem modeladas. Essa medida visa proporcionar mais leveza na leitura de modelos, como também uma redução significativa na probabilidade de erros de difícil diagnóstico como o exemplificado anteriormente acontecerem. Um exemplo das novas cores podem ser visualizadas na Figura 3.27

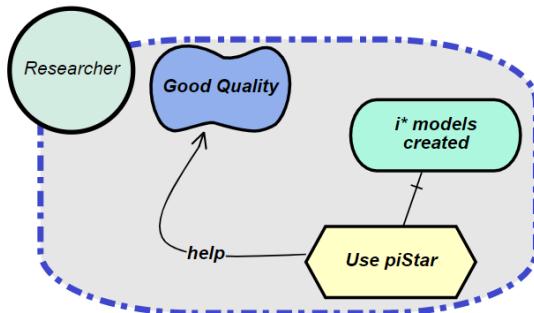


Figura 3.27: Ilustrando diferenciação por cores.

Além da mudança de cores, uma segunda alteração foi realizada no *front-end*: a adoção de mensagens de erro informativas. A atual versão do GODA utiliza apenas uma mensagem padrão de erro, como pode ser observado na Figura 3.28 a seguir:

No entanto, como pode ser antecipado, esse tipo de mensagem não é *user-friendly*, no sentido de que não informa qual foi a regra violada, e tampouco indica em que parte

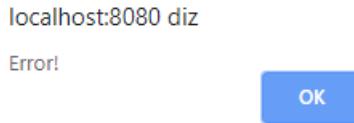


Figura 3.28: Mensagem de erro fixa.

da modelagem o erro se encontra. Isso é um problema, especialmente para modelos suficientemente grandes, uma vez que demanda que o usuário verifique cada variável definida no modelo procurando por algo que possa ter desencadeado o comportamento inesperado.

Assim, este trabalho propõe, também, a adoção de mensagens de erro informativas para guiar o usuário na identificação e compreensão dos erros eventualmente lançados pelo GODA, visando aumentar a praticidade e facilitar o rastreio de erros no modelo. A Figura 3.29 a seguir ilustra um exemplo de uma mensagem de erro informativa.

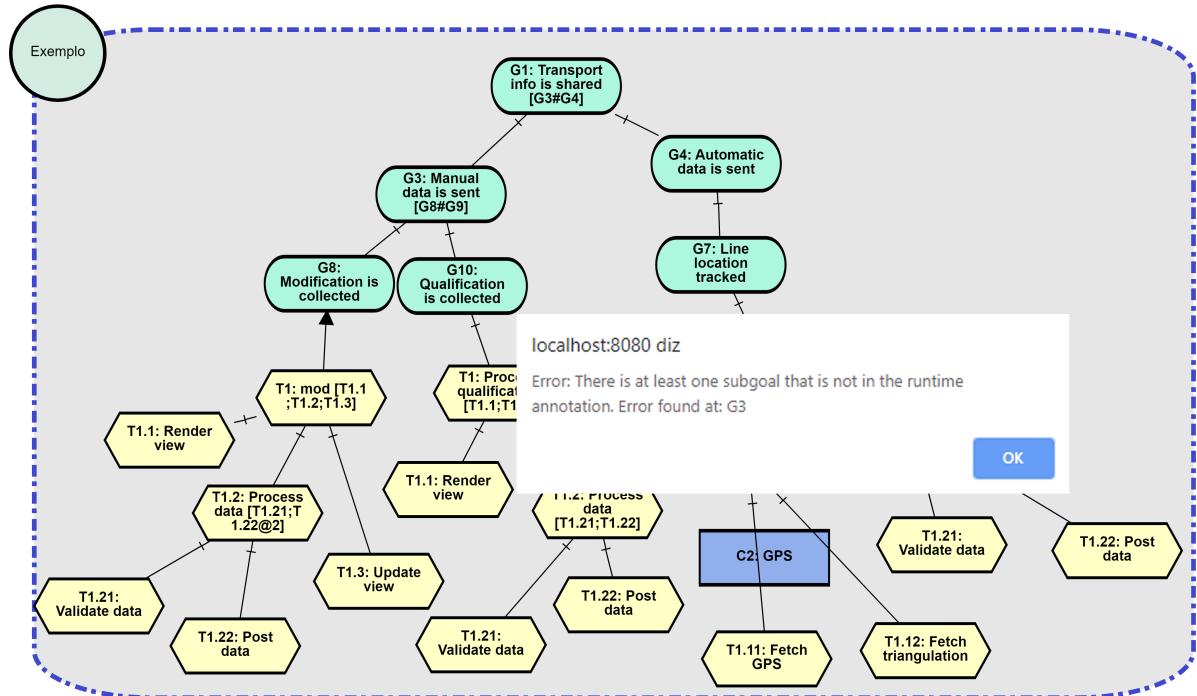


Figura 3.29: Mensagem de erro informativa. [3].

Para a captura das mensagens de erro do *back-end*, foi utilizada a API em forma de objeto denominada *XMLHttpRequest*, ou XHR. Esse objeto possui métodos para a transferência de dados entre cliente e servidor, e propriedades como *responseText*, que contém o conteúdo textual de uma resposta a uma solicitação http. Ou seja, essa propriedade informa ao cliente que efetuou a requisição XMLHttpRequest informações importantes

sobre o status da resposta do servidor, incluindo o conteúdo textual da mensagem de um eventual erro. Dessa forma, é necessário apenas projetar no *front-end* a mensagem de erro que está na *response*.

Capítulo 4

Resultados

Para avaliar a corretude dos métodos de verificação propostos neste trabalho foram utilizados os mesmos testes funcionais que foram desenvolvidos no projeto em [5], que por sua vez também foram utilizados para verificar o trabalho em [4]. Assim, a adoção dos mesmos casos de teste que foram utilizados em trabalhos anteriores possibilita a avaliação da evolução do GODA no decorrer de suas versões no que diz respeito à consistência e confiabilidade.

Para a execução do suite de testes, basta utilizar a opção *Run As JUnit Test* na IDE Eclipse. O resultado será composto dos resultados dos testes associados às classes de equivalência descritas na Tabela 4.1 [5], [4].

A Tabela 4.2, por sua vez, já apresenta os resultados obtidos da realização dos testes funcionais. Os números em laranja indicam os casos em que o GODA falhava em sua última versão, evidenciada em [4] (a mais atual até a realização deste trabalho), mas que foram devidamente corrigidos com o desenvolvimento deste projeto, enquanto os números em cor preta indicam os casos em que o GODA já apresentava o comportamento correto.

Os resultados dos casos de teste serão apresentados um a um a seguir:

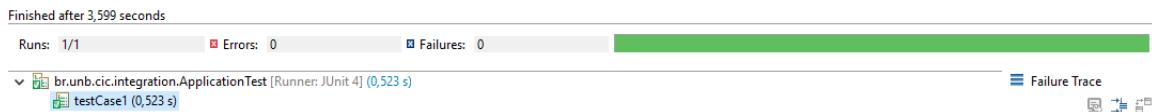


Figura 4.1: Resultado do caso de teste 1.

Errata: Este modelo foi originalmente realizado com o intuito de ser um modelo válido. No entanto, nos trabalhos em [5] e [4], o modelo exposto tinha um pequeno erro de digitação que invalidaria o modelo. Esse detalhe passou despercebido pelos trabalhos anteriores mas foi detectado pela verificação do caso 2j, da Seção 3.1, que diz: "Caso a *label* de uma tarefa tenha um colchete "["(para começar a escrita da anotação de tempo de

execução), é necessário que seu par "]" "exista". No modelo original a chave "}" foi utilizada no lugar do colchete "]", um erro de digitação quase imperceptível. (Figura 4.2).

Assim, como tratou-se de um erro de digitação, o modelo foi devidamente corrigido e o teste foi executado em cima do modelo que de fato representava um estado válido. Note que esse tipo de erro reforça a necessidade de soluções como as propostas neste trabalho.

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

Tabela 4.1: Classes de equivalência e respectivos casos de teste [4],[5]

A ilustração de todos os casos de teste aplicados podem ser consultadas no fim deste trabalho, em anexo.

Classe de Equivalência	Saída Esperada	Saída Não Esperada
<i>CorrectModelTest</i>	1, 2	-
<i>LabelPatternTest</i>	3, 4, 5, 6	-
<i>NoChildTest</i>	7, 8	-
<i>OneChildTest</i>	9, 10	-
<i>MultipleChildrenTest</i>	11, 12, 13, 14	-
<i>SameIdentifierGoalTest</i>	15, 16	-
<i>SameIdentifierTaskTest</i>	17, 18	-
<i>MultipleLevelTaskIdentifierTest</i>	19, 20	-
<i>NoRuntimeAnnotationTest</i>	21, 22	-
<i>RuntimeAnnotationReferenceTest</i>	23, 24	-
<i>RuntimeAnnotationPatternTest</i>	25, 26, 27, 28, 29, 30, 31	-
<i>OperandOperatorTest</i>	32, 33	-
<i>ContextAnnotationValueTest</i>	34, 35	-

Tabela 4.2: Resultado dos testes.

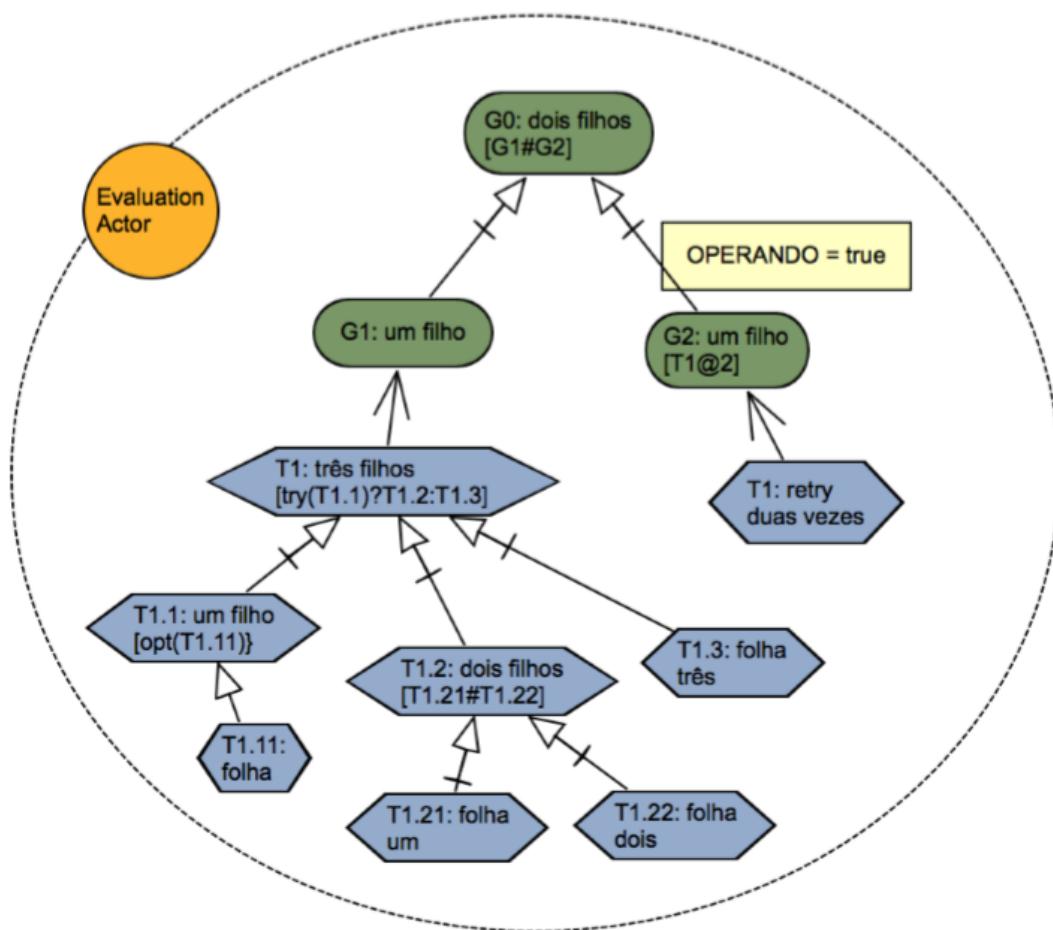


Figura 4.2: Errata: erro de digitação na tarefa T1.1 na realidade tornaria o modelo incorreto [5].

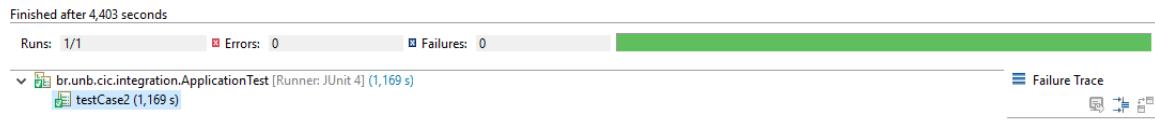


Figura 4.3: Resultado do caso de teste 2.



Figura 4.4: Resultado do caso de teste 3.

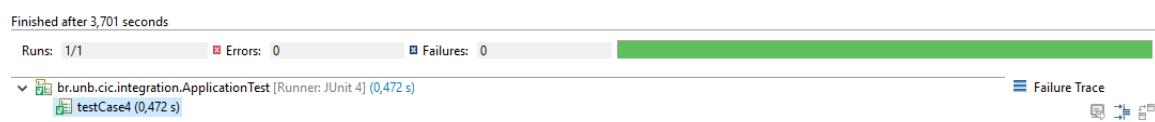


Figura 4.5: Resultado do caso de teste 4.

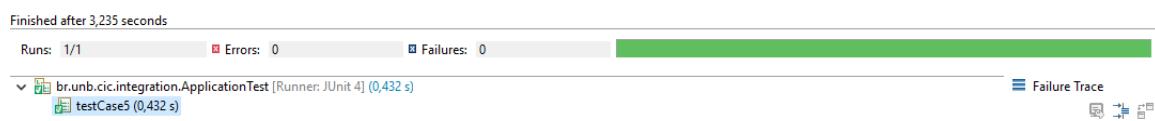


Figura 4.6: Resultado do caso de teste 5.

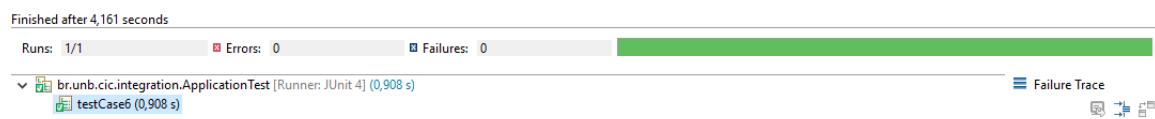


Figura 4.7: Resultado do caso de teste 6.

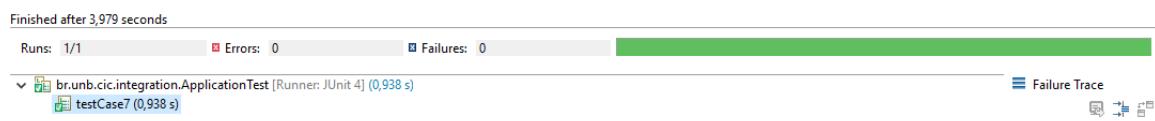


Figura 4.8: Resultado do caso de teste 7.

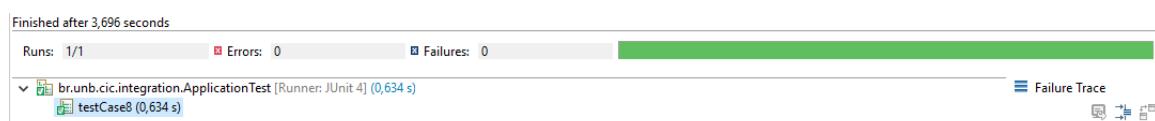


Figura 4.9: Resultado do caso de teste 8.

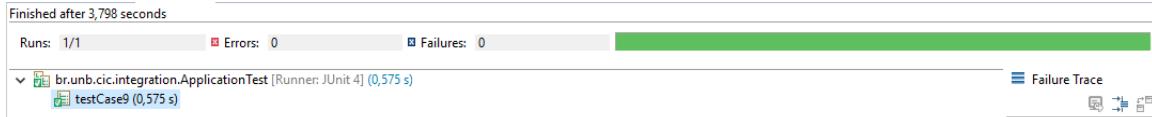


Figura 4.10: Resultado do caso de teste 9.



Figura 4.11: Resultado do caso de teste 10.

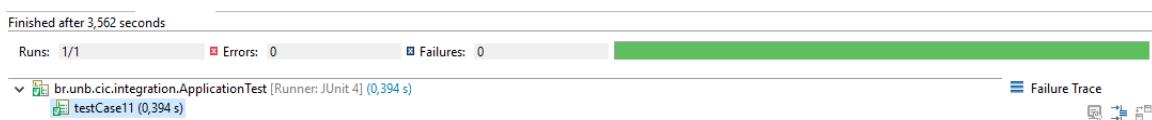


Figura 4.12: Resultado do caso de teste 11.

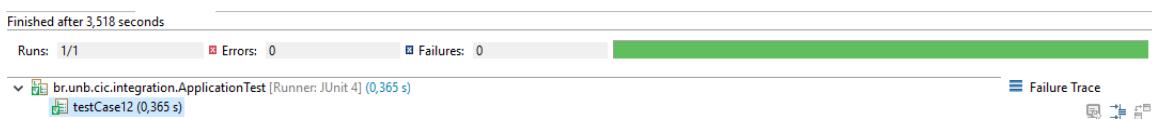


Figura 4.13: Resultado do caso de teste 12.

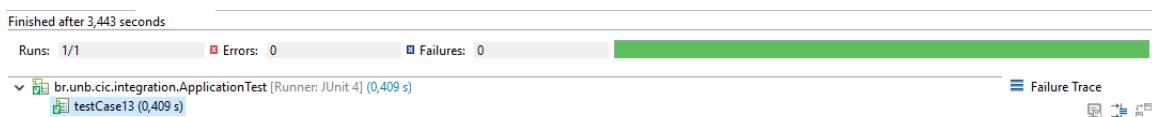


Figura 4.14: Resultado do caso de teste 13.

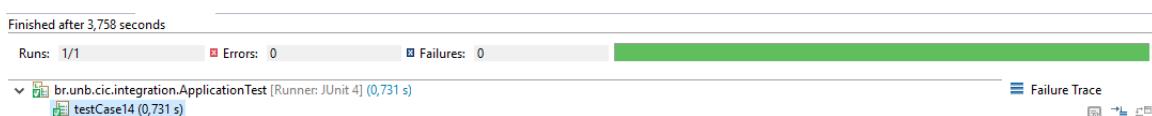


Figura 4.15: Resultado do caso de teste 14.

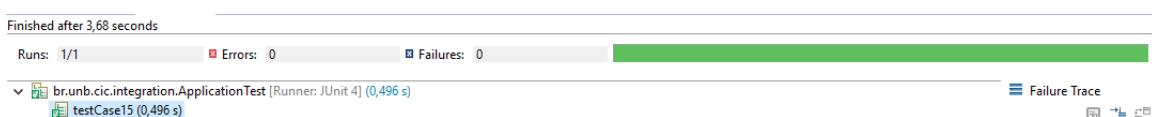


Figura 4.16: Resultado do caso de teste 15.



Figura 4.17: Resultado do caso de teste 16.

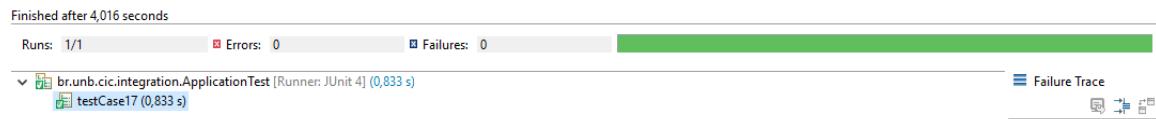


Figura 4.18: Resultado do caso de teste 17.

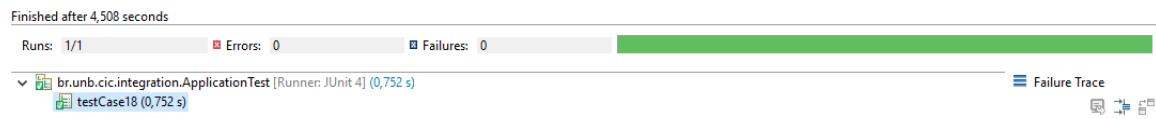


Figura 4.19: Resultado do caso de teste 18.

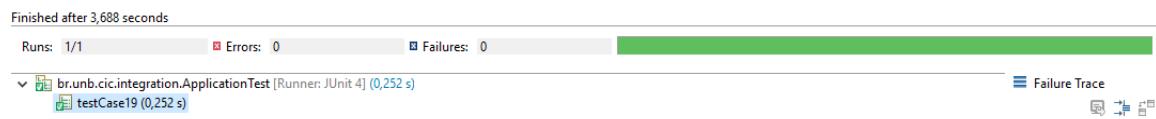


Figura 4.20: Resultado do caso de teste 19.

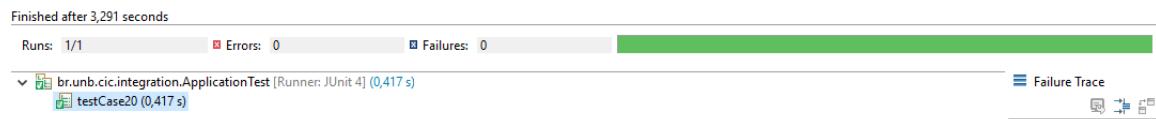


Figura 4.21: Resultado do caso de teste 20.

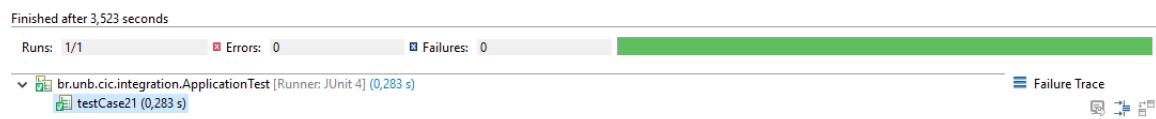


Figura 4.22: Resultado do caso de teste 21.

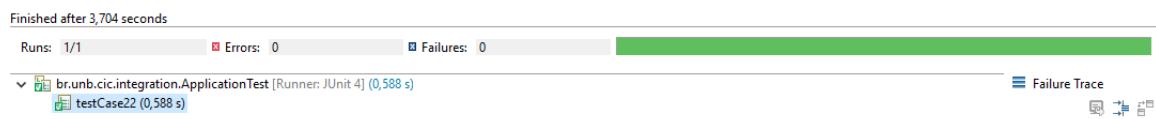


Figura 4.23: Resultado do caso de teste 22.

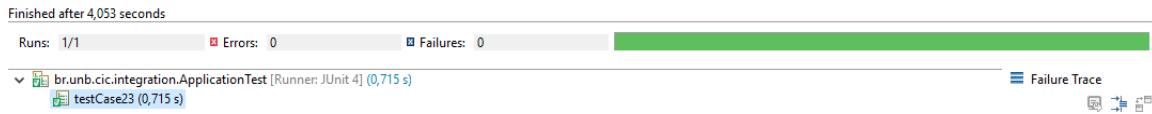


Figura 4.24: Resultado do caso de teste 23.

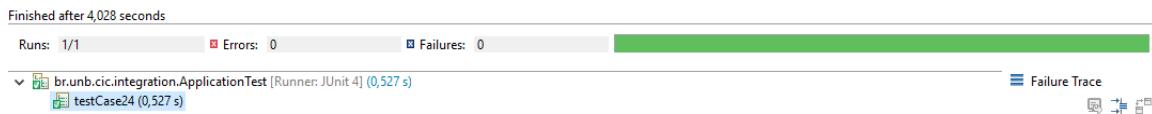


Figura 4.25: Resultado do caso de teste 24.

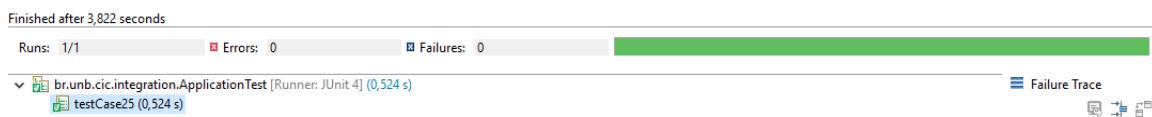


Figura 4.26: Resultado do caso de teste 25.

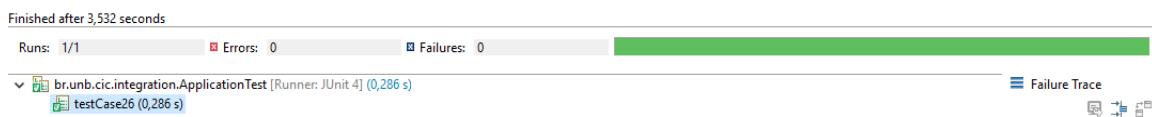


Figura 4.27: Resultado do caso de teste 26.

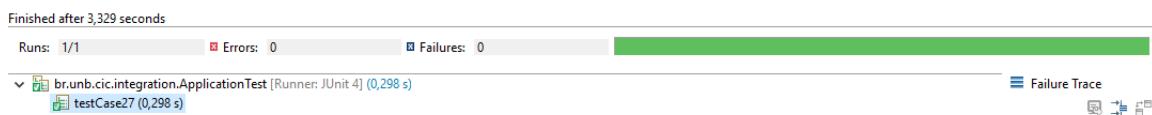


Figura 4.28: Resultado do caso de teste 27.

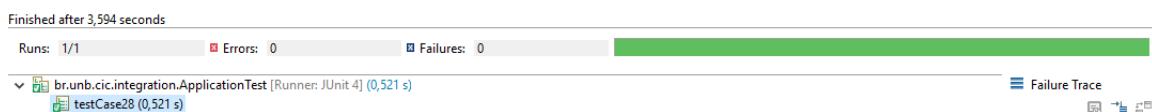


Figura 4.29: Resultado do caso de teste 28.

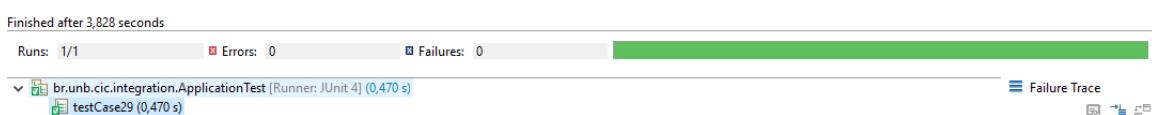


Figura 4.30: Resultado do caso de teste 29.

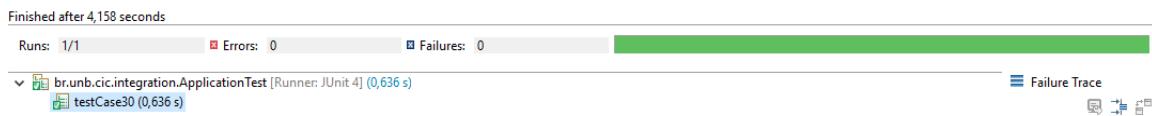


Figura 4.31: Resultado do caso de teste 30.



Figura 4.32: Resultado do caso de teste 31.



Figura 4.33: Resultado do caso de teste 32.

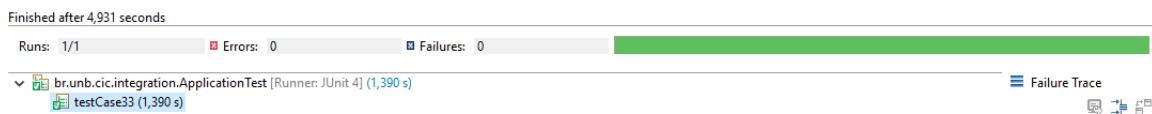


Figura 4.34: Resultado do caso de teste 33.

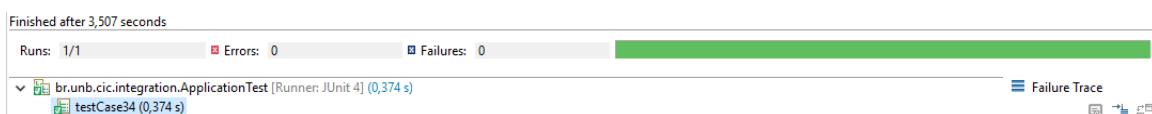


Figura 4.35: Resultado do caso de teste 34.

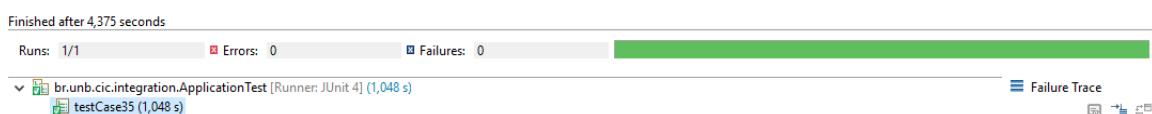


Figura 4.36: Resultado do caso de teste 35.

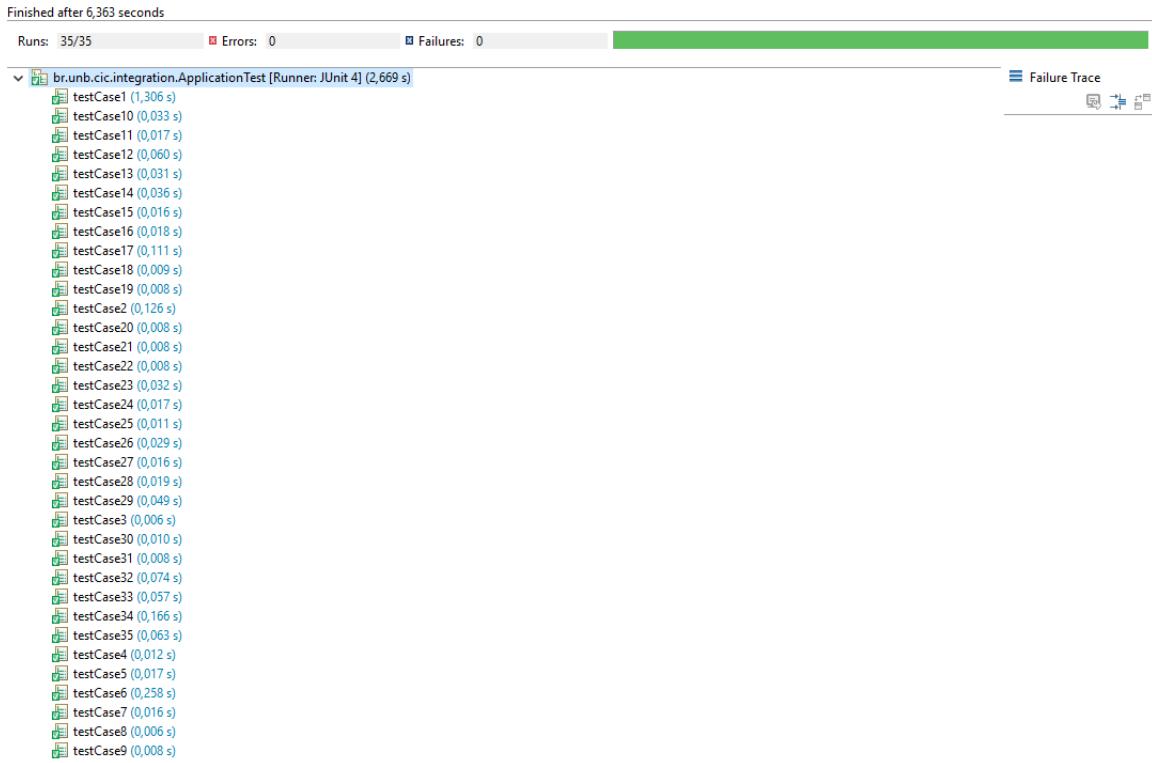


Figura 4.37: Sumário dos resultados dos testes.

É importante salientar que os incrementos do Analisador de Corretude aqui proposto não se restringem aos casos de teste executados e relatados acima. Dentre as regras de corretude definidas, 8 não foram exercitadas pelos testes de Solano *et al.* (2016). Por esse motivo, foram implementados 8 testes adicionais para validar o comportamento adequado do GODA em cada um desses casos.

A implementação desses casos de testes foi acrescida aos 35 testes funcionais pré-existentes no código-fonte, e exercitam as seguintes regras de corretude:

- **1 (b):** Objetivos não podem se refinar em sub-objetivo e tarefa simultaneamente.
- **1 (c):** Objetivos não podem se refinar através de refinamento AND e através de refinamento OR ao mesmo tempo.
- **1 (e):** O número de nós na anotação de tempo de execução de um objetivo deve ser igual à quantidade de nós filhos dele.
- **1 (g):** Objetivos não podem se refinar em mais de uma tarefa.
- **1 (j):** Caso a label de um objetivo tenha um colchete "[" (para começar a escrita da anotação de tempo de execução), é necessário que seu par "]" exista.

- **1 (k):** Caso a label de um objetivo tenha um colchete "](para terminar a escrita da anotação de tempo de execução), é necessário que seu par "[" exista.
- **2 (e):** Tarefas não podem se refinar através de refinamento AND e através de refinamento OR ao mesmo tempo.
- **2 (g):** A quantidade de nós filhos da Tarefa deve ser igual ao número de nós presente na anotação de tempo de execução.

A execução dos testes revelou a obtenção de sucesso em todos os 8 casos de teste adicionais implementados. A imagem a seguir ilustra o retorno do JUnit após a execução (Figura 4.38). Os diagramas de cada um desses testes está disponível em Anexo II.

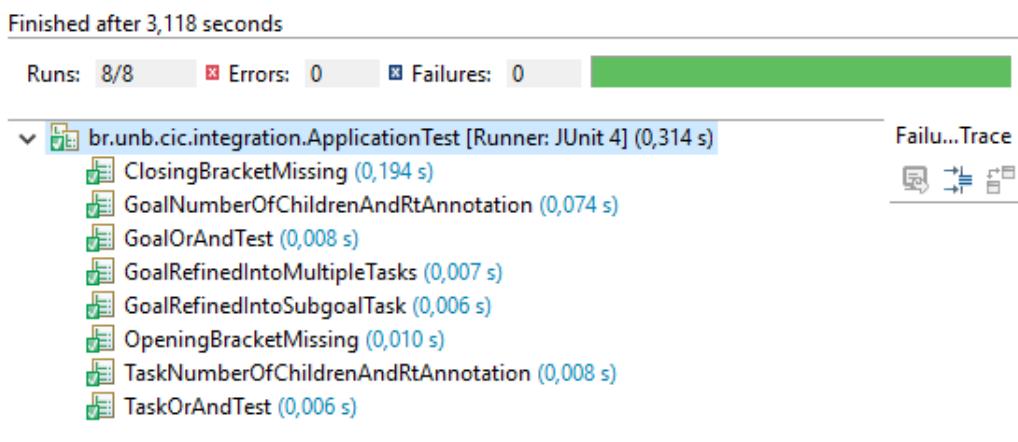


Figura 4.38: Resultados dos casos de teste adicionais via JUnit.

Dessa forma, a Tabela 4.3 a seguir ilustra os resultados obtidos nos 8 novos casos de teste.

Testes Adicionais	Sucesso	Falha
Teste Regra 1b	✓	-
Teste Regra 1c	✓	-
Teste Regra 1e	✓	-
Teste Regra 1g	✓	-
Teste Regra 1j	✓	-
Teste Regra 1k	✓	-
Teste Regra 2c	✓	-
Teste Regra 2g	✓	-

Tabela 4.3: Resultados dos casos de teste adicionais.

4.1 Discussão

Os resultados obtidos expressam sucesso no comportamento do framework GODA em relação aos casos de teste funcionais realizados, e nesta Seção será discutida a importância do ANTLR e do analisador de corretude do modelo de objetivos do GODA para que esse resultado fosse alcançado.

Conforme a Tabela 4.2, o analisador de corretude proposto neste projeto foi responsável por vinte e uma (21) correções em relação à identificação de estados inválidos, no que diz respeito às anotações do CRGM e às declarações de variáveis instanciadas no modelo. Dessa maneira, é possível atribuir à solução proposta neste trabalho a responsabilidade pelo sucesso de 60% dos casos de teste.

Apesar da ferramenta proposta alcançar tamanha melhoria (mais da metade dos testes), o alcance de 100% de sucesso deve ser atribuído também ao ANTLR (Seção 2.10), implementado na versão original do GODA por *Mendonça et al.*, e às melhorias realizadas no processo de integração entre o piStar e o GODA [4].

Como visto, o ANTLR gera um *parser* a partir de uma gramática, e assegura a corretude sintática das entradas textuais. A versão original do GODA foi proposta já munida de uma gramática e do ANTLR [7], e os resultados dos mesmos casos de teste naquela versão foram descritos por Solano *et al.*. A utilização do ANTLR foi capaz de levar o GODA a se comportar como esperado em sete (7) casos, que representa 20% dos casos de teste. Contudo, as análises realizadas pelo ANTLR não englobam os casos avaliados pelo analisador de corretude proposto neste trabalho, pois sua análise é individual, avaliando a *label* de cada nó em relação à sintaxe.

De maneira análoga, apenas o analisador de corretude do modelo de objetivos também seria insuficiente para um sistema complexo como o GODA, pois a análise realizada por esta ferramenta se pautou em uma análise conjunta entre os elementos modelados. As avaliações realizadas buscaram verificar a correlação entre as anotações e a modelagem, levando em consideração a corretude em relação a utilização das variáveis instanciadas no modelo.

Da mesma forma que os casos analisados pela gramática e pelo ANTLR não englobam os casos do analisador de corretude do modelo de objetivos, a recíproca também vale. As ferramentas se complementam e proporcionam ao framework GODA maior confiança nas transformações inerentes ao processo do GODA.

Dessa forma, podemos constatar que a existência mútua de um analisador de corretude do modelo de objetivos do GODA como o proposto neste trabalho e o ANTLR proporcionou o sucesso do framework em relação aos testes funcionais realizados. Ambos, juntos, foram responsáveis pela satisfação de 80% dos casos teste a que o GODA foi submetido. O restante dos casos está atrelado ao processo de integração entre o GODA e o piStar

[4], onde algumas melhorias em relação às anotações de contexto foram realizadas, que representam o restante dos 20% dos casos de teste.

4.2 Histórico de Resultados do GODA

Com os resultados obtidos, é possível gerar um comparativo em relação aos resultados de versões anteriores do GODA e criar uma *timeline* do grau de sucesso em relação às atualizações que o framework recebeu.

Esse comparativo foi devidamente representado na Tabela 4.4. Nela estão representadas as três versões do GODA, indicando em quais casos de teste cada versão obteve sucesso.

1. GODA Original [7], [3]: versão proposta por Mendonça *et al.*, em 2015. Foi lançada já com uma gramática estabelecida e com o ANTLR para realizar a verificação sintática.
2. piStarGODA [4]: versão proposta por Bergmann *et al.*, em 2018. Promoveu a integração do piStar com o GODA, e incrementou a verificação de alguns casos, principalmente relacionados às anotações de contexto.
3. piStarGODA com Analisador de Corretude: versão proposta neste trabalho visando incrementar a última versão do GODA (até a realização deste), que é a versão piStarGODA, com um Analisador de Corretude.

Casos de Teste	GODA Original (2015)	piStarGODA (2018)	piStarGODA com Analisador de Corretude (2019)
1, 2	-	-	1, 2
3, 4, 5, 6	-	3, 6	3, 4, 5, 6
7, 8	-	-	7, 8
9, 10	-	9	9, 10
11, 12, 13, 14	-	-	11, 12, 13, 14
15, 16	-	-	15, 16
17, 18	-	-	17, 18
19, 20	-	-	19, 20
21, 22	-	-	21, 22
23, 24	-	-	23, 24
25, 26, 27, 28, 29, 30, 31	25, 26, 27, 28, 29, 30, 31	25, 26, 27, 28, 29, 30, 31	25, 26, 27, 28, 29, 30, 31
32, 33	-	32, 33	32, 33
34, 35	-	34, 35	34, 35

Tabela 4.4: Versões do GODA e seus desempenhos nos testes funcionais.

Os resultados expostos na Tabela 4.4 indicam em quais casos de teste cada versão do GODA obteve sucesso. Note que os casos em que a primeira versão obteve sucesso estão com a cor padrão. Já os casos que passaram devido às novidades do piStarGODA estão representados na cor azul. Por fim, os casos satisfeitos através do Analisador de Corretude do Modelo de Objetivos do GODA proposto neste trabalho estão representados com a coloração laranja.

Com o acréscimo do Analisador de Corretude ao piStarGoda, portanto, o framework obteve sucesso em todos os casos de teste, graças à complementação entre as ferramentas distribuídas nessas versões no decorrer dos anos.

Como sugestão de trabalhos futuros, pode-se elaborar outros testes funcionais para continuar a avaliar a confiabilidade do GODA em relação à corretude do modelo CRGM, para que todo potencial do Analisador de Corretude do Modelo de Objetivos do GODA seja analisado, e para sugerir atualizações nesta ferramenta, caso se faça necessário.

4.3 Lições Aprendidas e Dificuldades Encontradas

Uma das principais lições aprendidas durante o desenvolvimento deste trabalho foi no que diz respeito à importância de verificação da corretude de modelos orientados a objetivos. Erros simples, até mesmo de digitação, podem ser responsáveis pela má formação de um modelo orientado a objetivo e pode vir a causar inconsistências no levantamento de requisitos e/ou na elaboração de um sistema de software modelado a partir desse conceito.

Outra lição aprendida através da realização deste projeto foi em relação à forma como abordar uma manutenção de um código, no sentido de elaborar uma maneira de estudar e entender um código idealizado e implementado por outras pessoas. Além disso, nessas linhas também pode-se ressaltar o aprendizado relativo à linguagem Java (e por consequência da IDE Eclipse), principalmente em relação aos recursos e estruturas que a linguagem oferece e que foram determinantes para obter caminhos eficazes que viabilizaram soluções mais sucintas.

A maior dificuldade encontrada durante o processo de realização deste trabalho foi a compreensão do código complexo do GODA, principalmente no que se refere à identificação das estruturas que armazenam as informações do modelo de objetivos.

A falta de familiaridade com sistemas desse porte fez com que a curva de aprendizado em relação ao projeto fosse relativamente lenta. No entanto, apesar desses empecilhos, foi possível alcançar êxito na realização deste trabalho, muito devido à abordagem sugerida utilizada.

Capítulo 5

Conclusão

O objetivo deste trabalho foi prover uma ferramenta de verificação, aqui denominada analisador de corretude do modelo de objetivos, para garantir que o modelo CRGM fosse verificado e que os estados inválidos fossem devidamente reconhecidos para que não houvesse propagação de erro para a análise de dependabilidade, assegurando uma maior confiabilidade ao GODA no desempenho de sua função.

A fim de atender a esse objetivo, uma sequência de atividades foi adotada para garantir a organização do processo de desenvolvimento. Assim, primeiramente foi realizado uma coleta das discrepâncias do GODA em relação à identificação de estados inválidos, contando com a devida ajuda dos trabalhos anteriores [5] e [4]. Em seguida, foi necessário estimar as informações que seriam necessárias para realizar o correto tratamento de cada caso levantado (exposto na Seção 3.1), para que pudesse ser dado início ao processo de rastreio dessas informações no código e eventual uso na implementação propriamente dita.

Os resultados obtidos foram bastante satisfatórios no sentido de que a ferramenta proporcionou ao framework GODA a obtenção de êxito na realização de todos os casos de teste desenvolvidos em [5], sem nenhuma falha encontrada. Para fins comparativos, a última aplicação de testes no GODA relatou vinte (21) situações em que o sistema não se comportou como esperado [4], utilizando os mesmos testes funcionais. A implementação foi realizada utilizando a linguagem de programação Java e a IDE Eclipse.

Esses resultados são bastante importantes para viabilizar uma maior utilização do framework GODA, uma vez que o sistema está agora munido de uma ferramenta que proporciona maior consistência e confiabilidade na geração do modelo CRGM, e por conseguinte na geração automática do modelo DTMC e na análise de dependabilidade.

As limitações deste trabalho estão pautadas no fato de que embora a chance de erros de nomenclatura e/ou sintáticos ocorrerem tenha sido显著mente reduzida, não é possível afirmar que todas possibilidades de erros desses tipos foram tratadas. Como visto, os casos tratados foram identificados através da utilização do GODA e dos levantados no

estudo em [5], e foram devidamente descritos na Seção 3.1. No entanto, pode ser que mais erros (não detectados durante a realização deste projeto) sejam identificados futuramente.

Isso permite concluir que a implementação dessa ferramenta foi apenas o passo inicial da consistência ao GODA, e precisa ser mantido atualizado conforme novos erros de nomenclatura e/ou de formação do modelo sejam descobertos, para que a corretude dos modelos seja assegurada e a análise de dependabilidade do GODA seja a mais precisa possível.

Referências

- [1] Giorgini, Paolo: *Tropos: basics.* <http://www.troposproject.eu/files/8-Tropos-Basics.pdf>, 2009. ix, 6, 7
- [2] Marta Kwiatkowska, Gethin Norman e David Parker: *Probabilistic model checking - part 2 - discrete time markov chains.* lecture in University of Oxford, <http://www.prismmodelchecker.org/lectures/biss07/02-dtmcs.pdf>. ix, 9, 11, 12
- [3] Danilo Filgueira Mendonça, Genaína Nunes Rodrigues, Vander Alves Raian Ali e Luciano Baresi: *Goda: A goal-oriented requirements engineering framework for runtime dependability analysis.* 80, 2016. ix, x, xiii, 1, 13, 14, 15, 16, 17, 37, 38, 39, 52
- [4] Bergmann, Leandro Santos: *pistar-goda: Integração entre os projetos pistar e goda.* Monografia apresentada como requisito parcial para conclusão do Bacharelado em Ciência da Computação, UnB, 2018. x, xiii, 3, 37, 41, 42, 51, 52, 54
- [5] Solano, Gabriela Félix: *Verificando a corretude da transformação de modelos no goda.* Monografia apresentada como requisito parcial para conclusão do Bacharelado em Ciência da Computação, UnB, 2016. x, xiii, 3, 19, 41, 42, 43, 54, 55, 58
- [6] Fabiano Dalpiaz, Alexander Borgida, Jennifer Horkoff e John Mylopoulos: *Runtime goal models.* 2013. xiii, 8, 9, 15
- [7] Mendonça, Danilo F.: *Dependability verification for contextual/runtime goal modelling.* Dissertação apresentada como requisito parcial para conclusão do Mestrado em Informática, UnB, 2015. 1, 8, 12, 13, 17, 51, 52
- [8] Lamsweerde, Axel van: *Goal-oriented requirements engineering: A guided tour.* Proceedings Fifth IEEE International Symposium on Requirements Engineering, 2001. 1, 5
- [9] Raian Ali, Fabiano Dalpiaz e Paolo Giorgini: *Contextual goal models.* 2010. 1, 8
- [10] Algirdas Avizienis, Jean Claude Laprie Brian Randell e Carl Landwehr: *Basic concepts and taxonomy of dependable and secure computing.* IEEE Transactions on Dependable and Secure Computing, 2004. 1
- [11] Paolo Giorgini, John Mylopoulos e Roberto Sebastiani: *Goal-oriented requirements analysis and reasoning in the tropos methodology.* 18, 2005. 5, 6

- [12] Mylopoulos, John, Lawrence Chung e Eric Yu: *From object-oriented to goal-oriented requirements analysis*. Commun. ACM, 42(1):31–37, janeiro 1999, ISSN 0001-0782. 6
- [13] Clarke A. B., DISNEY R. L.: *Probabilidade e Processos Estocásticos*. Livros Técnicos e Científicos Editora, 1979. 10
- [14] Glynn, Peter W.: *Stochastic systems*. <https://stanford.edu/class/msande321/Handouts/02%20Discrete%20Time%20Markov%20Chains.pdf>, 2013. 10
- [15] Nogueira, Fernando: *Cadeias de markov*. Notas de aula, Universidade Federal de Juiz de Fora. <http://www.ufjf.br/epd042/files/2009/02/cadeiaMarkov1.pdf>, 2009. 10
- [16] Cassandras, C. e S. Lafortune: *Introduction to Discrete Event Systems*. 2008, ISBN 978-0-387-33332-8. 11
- [17] Marta Kwiatkowska, Gethin Norman e David Parker: *Prism: Probabilistic model checking for performance and reliability analysis*. ACM SIGMETRICS Performance Evaluation Review, 36(4), pages 40-45, ACM., 2009. 11, 12
- [18] E. M. Clarke, Orna Grumberg, Doron Peled: *Model Checking*. The MIT Press, 1999. 12
- [19] Grunske, Lars: *Specification patterns for probabilistic quality properties*. 2008. 13
- [20] Hansson, H. e B. Jonsson.: *A logic for reasoning about time and reliability*. Formal Aspects of Computing, 6(5):512-535, 1994. 14
- [21] *About the antlr parser generator*. <https://www.antlr.org/about.html>. 17
- [22] Pierce, Benjamin C.: *Types and Programming Languages*. 2002, ISBN 0-262-16209-1.
- [23] Tomassetti, Gabriele: *The antlr mega tutorial*. <https://tomassetti.me/antlr-mega-tutorial/>, 2017.

Anexo I

Testes Funcionais

Os testes funcionais descritos a seguir foram propostos e desenvolvidos por *Solano et al.* (2016) [5].

1.
 - Entrada: modelo correto (Figura I.1).
 - Saída Esperada: Execução finalizada sem exceções lançadas.

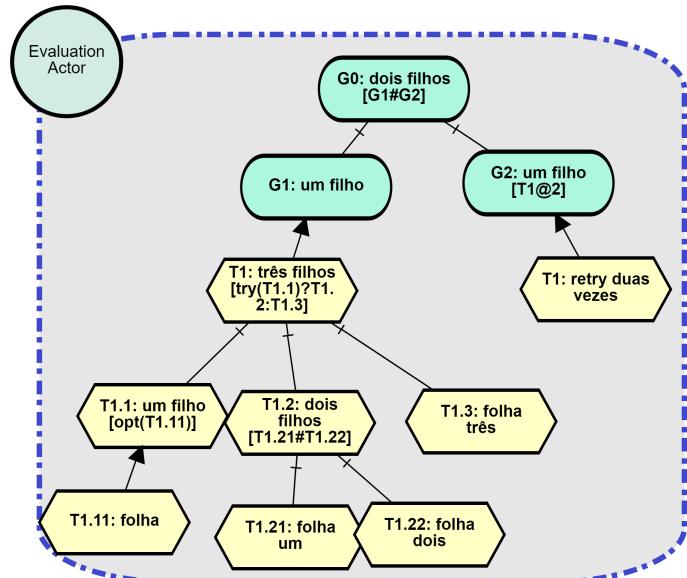


Figura I.1: Ilustração para o caso de teste 1.

2.
 - Entrada: modelo correto (Figura I.2).
 - Saída Esperada: Execução finalizada sem exceções lançadas.
3.
 - Entrada: Objetivo com label fora do padrão (Figura I.3).
 - Saída Esperada: Execução interrompida com exceção lançada.

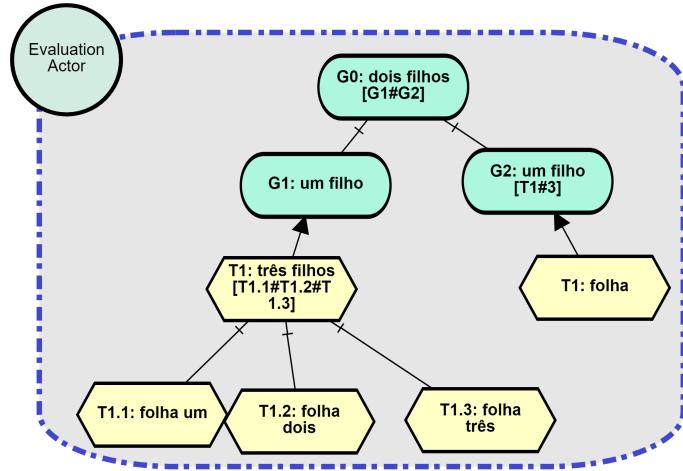


Figura I.2: Ilustração para o caso de teste 2.

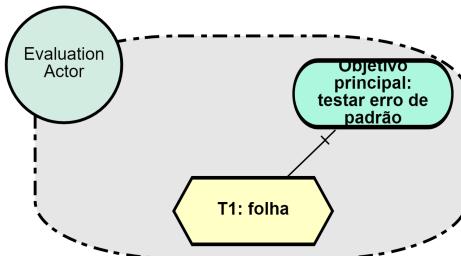


Figura I.3: Ilustração para o caso de teste 3.

4. • Entrada: Tarefa com label fora do padrão (Figura I.4).
 • Saída Esperada: Execução interrompida com exceção lançada.

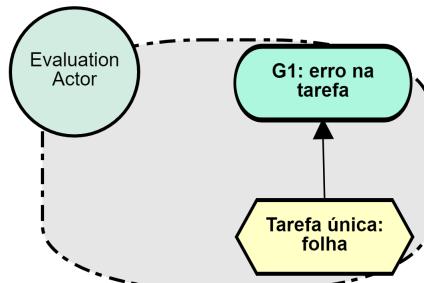


Figura I.4: Ilustração para o caso de teste 4.

5. • Entrada: Tarefa com label fora do padrão (Figura I.5).
 • Saída Esperada: Execução interrompida com exceção lançada.
6. • Entrada: Anotação de contexto com label fora do padrão (Figura I.6).
 • Saída Esperada: Execução interrompida com exceção lançada.

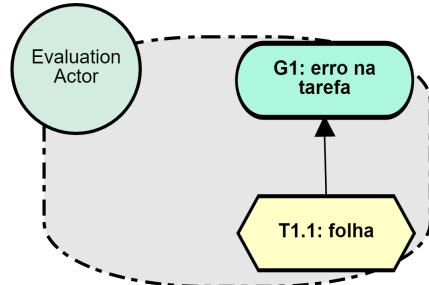


Figura I.5: Ilustração para o caso de teste 5.

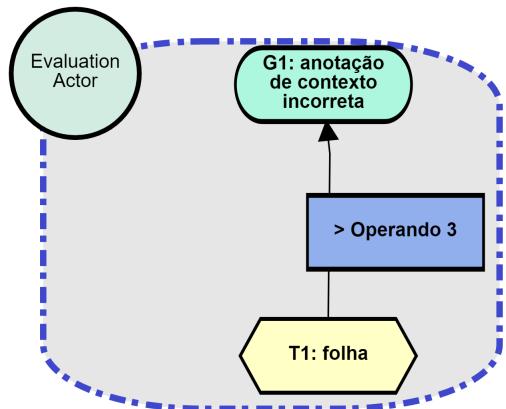


Figura I.6: Ilustração para o caso de teste 6.

7.
 - Entrada: Objetivo sem nó filho e com anotação de tempo de execução (Figura I.7).
 - Saída Esperada: Execução interrompida com exceção lançada.

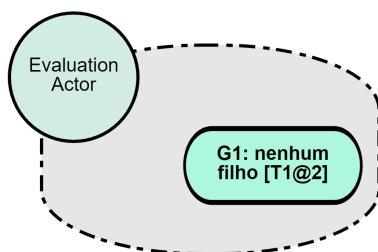


Figura I.7: Ilustração para o caso de teste 7.

8.
 - Entrada: Tarefa sem nó filho e com anotação de tempo de execução (Figura I.8).
 - Saída Esperada: Execução interrompida com exceção lançada.
9.
 - Entrada: Objetivo com um nó filho, apresentando anotação de tempo de execução inválida (Figura I.9).

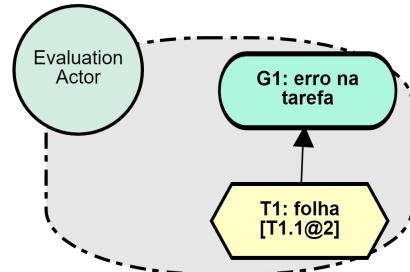


Figura I.8: Ilustração para o caso de teste 8.

- Saída Esperada: Execução interrompida com exceção lançada.

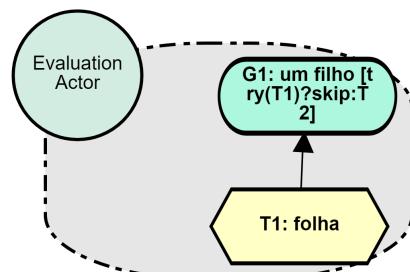


Figura I.9: Ilustração para o caso de teste 9.

- 10.
- Entrada: Tarefa com um nó filho, apresentando anotação de tempo de execução inválida (Figura I.10).
 - Saída Esperada: Execução interrompida com exceção lançada.

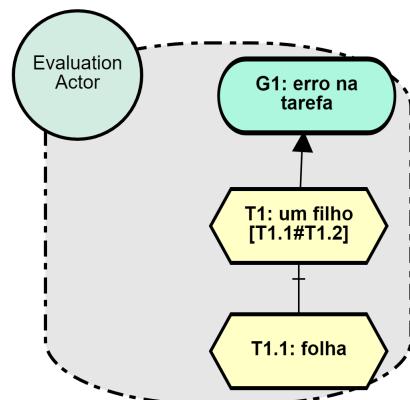


Figura I.10: Ilustração para o caso de teste 10.

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

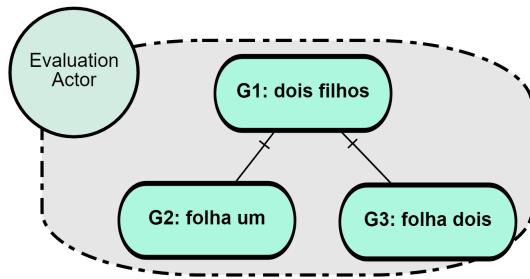


Figura I.11: Ilustração para o caso de teste 11.

12. • Entrada: Objetivo com quatro nós filhos, sem anotação de tempo de execução (Figura I.12).
 • Saída Esperada: Execução interrompida com exceção lançada.

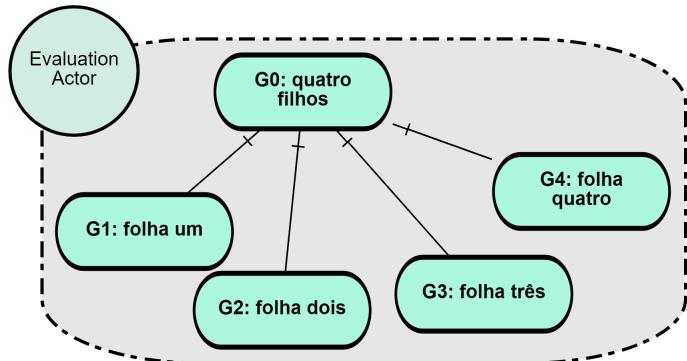


Figura I.12: Ilustração para o caso de teste 12.

13. • Entrada: Tarefa com dois nós filhos, sem anotação de tempo de execução (Figura I.13).
 • Saída Esperada: Execução interrompida com exceção lançada.

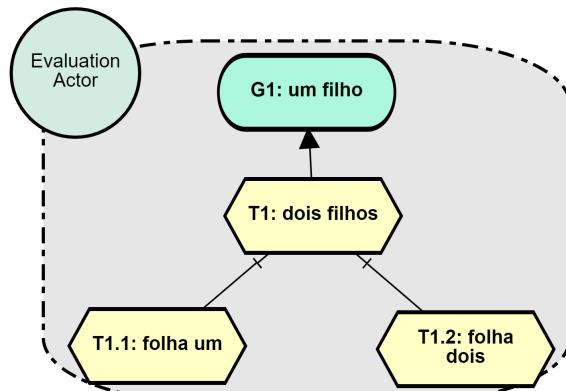


Figura I.13: Ilustração para o caso de teste 13.

14. • Entrada: Tarefa com quatro nós filhos, sem anotação de tempo de execução (Figura I.14).
• Saída Esperada: Execução interrompida com exceção lançada.

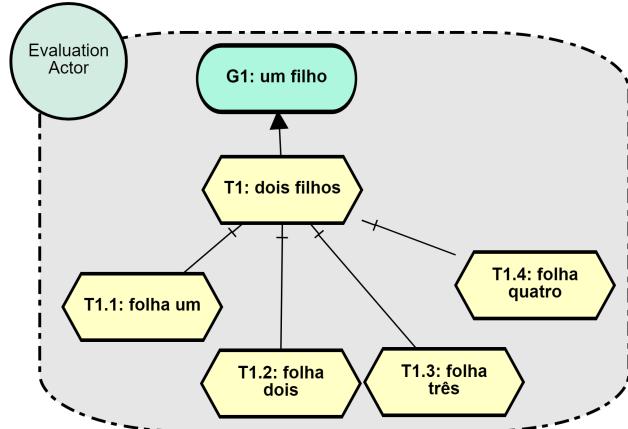


Figura I.14: Ilustração para o caso de teste 14.

15. • Entrada: Dois objetivos tendo o mesmo identificador (Figura I.15).
• Saída Esperada: Execução interrompida com exceção lançada.

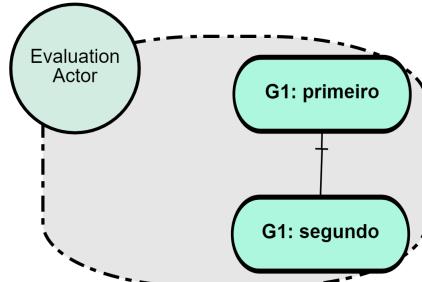


Figura I.15: Ilustração para o caso de teste 15.

16. • Entrada: Quatro objetivos tendo o mesmo identificador (Figura I.16).
• Saída Esperada: Execução interrompida com exceção lançada.
17. • Entrada: Duas tarefas irmãs com profundidade três e o mesmo identificador (Figura I.17).
• Saída Esperada: Execução interrompida com exceção lançada.
18. • Entrada: Três tarefas irmãs com profundidade dois e o mesmo identificador (Figura I.18).
• Saída Esperada: Execução interrompida com exceção lançada.

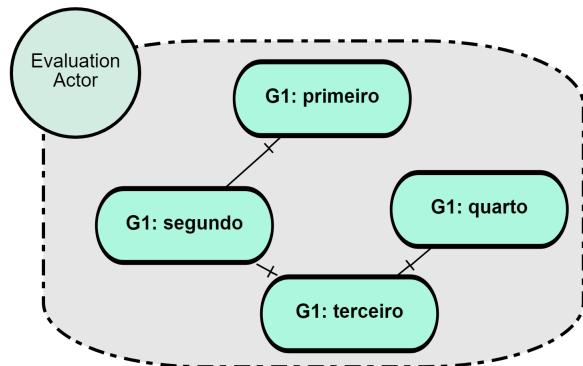


Figura I.16: Ilustração para o caso de teste 16.

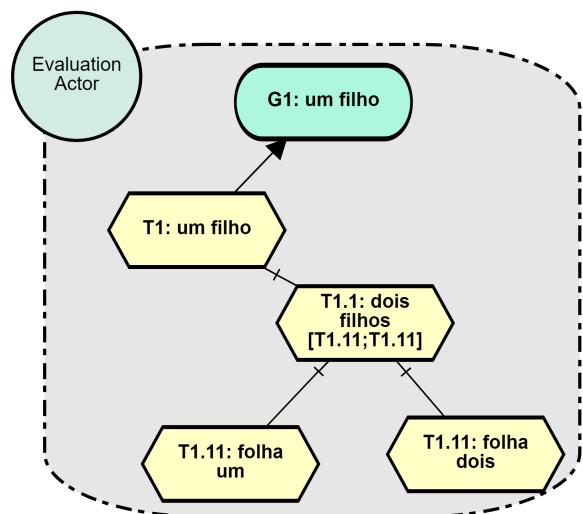


Figura I.17: Ilustração para o caso de teste 17.

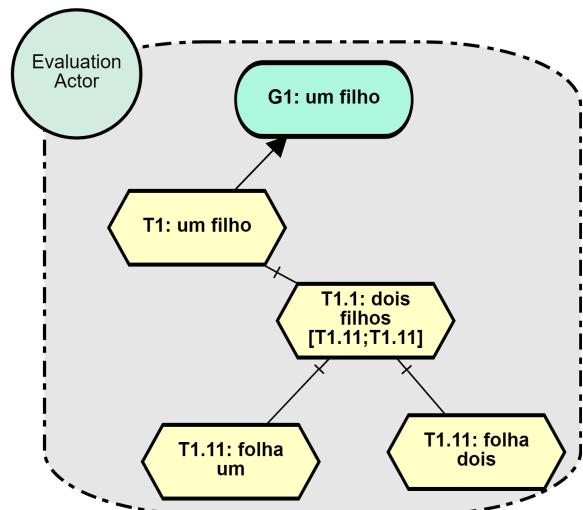


Figura I.18: Ilustração para o caso de teste 18.

19. • Entrada: Tarefa com profundidade dois e identificador fora do padrão (Figura I.19).
• Saída Esperada: Execução interrompida com exceção lançada.

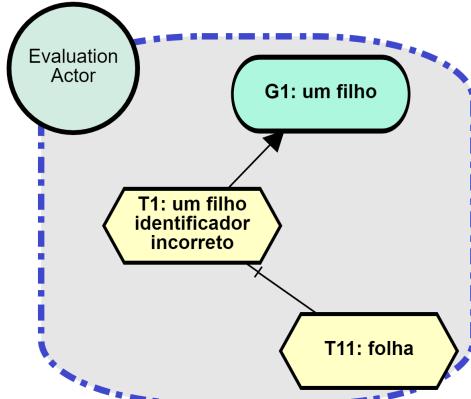


Figura I.19: Ilustração para o caso de teste 19.

20. • Entrada: Tarefa com profundidade três e identificador fora do padrão (Figura I.20).
• Saída Esperada: Execução interrompida com exceção lançada.

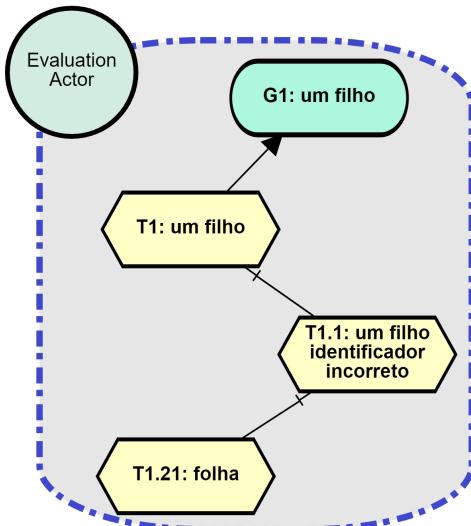


Figura I.20: Ilustração para o caso de teste 20.

21. • Entrada: Anotação de tempo de execução fora de colchetes (Figura I.21).
• Saída Esperada: Execução interrompida com exceção lançada.
22. • Entrada: Anotação de tempo de execução antes da descrição do objetivo ou tarefa (Figura I.22).

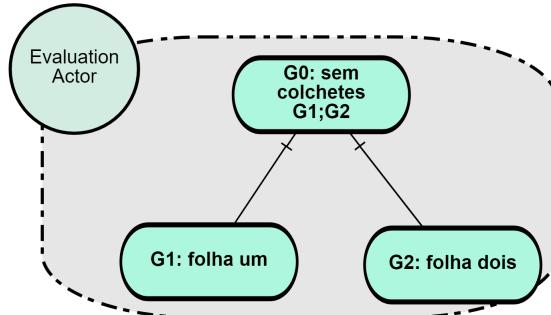


Figura I.21: Ilustração para o caso de teste 21.

- Saída Esperada: Execução interrompida com exceção lançada.

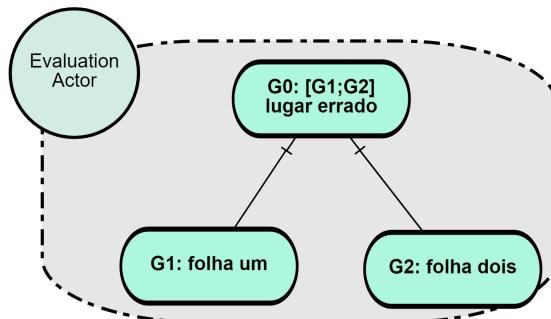


Figura I.22: Ilustração para o caso de teste 22.

23. • Entrada: Anotação de tempo de execução referenciando um nó que não seja filho (Figura I.23).
 • Saída Esperada: Execução interrompida com exceção lançada.

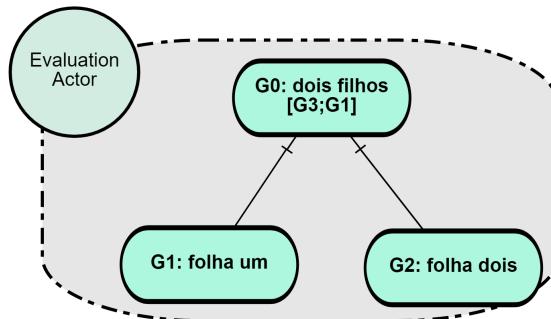


Figura I.23: Ilustração para o caso de teste 23.

24. • Entrada: Anotação de tempo de execução referenciando mais de um nó que não seja filho (Figura I.24).
 • Saída Esperada: Execução interrompida com exceção lançada.

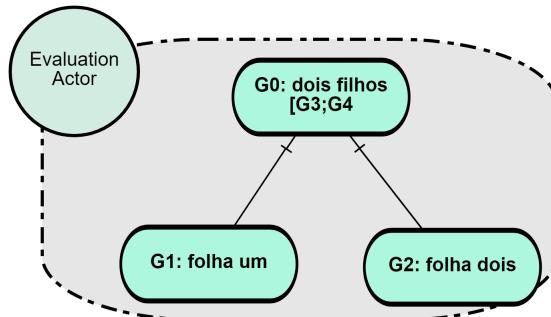


Figura I.24: Ilustração para o caso de teste 24.

25. • Entrada: Expressão “[E1;E2]”, que não segue o padrão (Figura I.25).
• Saída Esperada: Execução interrompida com exceção lançada.

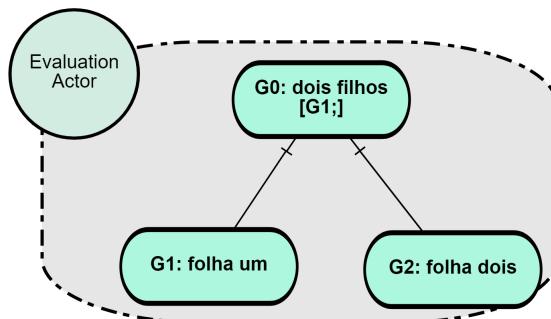


Figura I.25: Ilustração para o caso de teste 25.

26. • Entrada: Expressões “[E1E2]” e “[En]”, que não seguem o padrão (Figura I.26).
• Saída Esperada: Execução interrompida com exceção lançada.

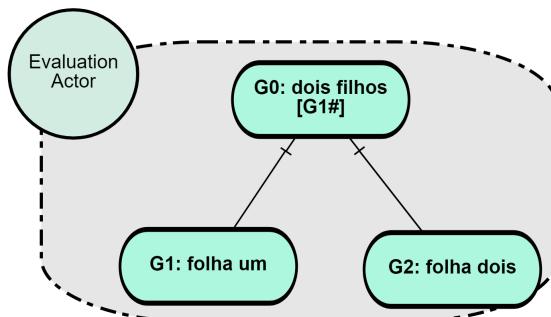


Figura I.26: Ilustração para o caso de teste 26.

27. • Entrada: Expressão “[E+n]”, que não segue o padrão (Figura I.27).
• Saída Esperada: Execução interrompida com exceção lançada.

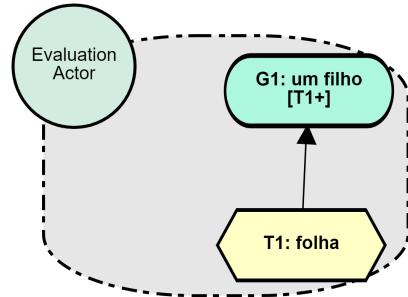


Figura I.27: Ilustração para o caso de teste 27.

28. • Entrada: Expressão “[E@n]”, que não segue o padrão (Figura I.28).
• Saída Esperada: Execução interrompida com exceção lançada.

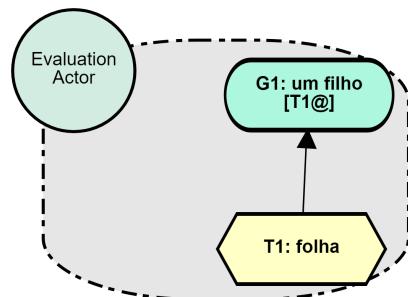


Figura I.28: Ilustração para o caso de teste 28.

29. • Entrada: Expressão “[E1|E2]”, que não segue o padrão (Figura I.29).
• Saída Esperada: Execução interrompida com exceção lançada.

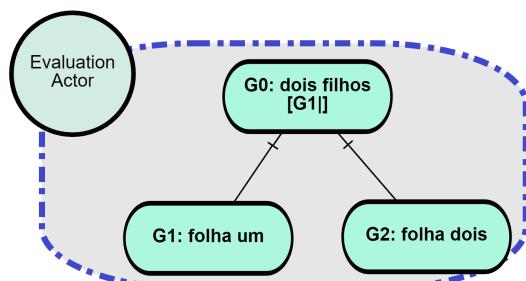


Figura I.29: Ilustração para o caso de teste 29.

30. • Entrada: Expressão “[opt(E)]”, que não segue o padrão (Figura I.30).
• Saída Esperada: Execução interrompida com exceção lançada.
31. • Entrada: Expressão “[try(E)?E1:E2]”, que não segue o padrão (Figura I.31).
• Saída Esperada: Execução interrompida com exceção lançada.

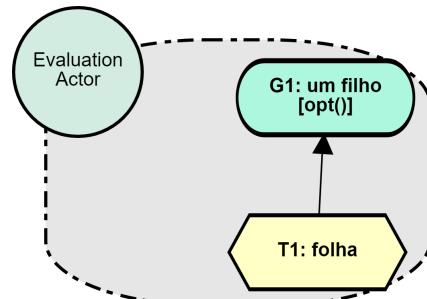


Figura I.30: Ilustração para o caso de teste 30.

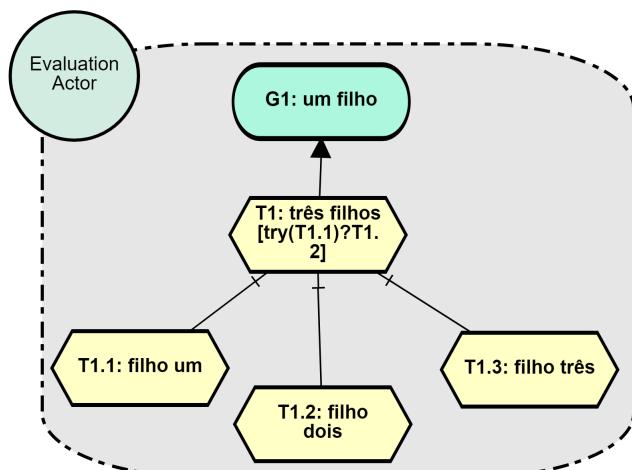


Figura I.31: Ilustração para o caso de teste 31.

32. • Entrada: Operando de anotação de contexto que se inicia com dígito (Figura I.32).
• Saída Esperada: Execução interrompida com exceção lançada.

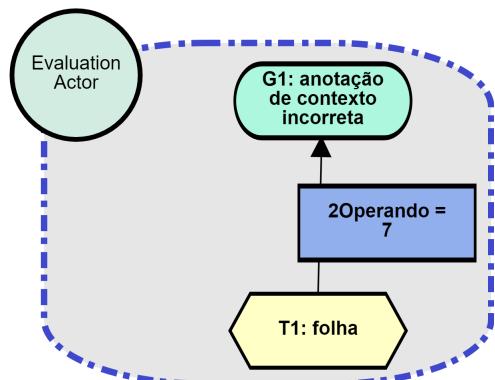


Figura I.32: Ilustração para o caso de teste 32.

33. • Entrada: Operador PRISM, de anotação de contexto, que é inválido (Figura I.33).
 • Saída Esperada: Execução interrompida com exceção lançada.

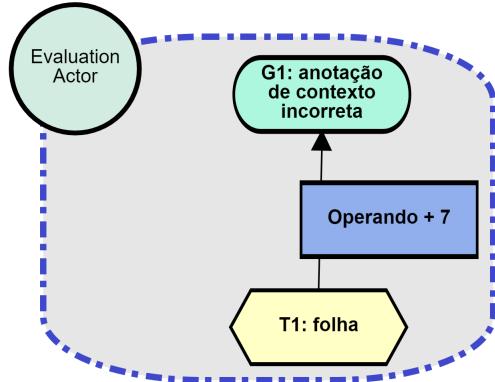


Figura I.33: Ilustração para o caso de teste 33.

34. • Entrada: Anotação de contexto com valor de comparação do tipo string (Figura I.34).
 • Saída Esperada: Execução interrompida com exceção lançada.

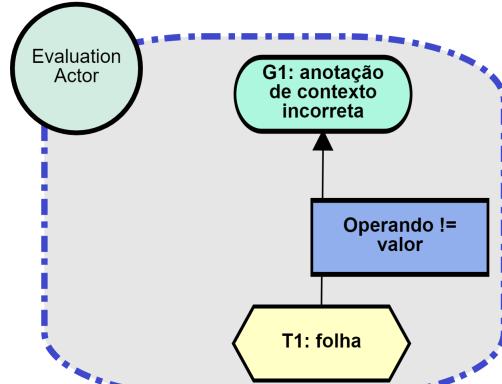


Figura I.34: Ilustração para o caso de teste 34.

35. • Entrada: Anotação de contexto contendo valor de comparação do tipo char (Figura I.35).
 • Saída Esperada: Execução interrompida com exceção lançada.

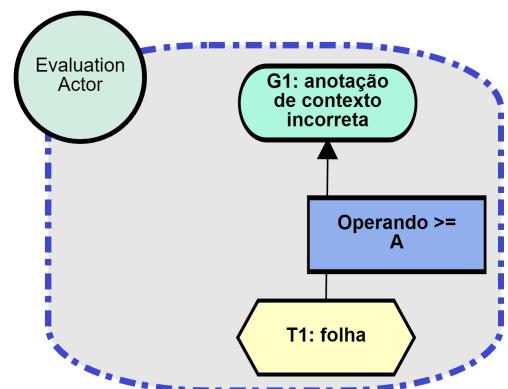


Figura I.35: Ilustração para o caso de teste 35.

Anexo II

Testes Adicionais

Os diagramas a seguir ilustram os testes funcionais que foram implementados neste trabalho e visam complementar os 35 casos de teste propostos por Solano *et al.* (2016), de forma a contemplar as novas regras de corretude definidas e não exercitadas por aqueles.

1.
 - Entrada: O objetivo G0 é refinado em um sub-objetivo e em uma sub-tarefa ao mesmo tempo. (Figura II.1).
 - Saída Esperada: Execução interrompida com exceção lançada.

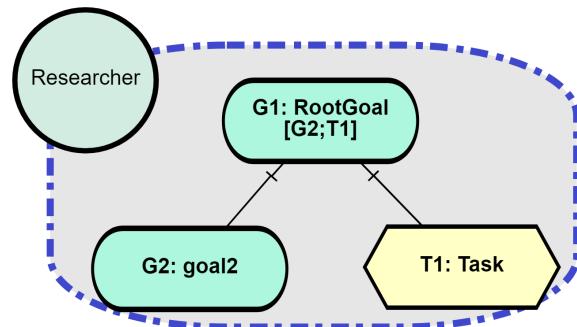


Figura II.1: Ilustração para o caso de teste da regra de corretude 1b.

2.
 - Entrada: O objetivo G0 é refinado por refinamento AND e refinamento OR de forma simultânea. (Figura II.2).
 - Saída Esperada: Execução interrompida com exceção lançada.
3.
 - Entrada: O número de nós na anotação de tempo de execução do objetivo G0 é diferente da quantidade de nós filhos dele. (Figura II.3).
 - Saída Esperada: Execução interrompida com exceção lançada.
4.
 - Entrada: O objetivo G1 é refinado em mais de uma task. (Figura II.4).
 - Saída Esperada: Execução interrompida com exceção lançada.

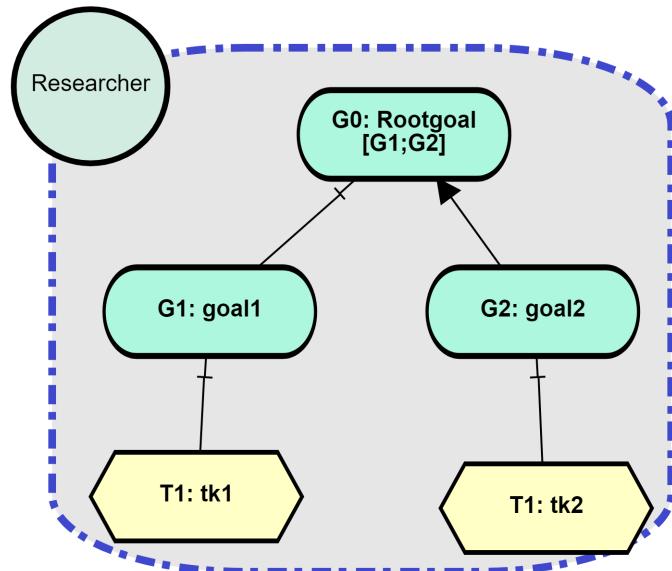


Figura II.2: Ilustração para o caso de teste da regra de corretude 1c.

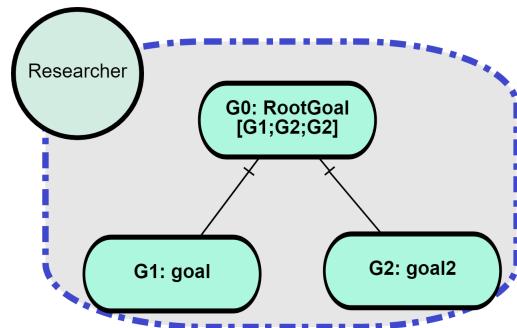


Figura II.3: Ilustração para o caso de teste da regra de corretude 1e.

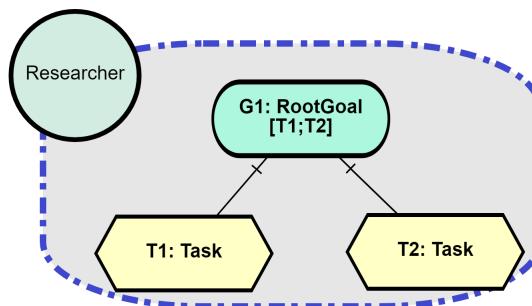


Figura II.4: Ilustração para o caso de teste da regra de corretude 1g.

5.
 - Entrada: Existe um colchete "[" para começar a escrita da anotação de tempo de execução, mas seu par "]" não existe. (Figura II.5).
 - Saída Esperada: Execução interrompida com exceção lançada.

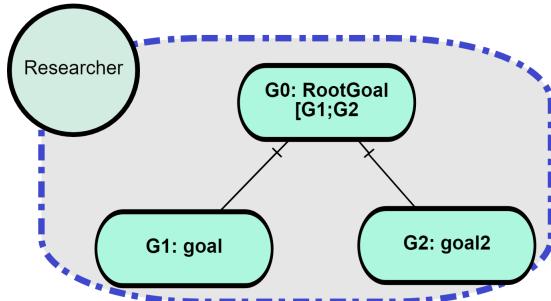


Figura II.5: Ilustração para o caso de teste da regra de corretude 1j.

6.
 - Entrada: Existe um colchete "]" para terminar a escrita da anotação de tempo de execução, mas seu par "[" não existe. (Figura II.6).
 - Saída Esperada: Execução interrompida com exceção lançada.

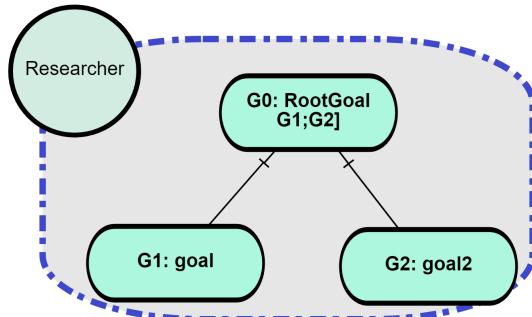


Figura II.6: Ilustração para o caso de teste da regra de corretude 1k.

7.
 - Entrada: A tarefa T1 é refinada por refinamento AND e refinamento OR de forma simultânea. (Figura II.7).
 - Saída Esperada: Execução interrompida com exceção lançada.
8.
 - Entrada: O número de nós na anotação de tempo de execução da tarefa T1 é diferente da quantidade de nós filhos dela. (Figura II.8).
 - Saída Esperada: Execução interrompida com exceção lançada.

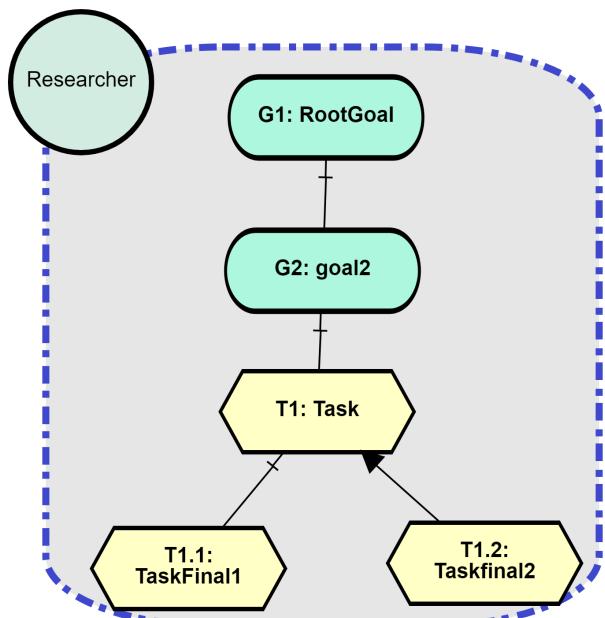


Figura II.7: Ilustração para o caso de teste da regra de corretude 2e.

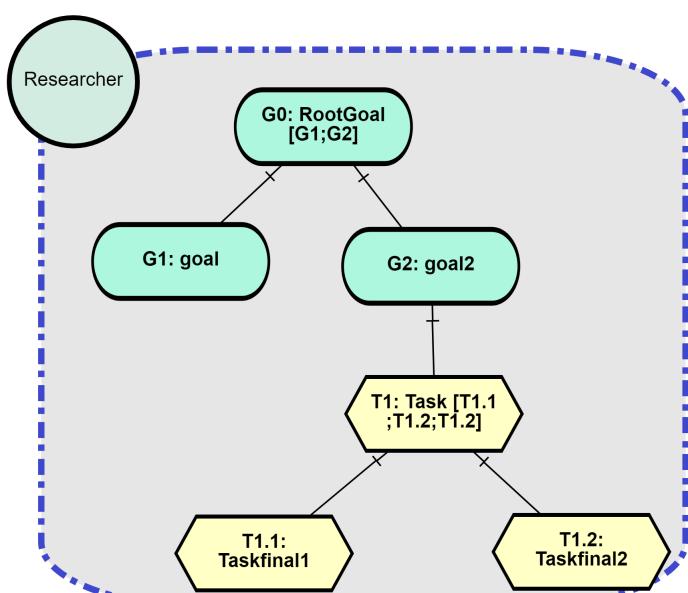


Figura II.8: Ilustração para o caso de teste da regra de corretude 2g.