sid.inpe.br/mtc-m21c/2020/06.08.20.44-TDI

SINGULARITY: UM MÉTODO PARA GERAÇÃO AUTOMÁTICA DE CASOS DE TESTES UNITÁRIOS BASEADO EM CONTRAEXEMPLOS DE VERIFICADOR DE MODELOS PARA APLICAÇÕES EM C++

Eduardo Rohde Eras

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Valdivino Alexandre de Santiago Júnior, aprovada em 26 de maio de 2020.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34R/42L2BFE>

INPE
São José dos Campos
2020

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GBDIR)
Serviço de Informação e Documentação (SESID)
CEP 12.227-010
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/7348
E-mail: pubtc@inpe.br

sid.inpe.br/mtc-m21c/2020/06.08.20.44-TDI

SINGULARITY: UM MÉTODO PARA GERAÇÃO AUTOMÁTICA DE CASOS DE TESTES UNITÁRIOS BASEADO EM CONTRAEXEMPLOS DE VERIFICADOR DE MODELOS PARA APLICAÇÕES EM C++

Eduardo Rohde Eras

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Valdivino Alexandre de Santiago Júnior, aprovada em 26 de maio de 2020.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34R/42L2BFE>

INPE
São José dos Campos
2020

Dados Internacionais de Catalogação na Publicação (CIP)

Eras, Eduardo Rohde.

Er14s Singularity: um método para geração automática de casos de testes unitários baseado em contraexemplos de verificador de modelos para aplicações em C++ / Eduardo Rohde Eras. – São José dos Campos : INPE, 2020.

xxii + 99 p. ; (sid.inpe.br/mtc-m21c/2020/06.08.20.44-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2020.

Orientador : Dr. Valdivino Alexandre de Santiago Júnior.

1. Casos de Teste de Software. 2. Teste Unitário. 3. Verificador de Modelo. 4. Automatização de Teste. 5. Aplicações em C++. I.Título.

CDU 004.415.53

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

FOLHA DE APROVAÇÃO

A FOLHA DE APROVAÇÃO SERÁ INCLUIDA APÓS RESTABELECIMENTO DAS ATIVIDADES PRESENCIAIS.

Por conta da Pandemia do COVID-19, as defesas de Teses e Dissertações são realizadas por vídeo conferência, o que vem acarretando um atraso no recebimento nas folhas de aprovação.

Este trabalho foi aprovado pela Banca e possui as declarações dos orientadores (confirmando as inclusões sugeridas pela Banca) e da Biblioteca (confirmando as correções de normalização).

Assim que a Biblioteca receber a Folha de aprovação assinada, esta folha será substituída.

Qualquer dúvida, entrar em contato pelo email: pubtc@inpe.br.

Divisão de Biblioteca (DIBIB).

"É preciso ser alguma coisa para parecer alguma coisa".

Ludwig van Beethoven
(1770 - 1827)

A meu falecido tio ***Bernardo Rohde*** *quem me ensinou que estudar é o único caminho.*

AGRADECIMENTOS

Agradeço meu orientador, Dr. Valdivino Alexandre de Santiago Júnior, pela oportunidade de fazer o curso de Mestrado em Computação Aplicada no Instituto Nacional de Pesquisas Espaciais e ao CNPq pelo suporte financeiro que tornou possível essa conquista. Agradeço a todos que me apoiaram em meus estudos nessa última década, em especial à Dra. Luciana Brasil Rebelo dos Santos por seu imensurável suporte em todos os momentos, mais que uma professora e uma orientadora, uma verdadeira amiga. Aos alunos da Camila, Leone e Juliana por compartilharem dessa caminhada na Computação Aplicada. Aos professores da ETEC de São José dos Campos, FATEC de São José dos Campos, COC São José dos Campos e UNESP de Guaratinguetá, pois a vocês devo tudo que sei. E o maior de todos os agradecimentos a minha amada Renata Aparecida Alves Rocha quem me tirou da prisão da ignorância e me trouxe à luz dos estudos, pois nunca fui nada e nem nunca seria alguma coisa sem você.

RESUMO

Uma das tarefas mais desafiadoras na atividade de teste de software é a geração de casos/dados de teste. Embora haja uma quantidade significativa de estudos nesse sentido, ainda é necessário avançar em direção a abordagens que possam gerar casos/dados de teste com base apenas no código-fonte, considerando que muitos sistemas possuem apenas o código-fonte, sem a existência de uma documentação adequada. Esta Dissertação de Mestrado apresenta um novo método, denominado Singularity, para a geração de casos de teste unitários a partir de código-fonte em C++ e utilizando verificação de modelos, sendo este um método de Verificação Formal. Na presente abordagem, que é apoiada por uma ferramenta, o código-fonte C++ é traduzido automaticamente para modelos intermediários (Máquina de Estados Finitos, Grafo de Fluxo de Controle) e depois para a notação do Verificador de Modelos NuSMV. Posteriormente, uma técnica baseada no método HiMoST e em propriedades reconhecidas na literatura como "armadilha" produzem contraexemplos do verificador de modelos que são, de fato, os casos de teste abstratos em si. O método Singularity foi aplicado a um conjunto de códigos em C++ extraído de duas ferramentas complexas de sensoriamento remoto desenvolvidas no Instituto Nacional de Pesquisas Espaciais (INPE). Os resultados demonstraram a viabilidade do método para a geração de casos de teste.

Palavras-chave: Casos de Teste de Software. Teste Unitário. Verificador de Modelo. Automatização de Teste. Aplicações em C++.

SINGULARITY: A METHOD FOR AUTOMATIC GENERATION OF UNIT TEST CASES BASED ON MODEL CHECKER COUNTEREXAMPLES FOR C++ APPLICATIONS

ABSTRACT

One of the most challenging tasks in software testing activity is the generation of test cases/data. Although there is a significant amount of studies in this direction, it is still necessary to move towards approaches that can generate test cases/data based only on the source code, since many systems have only the source code, without necessarily the existence of adequate documentation. This Master Dissertation presents a new method, called Singularity, for the generation of unit test cases/data from C++ source codes using Model Checking, a Formal Verification method. In the present approach, which is supported by a tool, the C++ source code is automatically translated into intermediate models (Finite State Machine, Control Flow Graph) and then into the NuSMV model checker notation. Subsequently, a technique based on the HiMoST method and the called "trap properties" produces counterexamples of the model checker which are, in fact, the test cases/data itself. The Singularity method was applied to a set of C++ codes extracted from two complex remote sensing tools developed at the National Institute for Space Research (INPE). The results demonstrated the viability of the method for generating test cases/data.

Keywords: Software Testing Cases. Unit Testing. Model Checking. Test Automation. C++ Applications.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

AST	–	Abstract Syntax Tree
CTL	–	Computation Tree Logic
GFC	–	Grafo de Fluxo de Controle
INPE	–	Instituto Nacional de Pesquisas Espaciais
MEF	–	Máquina de Estados Finitos
NuSMV	–	"New" Symbolic Model Verifier
PST	–	Programa Sob Teste
SIG	–	Sistema de Informação Geográfica
SMV	–	Symbolic Model Verifier
TTR	–	T-Tuple Reallocation
XML	–	Extensible Markup Language

SUMÁRIO

Pág.

1 INTRODUÇÃO

A produção de software é uma atividade contínua. A onipresença da tecnologia na atualidade sugere uma grande e constante produção de software em um incontável número de áreas. Dentro do contexto do Instituto Nacional de Pesquisas Espaciais (INPE) a criação de software decorre de pesquisa espacial, um cenário que demanda qualidade. Para garantir essa qualidade, o código produzido precisa ser testado de maneira rigorosa. Desse modo, atividades de Verificação e Validação (V & V) podem ser empregadas visando esse objetivo (BAIER; KATOEN, 2008). O teste de software é a atividade de V & V mais utilizada na prática (DE SANTIAGO JÚNIOR; DA SILVA, 2017). Idealmente, deve-se ter uma documentação de software adequada, tal como especificação de requisito e a condução contínua de testes, buscando certificar se o que o código-fonte faz reflete o comportamento especificado na documentação estabelecida. Porém, isso pode não ser uma realidade, mesmo em áreas abrangidas pela pesquisa espacial. Infelizmente, é comum encontrar softwares em produção, independente do contexto onde se encontram, com uma documentação muito pobre (BRIAND, 2003). Nesta situação, para elaborar testes necessários, o próprio código-fonte pode ser útil para a geração de casos/dados teste.

A escrita de software de qualidade é um diferencial desejado e que requer sólidas técnicas de V & V. Verificação de Modelo (do inglês, *Model Checking*) é um método de Verificação Formal já conhecido da academia e de alguns setores da indústria. O potencial inerente aos métodos formais tem levado ao seu crescente uso por engenheiros para verificação de software complexos ou sistemas de hardware (BAIER; KATOEN, 2008). A geração de casos [1] baseada na Verificação de Modelo tem atraído a atenção da comunidade de engenharia de software nas últimas décadas. Fraser & Wotawa (2007) destacam que a abordagem de geração de casos de teste utilizando Verificação de Modelo (MCMILLAN, 1993), um tipo de ferramenta que é uma instância da teoria de Verificação de Modelo, tem vivenciado boa popularidade, sendo uma técnica conveniente que pode ser totalmente automatizada, bastante flexível e, sob devidas circunstâncias, bastante eficiente.

1.1 Motivação

Como mencionado anteriormente, em diversas ocasiões inclusive em áreas relacionadas à pesquisa espacial, projetistas de teste e testadores somente têm à sua disposição o código-fonte das aplicações. Desse modo, esse é um dos únicos artefatos

[1] A partir desse ponto, será usado o termo “casos de teste" apenas, e não "casos de teste". No capítulo 2, esses conceitos serão definidos adequadamente.

disponíveis para apoiar a atividade de teste de software e suas tarefas.

Não é novidade falar na automatização de testes de software para facilitar e melhorar a busca por qualidade. Como salientado por Deutsch (1981), diante de um software complexo, o esforço necessário para testá-lo é ainda maior e se confiado esse trabalho à mãos humanas, os recursos exigidos quase que certamente serão dispendiosos e ineficientes. Esse cenário inspirou a criação de testes automatizados para assistir a produção de testes efetivos, eficientes e mensuráveis. Escrever testes de software manualmente é um processo difícil e entediante que muitas vezes resulta em uma cobertura pobre do cenário sob teste, obrigando os desenvolvedores a gastarem muita energia melhorando os casos de teste escritos à mão, e extrapolando o orçamento previsto pelos desenvolvedores (YOSHIDA et al., 2017).

Outra motivação é a linguagem de programação utilizada para o desenvolvimento de software. O INPE desenvolveu e desenvolve diversos produtos de software em C++. Esta linguagem tem vivido um bom momento no cenário mundial de programação, estando em quarto lugar entre as linguagens mais utilizadas do mundo de acordo com o índice TIOBE (TIOBE, 2020). Ferramentas que auxiliam a automatização de testes para código-fonte desenvolvido nessa linguagem são fundamentais para que tais produtos de software tenham alta qualidade.

Um exemplo da aplicação dessa linguagem é a ferramenta *TerraLib*, uma biblioteca de software GIS de código aberto para apoiar o desenvolvimento de aplicativos geográficos personalizados (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, b). Parte dessa ferramenta, o Sistema de Informação Geográfico (SIG) *TerraView*, trata dados vetoriais (pontos, linhas e polígonos), matriciais (grades e imagens) e seus respectivos atributos (tabelas) armazenados em bancos de dados relacionais ou geo-relacionais disponíveis no mercado (CREPANI; MEDEIROS, 2005). Outra ferramenta também desenvolvida em C++, o GeoDMA é um plugin para o *TerraView*, sendo uma caixa de ferramentas para integrar métodos de análise de imagens de sensoriamento remoto com técnicas de mineração de dados, produzindo um ambiente computacional rico, extensível e centrado no usuário para extração de informações e conhecimento em grandes bancos de dados geográficos (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, a). O GeoDMA é desenvolvido sob a Fundação Software Livre e de Código Aberto (FOSS).

Diante desse contexto, é muito interessante pensar em um método, apoiado por ferramenta computacional, que possa apenas baseando-se no código-fonte de aplicações em C++, gerar automaticamente casos de teste para avaliar a qualidade de produtos

de software desenvolvidos pelo INPE. Ao criar um método que alcance essa meta, é esperado que tal método possa também ser utilizado para diversas aplicações desenvolvidas em C++ em um contexto geral, e não somente para sistemas da área espacial.

Já são conhecidas muitas ferramentas que geram casos de teste a partir de um código fonte. Algumas delas serão apresentadas na Seção 2.6, como os trabalhos de Beyer et al. (2007), Rayadurgam e Heimdahl (2001) e Ammann et al. (1998), que geram casos de teste a partir de um código fonte, porém não suportam como entrada a linguagem C++, ou os trabalhos de Garg et al. (2013), Bashir e Goel (1994), Bonfanti et al. (2018) e Mao e Lu (2007), que suportam a linguagem C++ como entrada porém utilizam formas diferentes do método aqui proposto para geração de casos de teste. Outras propostas para geração de casos de teste a partir de código fonte são a ferramenta Proteum (DELAMARO, 1993), que produz casos de testes baseados em análise de mutantes a partir da linguagem C, a ferramenta Austin (LAKHOTIA et al., 2013), também para geração de casos de teste para a linguagem C, a ferramenta EvoSuite (FRASER; ARCURI, 2011), capaz de criar suites de testes completas para a linguagem Java, e a ferramenta JaBUTi (VINCENZI et al., 2003), que gera casos de teste caixa-branca a partir do bytecode da linguagem Java. Um destaque fica para a ferramenta KLOVER (YOSHIDA et al., 2017), que tem uma proposta semelhante de geração de casos de teste unitários para C++ ao que será apresentada nesse trabalho. Porém, tal ferramenta é de uso proprietário (PRASAD et al.,) e não se sabe, com clareza, qual é a escalabilidade da mesma ao lidar com sistemas de software mais complexos.

1.2 Objetivo e metodologia de pesquisa

O objetivo da presente Dissertação de Mestrado é contribuir para a melhoria da qualidade de produtos de software criados em linguagem de programação C++, via a automatização da geração de casos de teste.

Para alcançar esse objetivo, o método, denominado Singularity (ERAS et al., 2019), propõe apoiar a geração automatizada de casos de teste com foco na geração de testes unitários para a linguagem C++, considerando somente o código-fonte e baseando-se em contraexemplos de um Verificador de Modelo. O método Singularity, que é apoiado por uma ferramenta, é capaz de ler código-fonte em C++ e o traduz, automaticamente, para um modelo (Máquina de Estados Finitos (MEF) e posteriormente para um Grafo de Fluxo de COntrole (GFC)) e depois para a notação do verificador de modelos NuSMV (CIMATTI et al., 1999). Após isso, baseando-se

no método HiMoST (DE SANTIAGO JÚNIOR; DA SILVA, 2017) e propriedades denominadas "armadilha", contraexemplos do verificador de modelo são gerados e, tais contraexemplos são de fato os casos de teste em si. No entanto, esses contraexemplos são considerados casos de testes abstratos que servem para direcionar a criação dos casos de testes executáveis/concretos.

Ao elaborar o método Singularity, quanto a geração casos de teste abstratos para classes C++, buscou-se responder a seguinte hipótese:

"Gerar uma maior quantidade de casos de teste por classe, proporciona uma maior cobertura do código, prejudica a cobertura do código ou é irrelevante para cobertura do código?"

Para testa essa hipótese, o método Singularity foi aplicado a dois sistemas de software complexos da área espacial, os produtos TerraLib (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, b) e GeoDMA (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, a), utilizando propriedades armadilha escolhidas especialmente para gerar um grande número de casos de teste abstratos por classe, usando exclusivamente o código-fonte C++ como entrada.

1.3 Contribuições e limitações

A maior contribuição do método Singularity é justamente a capacidade de gerar casos de teste abstratos para aplicações desenvolvidas em C++. O método e a instância do mesmo, ou seja a ferramenta que o apoia, conseguiram lidar com classes bastantes complexas com até, aproximadamente, 1000 linhas de código. Mesmo se escolhido apenas um único contraexemplo (caso de teste abstrato), o maior, entre todos os contraexemplos gerados pelo método, é possível criar um teste unitário com uma cobertura interessante da MEF que representa o código.

A maior limitação do método Singularity é o fato de não se conseguir gerar os casos de testes executáveis/concretos. No entanto, é importante mencionar a complexidade das estruturas de dados que são lidadas pelos métodos das classes dos produtos, TerraLib e GeoDMA, que foram usados como estudos de casos. Em muitas situações, as próprias classes sob teste são de fato os dados de entrada dos casos de testes executáveis. Por exemplo, a entrada de um certo método A de uma classe que está sendo testada são geradas em outra classe B, que depende de uma estrutura descrita por outra classe C, e assim por diante. Essas estruturas de dados podem ser, por exemplo, uma descrição de um mapa de uma cidades, uma rodovia, etc.

Portanto, sob a perspectiva de automatização de testes unitários via código-fonte, a visibilidade/disponibilidade de tais estruturas de dados não é algo que se obtém trivialmente como no caso de outras aplicações que possuem métodos explícitos, todos com parâmetros em suas assinaturas e com dados de entrada de teste baseados em estruturas primitivas.

Apesar da limitação mencionada acima e algumas outras, como a criação de uma quantidade relativamente grande de contraexemplos por classe, onde muitos podem ser inválidos, não se pode ignorar que a abordagem gerou resultados positivos considerando aplicações não triviais, sendo uma base para futuras pesquisas e melhorias para o teste de sistemas, não somente da área espacial mas de outros domínios que usam a linguagem C++.

1.4 Organização do texto

- O Capítulo 2 expõe uma breve revisão sobre os temas que estão relacionados a essa dissertação de mestrado;
- O Capítulo 3 descreve o método com o auxílio de um exemplo de execução;
- O Capítulo 4 analisa os dados gerados pela ferramente mediante a execução de casos de teste extraídos dos códigos das aplicações de software TerraLib e GeoDMA;
- O Capítulo 5 expõe uma avaliação sobre os resultados obtidos e propõe melhorias a serem exploradas em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo apresenta uma breve fundamentação teórica sobre os principais temas abordados neste trabalho. Temas como Teste de Software e Verificação de Modelo serão tratados aqui, assim como uma lista de trabalhos relacionados.

2.1 Teste de software

O Teste de Software tem sido parte do desenvolvimento desde os primeiros computadores (BINDER, 2000). Testar é uma forma natural de garantir que o produto desenvolvido apresente as características desejadas e funcione dentro do esperado, especialmente útil para assegurar a qualidade de um sistema complexo, muito comum na computação.

A natureza muitas vezes complicada de um programa de computador requer o uso de métodos específicos para execução de testes que cubram de forma eficiente o sistema em avaliação. Segundo Fraser et al. (2009), o teste continua sendo o mais importante método de verificar a qualidade de um produto de software, e é desejável que este seja automatizado pois o teste manual requer um imenso esforço e é suscetível a erros. Testes automatizados de software são o foco do presente trabalho.

A área de teste de software é muito ampla e estudos relacionados compreendem um extenso vocabulário de definições e termos. Uma pequena amostra desses termos pode ser encontrada no trabalho de Jorgensen (2013) e Delamaro et al. (2017), onde seis termos básicos utilizados para descrever um cenário de teste foram selecionados e brevemente explicados:

- **Defeito**: Pessoas cometem enganos. Enganos cometidos durante a codificação de um programa geram defeitos, também conhecidos por faltas ou *bugs*;
- **Erro**: Um erro decorre da execução de um defeito levando o programa a um estado incorreto;
- **Falha**: Uma falha decorrente da execução de um erro que leva a uma falha do sistema. Falhas são dinâmicas e requerem uma ação, são comumente associadas a erros de inclusão de elemento defeituoso. Mas existem também falhas causadas por erros de omissão de elemento correto.
- **Incidente**: Um incidente é um sintoma associado à ocorrência de uma

falha. Usualmente, o incidente é o alerta ao usuário da existência de uma falha.

- **Caso de Teste**: Um caso de teste é caracterizado por uma sequência de passos de teste. Uma sequência difere de um conjunto porque a repetição de elementos é permitida e a ordem é importante. Os passos de teste podem ter dados de entrada de teste e resultado esperado relacionados a eles, ou podem simplesmente representar ações que devem ser executadas.

- **Caso de Teste Abstrato**: Um caso de teste abstrato é aquele cuja representação não permite que o mesmo seja efetivamente executado contra o Software Sob Teste (SST). Tal caso de teste serve como uma espécie de guia para a geração de um caso de teste executável.

- **Caso de Teste Executável**: Um caso de teste executável, ou concreto, é aquele cuja representação permite que o mesmo seja efetivamente executado contra o SST. Isso significa que os dados de entrada de teste e resultado esperado estão todos em um formato adequado, onde é possível estimular o SST com tais informações.

- **Teste**: Um teste é o ato de exercitar um software com casos de teste. Permite verificar em tempo de execução se o sistema se comporta conforme ou não conforme o esperado, quando submetido às condições de teste especificadas.

No entanto, é importante salientar uma observação adicional no contexto desse trabalho. Formalmente falando, um contraexemplo gerado pelo Verificador de Modelos (*Model Ckecker*) é um traço, ou seja, uma sequência que mostra as proposições atômicas que são satisfeitas em cada estado de um caminho de um Sistema de Transição, onde tal caminho refuta uma propriedade formalizada em lógica (BAIER; KATOEN, 2008). No entanto, nesse trabalho, existe um relacionamento entre a caracterização dos estados do Sistema de Transição (no contexto do Verificador de Modelos) e os vértices do Grafo de Fluxo de Controle (GFC) e os estados da MEF intermediários obtidos a partir de um código-fonte em C++. Então, um estado c_i do contraexemplo se relaciona a um vértice v_k do GFC intermediário, que por sua vez se relaciona a um estado m_j da MEF intermediária. O GFC é tomado como base para gerar o código do Verificador de Modelos. Como esse GFC e MEF foram obtidos a partir do código-fonte, passar por c_i significa passar por v_k e m_j e, dessa forma, passar por determinada parte do código-fonte. Portanto, um contraexemplo sugere uma parte

do código-fonte a ser coberta sendo um guia para o projetista de teste para gerar um caso de teste executável. Desse modo, considera-se que um contraexemplo é, de fato, um caso de teste abstrato.

A forma como um teste avalia um programa pode ser classificada quanto uma avaliação de sua utilidade ou de sua eficiência. De acordo com Deutsch (1981), Verificação e Validação de software são definidas como:

- **Verificação**: Atividade que garante que cada etapa do processo de desenvolvimento reflete a correta intenção estabelecida para aquele determinado procedimento;
- **Validação**: Atividade que garante que o produto final reflita as características prescritas nos requisitos e especificações do software.

A forma com que um teste de software é aplicado pode também ser classificada. Existem duas abordagens comuns em teste de software: Testes funcionais ou teste caixa preta e testes estruturais ou teste caixa branca.

- **Testes Funcionais**: Segundo Jorgensen (2013), são baseados no princípio que qualquer programa pode ser considerado uma função que mapeia valores de seu domínio de entrada para um intervalo de saída. Esse tipo de teste é comumente conhecido por "teste caixa preta", onde o conteúdo dessa caixa (a implementação do programa em teste) é desconhecido, e o funcionamento da caixa preta é conhecido unicamente em termos de suas entradas e saídas. Myers et al. (2004) ressalta que para utilizar essa estratégia de teste com o intuito de encontrar todos os problemas possíveis em algum software é preciso recorrer a um teste exaustivo, o que é considerado impraticável. Esse tipo de teste é usado baseado nas especificações do que se espera que um programa seja capaz de fazer.
- **Testes Estruturais**: Segundo Jorgensen (2013), testes estruturais ou testes caixa branca são, em contraste com os testes funcionais, baseados no conhecimento da implementação do programa a ser testado. Possuem a habilidade de ver o que há "dentro da caixa" e identificar os casos de teste baseado na forma como a função é implementada. Novamente, Myers et al. (2004) enfatiza que uma abordagem de tentar cobrir todos os caminhos possíveis dentro de um programa é tão inviável quanto o teste exaustivo caixa

preta. É comum que programas apresentem um número muito grande de possíveis caminhos de execução e mesmo que fosse possível exercitar cada um deles isso não iria garantir que o que o programa faz no final é o que ele realmente deveria fazer. O conhecimento do funcionamento interno de uma aplicação abre portas para criação de testes que cobrem uma quantidade desejável do código e validem funcionalidades específicas.

Segundo Delamaro et al. (2017), a atividade de teste pode ser dividida em três fases com objetivos distintos:

- **Teste Unitário** : Tem como foco as menores unidades do sistema: funções, procedimentos, métodos ou classes. Busca encontrar erros de implementação, estruturas de dados incorretas, erros de programação, etc. Pode ser aplicado durante a implementação sem a necessidade de ter o sistema completo para seu uso. Mais detalhes sobre esse tipo de teste são abordados em 2.2.

- **Teste de Integração** : Ocorre após realizado os testes de unidade. Objetiva avaliar a interação das partes do software e avalia a construção da estrutura do sistema. Uma vez obtidas as partes que irão compor o programa final, o teste de integração verifica se a interação entre essas partes não leva a erros.

- **Teste de Sistema** : Executado após a conclusão do sistema como um todo. Verifica se as funcionalidades especificadas na documentação de requisitos estão corretamente implementadas. Requisitos funcionais e não funcionais são verificados e validados nessa etapa.

E para qualquer uma das fases anteriores, na decorrência de uma atualização ou correção de software, um teste específico para revalidar o programa pode ser utilizado:

Teste de Regressão : Executado durante a manutenção do software. Após a atualização de alguma parte do sistema ou mesmo após a conclusão do sistema e sua liberação para o usuário final, atualizações e modificações criam o risco de introdução de novos erros. É preciso verificar se as modificações realizadas estão corretamente implementadas e se os requisitos anteriores ainda pertinentes continuam válidos.

2.1.1 Cobertura de teste

A abrangência de um teste está conectada a qualidade do conjunto de testes utilizado. Segundo Howden (1981), uma forma ideal de garantir a completude de um conjunto de testes é a propriedade de que a corretude de um programa em relação a um conjunto completo de testes T implicaria a corretude do programa em todos os dados, sendo suficiente para um programa ser correto nesse conjunto T para ser considerado correto para qualquer outro tipo de entrada. Porém a obtenção de semelhante conjunto completo de testes é considerada inatingível, então definições mais fracas de completude são utilizadas. A obra de Westley (1979) menciona que um teste exaustivo de todas as partes de um sistema sobre todas as possibilidades de valores de entrada é impraticável e desnecessário. Logo foram desenvolvidos métricas de cobertura onde o exercício de um pequeno conjunto de testes pode ser extrapolado para todas as possíveis entradas e assim seria estimado um nível de confiança para todo código.

Howden (1981) sugere o uso de um teste de ramificação, onde afirma que um conjunto de casos de teste é considerado completo se cada ramo do código é visitado pelo menos uma vez. Um teste de ramificação completo é considerado um critério necessário para garantir a corretude de um programa dado que é preciso visitar todas as partes de um código para encontrar todos os defeitos.

2.2 Testes de unidade

De acordo com Pressman e Maxim (2016), o teste de unidade é o esforço de verificar a menor unidade de um software, o chamado módulo. Caminhos de controle importantes são testados para descobrir erros dentro das fronteiras do módulo, limitando a complexidade dos testes e a complexidade dos erros encontrados dentro do campo de ação restrito do teste unitário. Um teste de unidade é sempre baseado em caixa branca (PRESSMAN; MAXIM, 2016).

Uma vez que o módulo sob teste não é um programa individual, é preciso criar um ambiente onde esse módulo possa ser testado. Podem ser necessários *Drivers*, ou programas principais, que forneçam os dados necessários ao módulo em teste ou *stubs*, módulos subordinados, que são chamados pelo módulo em teste durante sua execução. Esse ambiente, seja utilizando elementos reais ou simulados, é chamado *overhead*. É desejável manter o *overhead* simples durante a implementação de testes mas infelizmente muitos módulos não podem ser testados a nível de unidade sem um *overhead* muito complexo (PRESSMAN; MAXIM, 2016). Módulos mais coesos

apresentam um número de casos de teste reduzido e mais fácil detecção de erros.

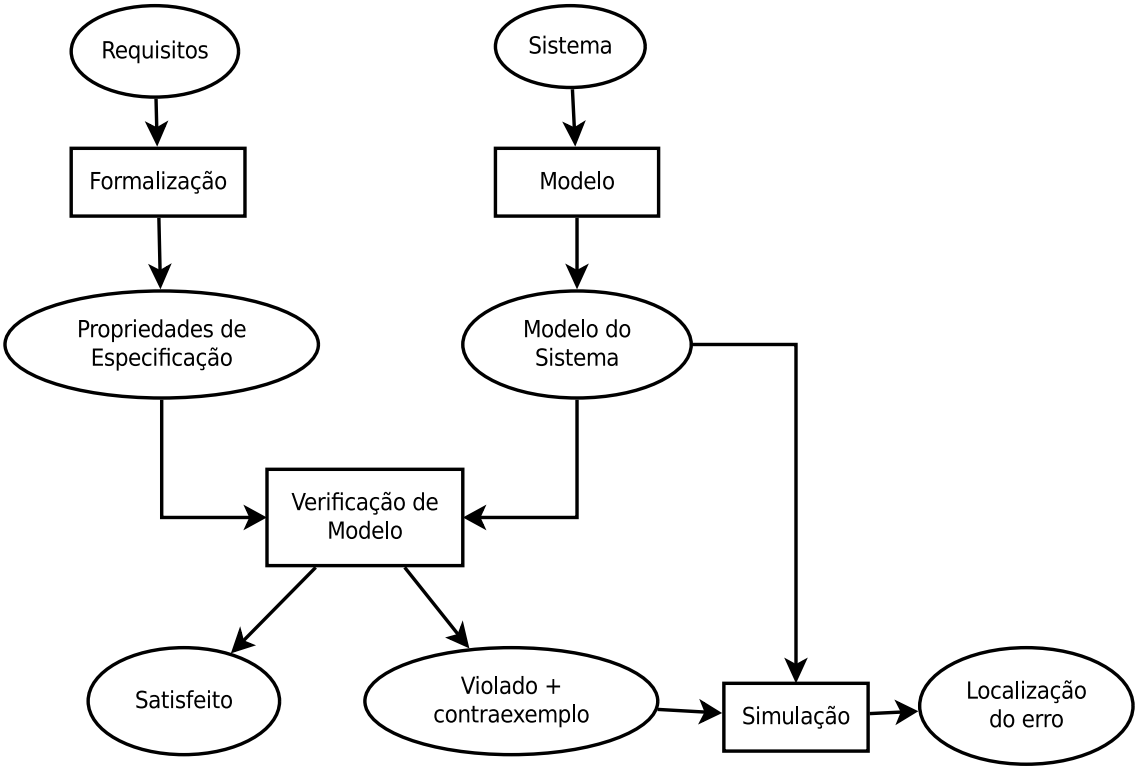
2.3 Verificação de modelo

Segundo Delamaro et al. (2017), é possível fazer um modelo que represente o sistema que se deseja testar, sendo esse modelo utilizável como um oráculo para testes definindo o que seria um comportamento adequado ou errôneo. Para isso no entanto, é preciso saber se o modelo está correto, isso é, ele precisa ser testado da mesma forma que o próprio sistema. Comprovado o modelo estar adequado ele é de grande valor para geração de cenários de teste. Um ponto importante é a estrutura do modelo: é possível descrever as ações realizadas por um sistema através de um grafo onde os nós representam os estados e as arestas as transições entre os estados. Esse formalismo é denominada de "Sistema de Transição de Estados"ou "máquina de estados".

Um modelo de um sistema descrito por uma máquina de estados é o elemento base da verificação formal de software. Baier e Katoen (2008) apresentam uma revisão sobre o tema que será utilizada como base para o restante desta seção. O objetivo da verificação formal de software é estabelecer a corretude de um sistema com rigor matemático. Técnicas de verificação baseada em modelo descrevem matematicamente o possível comportamento de um sistema de forma precisa e livre de ambiguidades. Como ponto de partida dessa técnica, temos o modelo do sistema em consideração: Qualquer verificação baseada em modelo é tão boa quanto o modelo utilizado.

A Verificação de Modelo requer uma declaração precisa e inequívoca das propriedades a serem examinadas. O modelo do sistema é comumente gerado automaticamente a partir de uma descrição de modelo que é especificada em algum dialeto apropriado de linguagens de programação. As propriedades de especificação devem prescrever o que o sistema deve e o que o sistema não deve fazer enquanto a descrição relata como o sistema de fato faz. A Verificação de Modelo examina todos os estados relevantes do sistema para verificar se eles satisfazem a propriedade desejada. Um modelo pode satisfazer as propriedades testadas quando todos os estados estão de acordo com a propriedade em consideração ou algum estado pode violar essa propriedade, gerando um contraexemplo. Esse contraexemplo descreve um caminho de execução que leva do estado inicial do sistema ao estado que violou a propriedade que está sendo verificada. Com o auxílio de um simulador, o usuário pode reproduzir o cenário de violação, obtendo assim informações úteis de depuração e correção do modelo (ou da propriedade) de acordo com o defeito detectado. Esse processo pode ser visto na Figura 2.1.

Figura 2.1 - Esquema da Verificação de Modelo



Fonte: Adaptado de Baier e Katoen (2008).

2.3.1 Sistema de transição de estados

Utilizado para descrever o comportamento de um modelo a ser verificado, Sistemas de Transição de Estados são basicamente grafos direcionais onde cada nó representa um estado e cada aresta uma transição entre os estados. De acordo com Baier e Katoen (2008), um Sistema de Transição de Estados TS é uma tupla $(S, A, \rightarrow, I, PA, L)$ onde

- S é um conjunto de estados,
- A é um conjunto de ações,
- $\rightarrow \subseteq S \times A$ é uma relação de transição,
- I é um conjunto de estados iniciais,
- PA é um conjunto de proposições atômicas, e
- $L : S \rightarrow 2^{PA}$ é uma função de rotulação.

2.3.2 Lógica CTL

Computation Tree Logic (CTL) ou Lógica de Árvore de Computação, uma ramificação da lógica temporal, é um formalismo utilizado para descrever propriedades na Verificação de Modelo (BAIER; KATOEN, 2008).

Brevemente, é utilizado a seguinte notação para descrição das propriedades em CTL:

Tabela 2.1 - Notação CTL.

	Notação	Significado
Quantificador de Caminho	$\forall$ ou A	para todos os caminhos
	$\exists$ ou E	para algum caminho
Modalidade Temporal	$\Box$ ou G	sempre ou globalmente
	$\diamond$ ou F	eventualmente
	$\bigcirc$ ou X	próximo
	$\cup$ ou U	até

Para ilustrar o uso dessa notação, vamos considerar a o seguinte requisito. Conhecido como "exclusão mútua", esse requisito descreve a necessidade de que durante a execução de dois programas concorrentes (programas p_1 e p_2), suas chamadas seções críticas nunca devem ser executadas simultaneamente:
Requisito. *"Processos p_1 e p_2 nunca devem, simultaneamente, ter acesso a suas seções críticas c_1 e c_2".*

Em notação CTL esse mesmo requisito pode ser descrito usando os símbolos $\forall$ (sempre) e $\Box$ (globalmente) para descrever que as zonas críticas nunca podem ocorrer juntas:
Requisito. $\forall\Box(\neg(c_1 \wedge c_2))$

2.4 Grafo de fluxo de controle

A maioria dos critérios estruturais utiliza uma representação de programa conhecida como Grafo de Fluxo de Controle (GFC) ou Grafo de Programa, no qual blocos disjuntos de comandos são considerados nós. A execução do primeiro comando do bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos em um bloco têm um único predecessor (possivelmente com exceção do primeiro) e um único sucessor (possivelmente com exceção do último) (DELAMARO et al., 2017).

A representação de um programa como um GFC estabelece uma correspondência entre nós e blocos de comandos e indica possíveis fluxos de controle entre blocos por meio de arcos. Um GFC é um grafo orientado com um único nó de entrada e um único nó de saída. Cada nó representa um bloco indivisível (sequencial) de comandos e cada arco representa um possível desvio de um bloco para outro. A partir do GFC, podem ser escolhidos os elementos que descrevem as propriedades de interesse, caracterizando assim o teste estrutural (DELAMARO et al., 2017)

2.5 Complexidade ciclomática

As informações sobre a complexidade do programa podem ser utilizadas para derivar casos de teste. O Critério de McCabe ou Teste de Caminho Básico, é o critério baseado em complexidade mais conhecido (DELAMARO et al., 2017). Chamado de complexidade ciclomática, esse critério dá o número de caminhos independentes de um programa, sendo "caminho independente" qualquer caminho que introduza pelo menos um novo conjunto de instruções ou uma nova condição, ou seja, que introduza pelo menos um arco que não tenha sido percorrido no GFC do programa. Essa métrica proporciona uma medida quantitativa da complexidade lógica de um programa.

De acordo com Mccabe (1976), suponhamos que um programa P tenha sido escrito, sua complexidade v tenha sido calculada e o número de caminhos testados seja c. Se $c < v$, uma das seguintes condições deve ser verdadeira:

- Há mais testes a serem realizados;
- O fluxo do programa pode ser reduzido para $v - c$;
- Partes do programa podem ser resumidas.

Em outras palavras, se um programa tem uma complexidade v, o número mínimo de caminhos a serem testados para cobrir todo programa é v. A complexidade ciclomática é calculada pela fórmula:

$$c(g) = e - n + 2$$

onde:

- $c(g)$ é a complexidade ciclomática do grafo g;

- e é o número de arestas do grafo g;

- n é o número de vértices do grafo g.

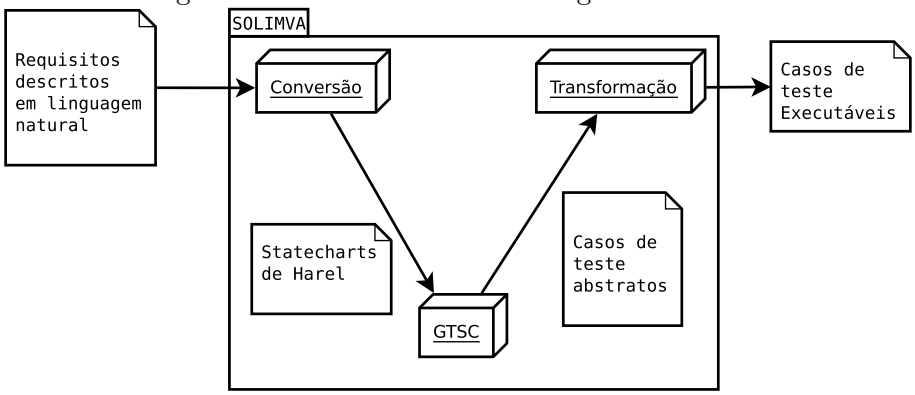
Para aplicação do critério de complexidade ciclomática, podemos considerar que a estrutura final do modelo gerado pelo método, testada pelo verificador de modelos, está muito próximo de um GFC. Nesse ponto, o critério de McCabe pode ser aplicado e a complexidade ciclomática do programa sob teste pode ser estimada. Mais detalhes sobre a estrutura do método serão vistos no Capítulo 3.

2.6 Trabalhos relacionados

Existem muitos trabalhos na área de geração automática de testes de software. Essa seção traz uma pequena amostra de trabalhos relacionados à presente pesquisa, alguns utilizando contraexemplos do verificador de modelo para geração de casos de testes, outros gerando testes de unidade e alguns apenas mencionando técnicas similares, cada um com suas particularidades.

de Santiago Júnior (2011) apresenta a metodologia SOLIMVA para geração de casos de testes a partir de requisitos escritos em Linguagem Natural (LN). Uma ferramenta, que apóia a metodologia, é responsável pela transformação dos requisitos em LN para modelos em Statecharts (HAREL, 1987), enquanto outra ferramenta, Geração Automática de Casos de Teste Baseada em Statecharts (GTSC) (SANTIAGO et al., 2008), é usada para gerar casos de teste abstratos que depois são transformados em casos de teste executáveis. Um resumo dessa metodologia é apresentado na Figura 2.2. A diferença básica desse trabalho e do método proposto nessa dissertação é que o Singularity se baseia em cógido-fonte em C++ e Verificação de Modelo para gerar os casos de teste, enquanto a SOLIMVA se baseia em requisitos em LN e os casos de testes são gerados via critérios de teste para Máquinas de Estados Finitos. Juntamente com a extensão proposta em (DE SANTIAGO JÚNIOR; DA SILVA, 2017), esse trabalho é base para o desenvolvimento da pesquisa atual.

Figura 2.2 - Resumo da metodologia SOLIMVA.



Fonte: O autor.

Em (DE SANTIAGO JÚNIOR; DA SILVA, 2017), os autores apresentam um método de tradução baseado em hierarquia, denominado ***Hi****erarchy-based translation from Statecharts into* ***Mo****del Checking and* ***S****pecification Patterns Properties for* ***T****esting* (HiMoST), para gerar casos de teste de software via Model Checking. Começando com uma modelagem comportamental em notação de Statecharts (HAREL, 1987), o método converte de Statecharts para uma estrutura geral baseada na linguagem do Verificador de Modelo NuSMV. Além disso, as propriedades são formalizadas em CTL por meio de Padrões de Especificação (DWYER et al., 1999) e um algoritmo de Teste de Interação Combinatória, chamado *T-Tuple Reallocation* (TTR) (BALERA; SANTIAGO JÚNIOR, 2017). Esse trabalho foi a base para o desenvolvimento do método Singularity, mas no HiMoST a notação de entrada é um Statecharts enquanto no Singularity é o código-fonte, o que não requerer que o profissional de teste tenha que converter o código-fonte para Statecharts, ou qualquer outra notação, para ser usado.

Yoshida et al. (2017) introduziram o KLOVER, uma metodologia para geração de testes automatizados de softwares embarcados implementado em C ou C ++ baseada na tecnologia KLEE (LI et al., 2012), utilizando uma técnica de análise de programa chamada execução simbólica. A metodologia gera testes em nível de unidade, e clama propiciar excelente cobertura de teste. A abordagem proposta gera o ambiente necessário para isolar o código de entrada e entrega testes de unidade escritos com o framework da Google, o *gtest*. A ferramenta é de uso interno da Fujitsu Co. juntamente com as tecnologias KLEE e uma nova ferramenta de geração de teste para C++ construída de forma independente, chamada FSX (YOSHIDA et al., 2016; PRASAD et al.,). O KLOVER e o SFX são ferramentas que apresentam

uma proposta muito próxima do Singularity, tendo no entanto a desvantagem de serem proprietárias, de uso exclusivo das empresas Fujitsu.

O trabalho de Beyer et al. (2004) propõe uma variação de seu verificador de modelo *BLAST* (BEYER et al., 2007) para geração automática de casos de teste a partir de contraexemplos com respeito a uma dada propriedade p. Dado um código escrito em linguagem C, são descobertos todos os caminhos onde a dada propriedade p é satisfeita, gerando assim uma suíte completa de testes capaz de cobrir todos os pontos do código onde p (geralmente uma propriedade de segurança) é verdadeiro. A abordagem proposta foi usada para localizar, por exemplo, os pontos em um código onde o acesso só é permitido mediante uma senha válida (acesso *root*) e foi testado para localização do chamado "código morto", linhas de código inatingíveis por erros de lógica, em aplicações em linguagem C com até 3×10^4 linhas de código, relatando uma cobertura livre de falsos positivos e um procedimento inteiramente automatizado. Como principal diferença para o método Singularity, não existe suporte para linguagem C++.

Outro trabalho que faz o uso de contraexemplos do verificador de modelo para geração de casos de teste foi escrito por Rayadurgam e Heimdahl (2001) onde é apresentado um método de geração automática de casos de teste para critérios de cobertura estrutural. Uma sequência completa de testes é gerada automaticamente a partir de contraexemplos do verificador de modelo criando uma cobertura predefinida para qualquer tipo de software, desde que este seja representado por uma máquina de estados finitos. É formalizado um framework para geração de casos de teste que pode ser aplicado para obtenção de uma representação da entrada no formato de uma máquina de estados finitos a partir de qualquer tipo de artefato de software, seja o código fonte, as especificações do sistema ou o modelo de requisitos. A geração de contraexemplos faz o uso de propriedades armadilha para forçar a cobertura do código de entrada, usando exatamente as mesmas propriedades utilizadas pelo método proposto nesse trabalho. O método de Rayadurgam foi exemplificado com a geração de casos de teste para um pequeno sistemas de aviônicos de segurança crítica. Com foco na cobertura estrutural, esse método se diferencia do Singularity por não gerar dados de teste unitário.

Ammann et al. (1998) propõem o uso de Verificação de Modelo para geração de casos de teste a partir das especificações de um software. O método proposto utiliza testes baseados em mutantes para geração de contraexemplos do verificado de modelo, resolvendo um problema comum a esse tipo de teste que é a geração de

"mutantes equivalentes" (mutações no código que não afetam o seu funcionamento), pois esse tipo de mutação é simplesmente satisfeito pelo verificado de modelo e não gera nenhum contraexemplo. A geração de casos de teste é feita a partir desses contraexemplos. Os testes gerados são reduzidos pela eliminação de testes equivalentes e prefixos de caminhos repetidos. Os testes gerados são por fim aplicados ao código e analisados quanto a cobertura e validade dos testes. A validação é feita aplicando o método a um código escrito em linguagem Java. Esse método se diferencia da proposta do Singularity pois tem como entrada as especificações do software, e não o código fonte, o que exige a existência dessa documentação pra sua execução.

Garg et al. (2013) mostraram um método para geração automática de teste de unidade para programas escritos em C/C++. Focado em melhorar a cobertura obtida pelos métodos de geração de teste aleatório baseado em feedback, utiliza a chamada execução concólica (acrônimo das palavras concreto e simbólico) nos drivers de teste gerados. Foram utilizados solucionadores não lineares para programas com cálculos numéricos visando gerar novas entradas de teste. O método clama uma melhor cobertura que uma técnica puramente aleatória e um protótipo de ferramenta foi escrito para execução de benchmarks de código aberto C/C++. Com uma geração de testes baseada em aleatoriedade, sem foco específico em testes de unidade e em nível bem mais elevado de complexidade, esse método é bem diferente do método apresentado nessa dissertação.

O trabalho de Bashir e Goel (1994) propõe uma geração de testes unitários para classes C++ de forma semi exaustiva. Uma classe é dividida em "fatias" onde cada membro da classe é testado de forma individual sem causar a explosão de possibilidades oriunda da combinação dos membros. Uma sequência de combinações associada a cada fatia é criada e testada, permitindo uma abordagem próxima da exaustiva. Essa abordagem tem a proposta de ser complementar aos tradicionais testes funcionais e estruturais, suporta somente os comandos mais básicas da linguagem C++ e é apresentado e validado de forma exclusivamente teórica.

Bonfanti et al. (2018) propuseram uma abordagem para geração de testes unitários em C++ a partir de uma Máquina de Estados Abstrata (MEA). Munido de um modelo de sistema em MEA, uma ferramenta dos mesmos autores chamadas *Asm2C++* é capaz de gerar código C++ a partir desse modelo. A publicação mostra como gerar modelos de teste unitário usando o modelo do sistema e depois com a ferramenta transforma-se os dois em código C++, obtendo assim o código fonte e os testes unitários simultaneamente. A geração dos casos de teste é feita tanto

usando um verificador de modelo quando de forma aleatória. A solução proposta ainda requer algumas melhorias para solucionar o problema do não determinismo dos modelos MEA. Novamente, a exigência de representar o software a ser testado em uma notação diferente do código fonte é uma diferença em relação ao Singularity.

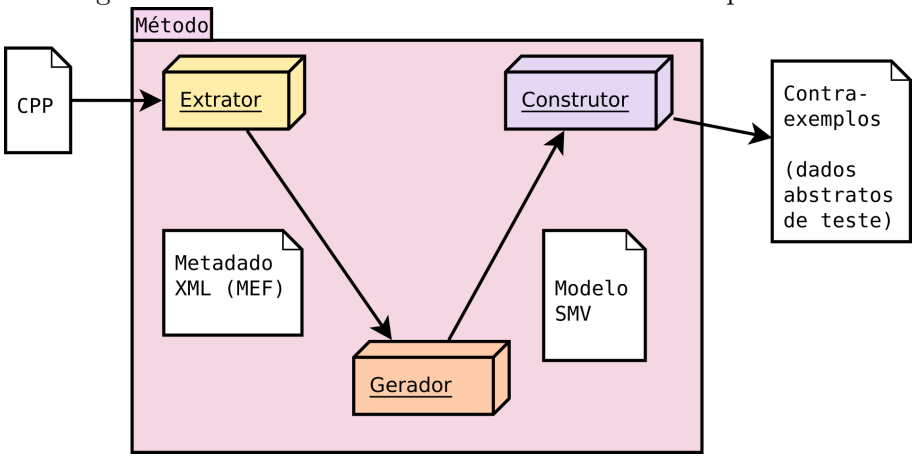
3 O MÉTODO SINGULARITY

Esse capítulo apresenta o método de geração de casos de teste a partir do código C++ com o auxílio de um exemplo, cuja concepção e resultados preliminares foram apresentados em ERAS et al. (2019).

3.1 A estrutura do método Singularity

O método é formado por três componentes: Extrator, Gerador e Construtor. Cada componente é responsável por uma parte do processo de transformação do código fonte C++ em uma entrada para o verificador de modelo, pela geração das propriedades e obtenção dos contraexemplos. A estrutura do método pode ser vista na Figura 3.1

Figura 3.1 - A estrutura do Método e seus três componentes.



Fonte: O autor.

A intenção do método é transformar o código fonte em um modelo de estados e transições para ser processado pelo verificador de modelo. Além disso, é preciso fazer com que o verificador de modelo produza contraexemplos que mostrem caminhos de execução possíveis dentro do código. A ideia de testar usando verificadores de modelo é interpretar os contraexemplos como casos de teste (FRASER; WOTAWA, 2007; DE SANTIAGO JÚNIOR; DA SILVA, 2017). Cada uma das três etapas principais desse processo serão aqui chamadas de componentes.

Figura 3.2 - Código fonte C++ do programa Problema do Triângulo.

```
#include <string>

class Triangle {
public:
    Triangle (int a, int b, int c) : a_(a), b_(b), c_(c) {}
    const int a_;
    const int b_;
    const int c_;
};

class Tool {
public:
    std::string classify (Triangle t)
    {
        if (is_valid(t))
        {
            if (t.a_ == t.b_ && t.a_ == t.c_)
            {
                return "Equilateral";
            }
            else if (t.a_ != t.b_ && t.a_ != t.c_ && t.b_ != t.c_)
            {
                return "Scalene";
            }
            else
            {
                return "Isoceles";
            }
        }
        else
        {
            return "Invalid";
        }
    }
private:
    bool is_valid (Triangle t)
    {
        if (t.a_ <= 0) {return false;}
        if (t.b_ <= 0) {return false;}
        if (t.c_ <= 0) {return false;}
        if (t.a_ + t.b_ <= t.c_) {return false;}
        if (t.a_ + t.c_ <= t.b_) {return false;}
        if (t.b_ + t.c_ <= t.a_) {return false;}
        return true;
    }
};
```

Fonte: O autor.

3.2 Um exemplo de programa em C++: o problema do triângulo

Para amparar a explicação do método, será utilizado um programa C++ simples como exemplo de execução. Esse programa, visto na Figura 3.2, propõe uma solução para o chamado *Problema do Triângulo*: Dado três valores inteiros a, b e c, verificar se esses valores podem ser utilizados como medidas para os lados de um triângulo e, caso configurem um triângulo válido, imprime na tela qual tipo de triângulo esses valores representam.

Para três valores serem considerados válidos na representação do tamanho dos três lados de um triângulo, eles precisam obedecer a duas regras:

a) Todos os três valores precisam ser maiores que zero;

b) A soma de dois valores precisa ser maior que o terceiro valor.

Uma vez considerado que os três valores de entrada formam um triângulo válido, esse triângulo será classificado como Equilátero, Escaleno ou Isósceles.

Este programa mostra alguns dos elementos básicos que se espera encontrar em um programa em C++: este apresenta duas classes, um construtor, duas funções e estruturas de decisão. Os atributos da classe *Triangle* foram mantidos com visibilidade pública para manter o código pequeno. Diferentes elementos serão processados pelo método e transformados em um diagrama de estados e transições.

3.3 O componente extrator

Quando um código fonte é recebido como entrada, o primeiro passo do método é extrair dele um modelo de estados e transições. Essa primeira etapa é realizada pelo componente extrator, responsável por identificar estados e transições no código C++. Para isso, cada linha do código é examinada podendo se tornar ou não um estado. Após a análise de cada linha, um conjunto de próximas linhas atingíveis é gerado. Essas transições possíveis são interpretadas como as transições do modelo, da mesma forma que uma linha de código pode ser interpretada como um estado. O processo de extração desse diagrama de transição de estados, visto no Algoritmo 1, é composto por duas funções: função de extração dos estados e função de extração das transições. É válido observar que nesse ponto o diagrama extraído não configura um GFC e sim modelo próximo a um sistema de estados e transições ou uma Máquina de Estados Finitos (MEF), somente depois da extração completa do modelo vista na Seção 3.4 é possível classificar o diagrama como um GFC. Essa classificação é desejável para futura avaliação de cobertura de estados e transições.

Algoritmo 1 extrator

entrada: código c++
saída: estados, transições

```
1: estados ← extratorDeEstados(código c++)
2: transições ← extratorDeTransições(código c++)
3: return estados, transições
```

Os estados e transições gerados a partir do código C++ são representados em lin-

guagem XML (Extensible Markup Language). Para manter a indentação do código fonte original, cada estado é dividido duas categorias: Estados que geram um novo escopo e estados que não geram um novo escopo. Considere o exemplo de código a seguir:

```
1   if (a > 0) {
2       b = a;
3       a ++;
4   }
```

Observa-se que a primeira linha do código `if (a > 0) {` cria um um novo nível de indentação pertencente ao escopo da estrutura de decisão `if`. Essa linha é classificada como um estado do tipo "nível", pois altera o nível de indentação do código. As segunda e terceira linhas do código por sua vez não alteram o nível de indentação: as instruções `b = a;` e `a ++;` pertencem ao mesmo escopo do código e não têm qualquer influência sobre o escopo ao qual pertencem. Esse tipo linha é classificado como um simples "estado". Manter uma boa indentação dos estados extraídos do código fonte, mesmo que o código original não esteja corretamente indentado, garante a organização e a melhor compreensão do código XML gerado pelo método.

3.3.1 Definição da representação em XML

Um estado do tipo nível é representado em linguagem XML por uma tag[1] de nome "level" com o seguinte formato:

```
<level label = "" element = "" visibility = "" id = "">
```

onde cada atributo é definido como:

- **label**: contém uma descrição ou palavra chave da linha de código C++ para ajudar a leitura do diagrama. Por exemplo; uma linha contendo a assinatura de uma função teria um "label" igual ao nome da função;
- **element**: descreve o elemento de código C++ que está sendo representado pela tag. Um estado do tipo nível pode ter um elemento igual a *namespace*, *function*, *class*, *decision*, *loop*, *statement* ou *exception*.
- **visibility**: um estado do tipo nível pode ser afetado pelo nível de visibi-

[1] "tag" é um elemento de marcação em linguagens de marcação como XML, HTML, XMI, dentre outras

lidade. Por exemplo, uma função pode ser declarada como "pública" ou "privada" dentro de uma classe. Porém nem todo estado do tipo nível precisa de uma visibilidade, um laço de repetição por exemplo não é afetado pela visibilidade do código e pode ser descrito sem esse atributo. Um estado do tipo nível pode ter uma visibilidade igual a *public* ou *private*.

- **id**: duas ou mais linhas de código podem ser idênticas, mas pertencendo a diferentes partes do código. Classificar uma tag somente por um "label" pode facilmente tornar o diagrama de estados ambíguo. Enquanto o "label" ajuda ao leitor identificar qual linha do código é representada pela tag, o "id" traz um identificador único que garante a exclusividade de cada estado do diagrama. Linhas de código que não irão configurar um "nó" no GFC não recebem um identificador único, por exemplo, uma assinatura de função pode ser muito importante para o código por ter o nome da função, o tipo do retorno, os parâmetros de entrada, mas não significam nada pro GFC pois não configuram um passo de execução, como uma instrução "if" ou "for" por exemplo. Um id é representado por um valor inteiro.

Um estado que não cria um novo escopo é representado por uma tag de nome "state" com o seguinte formato:

```
<state label = "" element = "" id = "">
```

onde os dois atributos, "label" e "id", seguem a mesma definição da tag de nível. O atributo "element" de um estado que não cria um novo escopo pode ser igual a *attribution*, *declaration*, *constructor*, *self*, *internal*, *external*, *jump*, *exception* ou *IO*.

Para descrever uma transição, uma tag de nome "transition" com o seguinte formato é utilizada:

```
<transitions from = "" to = "" event = "">
```

onde cada atributo é definido como:

- **from**: Estado de origem da transição. Representado pelo "id", possivelmente acrescido do "label" do estado.
- **to**: Estado de destino da transição. Representado pelo "id", possivelmente

acrescido do "label" do estado.

- **event**: Durante uma transição de estados, um evento pode ser disparado. A transição de uma condicional dispara os eventos "TRUE" ou "FALSE". Se nenhum evento é disparado durante uma dada transição, diz-se que um evento "lambda" ocorreu. Uma transição "lambda" entre dois estados que não geram um novo escopo não será considerada no GFC, pois uma sequencia de estados lineares é apenas um bloco de código simples, ou um "nó" do grafo.

3.3.2 O extrator de estados

Uma vez definido o diagrama de transições de estados em XML, o funcionamento da função extratora é bem simples: cada linha do código é iterada e, uma a uma, classificada como Nível ou Estado, como visto no Algoritmo 2.

Algoritmo 2 extratorDeEstados

entrada: código c++
saída: estados

```
1: for ∀ linha ∈ código c++ do
2:     if linha gera novo escopo then
3:         estados ← gerarNível(linha)
4:     else
5:         estados ← gerarEstado(linha)
6:     end if
7: end for
8: return estados
```

Para ilustrar esse funcionamento, será considerado o exemplo do Problema do Triângulo visto na Figura 3.2. Uma execução da função extratora de estados iria, primeiramente, ignorar a linha `#include <string>` por ser uma instrução do compilador e não uma linha de código C++. A primeira linha significativa encontrada seria:

```
class Triangle {
```

Essa linha contém a assinatura de uma classe e cria um novo escopo do código, logo é classificada como um estado do tipo "nível". O nome da classe é a melhor forma de identificar essa linha, portanto seu "label" será "Triangle". O elemento C++ presente

na linha, como já identificado, é uma classe. Uma classe não é afetada pelo nível de visibilidade, logo não recebe nenhum valor no atributo "visibility". Uma assinatura de classe é importante para manter a indentação e a organização do arquivo XML assim como é importante para posterior criação das transições, mas não irá configurar um nó no GFC, então não recebe um "id". A tag XML para essa linha fica:

```
<level label = "Triangle" element = "CLASS">
```

A próxima linha encontrada é uma declaração de visibilidade:

```
public:
```

o que significa que as próximas instruções terão visibilidade pública, mas essa linha por si só não gerará nenhum nó no GFC e também não afeta a indentação do XML, portanto essa linha não é considerada como um estado.

Em seguida, observa-se um construtor:

```
Triangle (int a, int b, int c){}
```

que assim como uma assinatura de função, um construtor cria um novo nível de escopo (mesmo que esse esteja vazio no nosso exemplo), é classificado como um estado do tipo "nível", tem seu nome como principal identificador ("label" igual à "Triangle"), seu elemento é um "construtor" e sua visibilidade é "pública". Novamente, não recebe nenhum "id". O código XML atualizado com essa nova tag fica:

```
<level label = "Triangle" element = "CLASS">
    <level label = "Triangle" element = "CONSTRUCTOR" visibility = "PUBLIC">
```

As três próximas linhas são declarações de constantes. Essas linhas não afetam o fluxo de execução do programa e, assim como a declaração de visibilidade, não são classificadas como estados no diagrama. Com isso, o escopo da classe é encerrado, e uma tag fechada é utilizada:

```
<level label = "Triangle" element = "CLASS">
    <level label = "Triangle" element = "CONSTRUCTOR" visibility = "PUBLIC">
<\level>
```

As duas próximas linhas são uma assinatura de classe e uma declaração de visibilidade:

```
class Tool {
public:
```

que recebem as mesmas classificações da assinatura e declaração de visibilidade da classe anterior:

```
<level label = "Tool" element = "CLASS">
```

Em seguida, observa-se uma assinatura de função:

```
std::string classify (Triangle t) {
```

Essa assinatura gera uma tag do tipo nível com um "label" igual ao nome da função, seu elemento é uma função e sua visibilidade é pública:

```
<level label='classify' element='FUNCTION' visibility='PUBLIC'>
```

Em seguida, observa-se uma estrutura de decisão "if":

```
if (is_valid(t)) {
```

essa linha abre um novo escopo no código, é identificada pelo nome "if", é do tipo "decisão" e será um nó muito importante no GFC, logo recebe um identificador único:

```
<level label='if' element='DECISION' id='1'>
```

O mesmo ocorre com a próxima linha:

```
<level label='if' element='DECISION' id='1'>
    <level label='if' element='DECISION' id='2'>
```

Dentro desse segundo "if" existe uma linha que não gera um novo escopo:

```
return "Equilateral";
```

Essa linha é um estado simples. Uma instrução de retorno de função é classificada como um "salto" e recebe um identificador único:

```
<state label='return' element='JUMP' id='3'>
```

Todo restante do código é composto por uma repetição destes elementos. Seguindo esses mesmos passos pode-se transformar todo o código C++ no diagrama de transição de estados em XML visto na Figura 3.3.

3.3.3 O extrator de transições

Os estados extraídos pela função vista na Subseção 3.3.2 contém informações importantes para se traçar as transições: é mostrado as assinaturas de classes e funções do código, as estruturas de decisão e todas as demais linhas, já assinaladas com um identificador único. As assinaturas de classes e funções mostram o caminho por onde começar a varrer o código em busca das transições e os identificadores únicos que delimitarão o GFC. A ideia do extrator de transições é simular uma função "main" onde todas as classes e funções do código são chamadas para que cada linha dentro delas possa ser percorrida, anotando as transições entre os estados marcados com um identificador único.

Para obter as transições entre os estados uma função recursiva, vista no Algoritmo 3, examina um estado inicial e identifica todas os seus possíveis próximos estados. Uma chamada recursiva é feita para cada próximo estado identificado, caso esse não seja um estado final.

Figura 3.3 - Representação em XML dos estados gerados pela função *extratorDeEstados()*.

```
<level label='Triangle' element='CLASS'>
    <level label='Triangle' element='CONSTRUCTOR'>
    </level>
</level>
<level label='Tool' element='CLASS'>
    <level label='classify' element='FUNCTION' visibility='PUBLIC'>
        <level label='if' element='DECISION' id='1'>
            <level label='if' element='DECISION' id='2'>
                <state label='return' element='JUMP' id='3'>
            </level>
            <level label='else_if' element='DECISION' id='4'>
                <state label='return' element='JUMP' id='5'>
            </level>
            <level label='else' element='DECISION' id='6'>
                <state label='return' element='JUMP' id='7'>
            </level>
        </level>
        <level label='else' element='DECISION' id='8'>
            <state label='return' element='JUMP' id='9'>
        </level>
    </level>
    <level label='is_valid' element='FUNCTION' visibility='PRIVATE'>
        <level label='if' element='DECISION' id='10'>
            <state label='return' element='JUMP' id='11'>
        </level>
        <level label='if' element='DECISION' id='12'>
            <state label='return' element='JUMP' id='13'>
        </level>
        <level label='if' element='DECISION' id='14'>
            <state label='return' element='JUMP' id='15'>
        </level>
        <level label='if' element='DECISION' id='16'>
            <state label='return' element='JUMP' id='17'>
        </level>
        <level label='if' element='DECISION' id='18'>
            <state label='return' element='JUMP' id='19'>
        </level>
        <level label='if' element='DECISION' id='20'>
            <state label='return' element='JUMP' id='21'>
        </level>
        <state label='return' element='JUMP' id='22'>
    </level>
</level>
```

Fonte: O autor.

Algoritmo 3 extratorDeTransições

entrada: estados
saída: transições

```
1: function identificarTodosOsPossíveisPróximosEstados(estado)
2:     próximos ← identificarTransições(estado)
3:     for próximo ∈ próximos do
4:         transições ← criar uma transição do estado atual para o próximo
5:         if próximo ≠ estadoFinal then
6:             identificarTodosOsPossíveisPróximosEstados(próximo)
7:         end if
8:     end for
9: end function
10: identificarTodosOsPossíveisPróximosEstados(estado inicial)
11: return transições
```

O estado inicial é identificado simulando uma função "main" que chama todas as classes e funções do arquivo C++. Ao observar o diagrama de estados descrito na Figura 3.3, é visto que existem duas classes: a primeira delas, chamada "Triangle" não possui nenhum estado marcado com um identificador único; a segunda, chamada "Tool", possui estados marcados com identificadores únicos, logo essa classe é escolhida para se encontrar um estado inicial. Dentro dessa classe existem duas funções, uma com visibilidade pública e outra com visibilidade privada. Somente uma função de visibilidade pública é chamada pela simulação da função "main". Para uma função com visibilidade privada ser considerada pelo método, essa precisa ser chamada por alguma função pública do código. A primeira tag que possuí identificador único dentro da função pública "classify" é uma estrutura de decisão "if":

```
<level label='if' element='DECISION' id='1'>
```

Observando o código fonte é observado que a linha de código original tem uma chamada para função privada "is_valid", e para estrutura de decisão "if" prosseguir com sua execução é preciso primeiro visitar essa função chamada.

Dentro da função privada "is_valid" observa-se uma estrutura de decisão "if":

```
<level label='if' element='DECISION' id='10'>
```

Não existe nenhuma outra chamada de função na linha de código representada por

essa tag, então inicia-se o procedimento de identificar as transições a partir desse estado. É dito que existe uma transição de um "estado inicial" para esse estado. É dito que esse estado inicial tem um identificador único negativo e uma transição é construída usando as regras do diagrama XML definidas nas Subseção 3.3.1:

```
<transition from = 'initial_-1' to = 'if_10' event = 'lambda'>
```

A partir do estado "if_10" identifica-se os possíveis próximos estados. Caso a condição de guarda da estrutura "if" seja satisfeita, ocorre uma transição para linha de código dentro de seu escopo gerando um evento "TRUE", caso não seja satisfeita, o fluxo de execução é direcionado para uma próxima instrução "if" gerando um evento "FALSE":

```
<transition from = 'if_10' to = 'return_11' event = 'TRUE'>
<transition from = 'if_10' to = 'if_12' event = 'FALSE'>
```

Todas as demais linhas dessa função apresentam os mesmos elementos e podem ser processadas da mesma forma:

```
<transition from = 'if_12' to = 'return_13' event = 'TRUE'>
<transition from = 'if_12' to = 'if_14' event = 'FALSE'>
<transition from = 'if_14' to = 'return_15' event = 'TRUE'>
<transition from = 'if_14' to = 'if_16' event = 'FALSE'>
<transition from = 'if_16' to = 'return_17' event = 'TRUE'>
<transition from = 'if_16' to = 'if_18' event = 'FALSE'>
<transition from = 'if_18' to = 'return_19' event = 'TRUE'>
<transition from = 'if_18' to = 'if_20' event = 'FALSE'>
<transition from = 'if_20' to = 'return_21' event = 'TRUE'>
<transition from = 'if_20' to = 'return_22' event = 'FALSE'>
```

As linhas que contém um retorno de função levam o fluxo de execução de volta à primeira linha da função "Tool": a estrutura "if" de onde a chamada para função "is_valid" foi feita, gerando um evento igual ao seu valor de retorno:

```
<transition from = 'return_11' to = 'if_1' event = 'FALSE'>
<transition from = 'return_13' to = 'if_1' event = 'FALSE'>
<transition from = 'return_15' to = 'if_1' event = 'FALSE'>
```

```
<transition from = 'return_17' to = 'if_1' event = 'FALSE'>
<transition from = 'return_19' to = 'if_1' event = 'FALSE'>
<transition from = 'return_21' to = 'if_1' event = 'FALSE'>
<transition from = 'return_22' to = 'if_1' event = 'TRUE'>
```

De volta à função "Tool", é identificado os dois possíveis próximos estados para a estrutura de decisão "if", com identificador único igual a 1: caso sua condição de guarda seja verdadeira, uma transição para a primeira linha de seu escopo é feita gerando um evento "TRUE":

```
<transition from = 'if_1' to = 'if_2' event = 'TRUE'>
```

Seguindo o mesmo procedimento para todos os demais estados, são obtidas as transições mostradas na Figura 3.4. Note que os estados que pertencem ao final da execução do programa têm uma transição para um estado "final" com identificador único negativo.

Figura 3.4 - Representação em XML das transições geradas pela função *extratorDeTransições()*.

```
1  <transition from = 'initial_-1' to = 'if_10' event = 'lambda'>
2  <transition from = 'if_10' to = 'return_11' event = 'TRUE'>
3  <transition from = 'if_10' to = 'if_12' event = 'FALSE'>
4  <transition from = 'if_12' to = 'return_13' event = 'TRUE'>
5  <transition from = 'if_12' to = 'if_14' event = 'FALSE'>
6  <transition from = 'if_14' to = 'return_15' event = 'TRUE'>
7  <transition from = 'if_14' to = 'if_16' event = 'FALSE'>
8  <transition from = 'if_16' to = 'return_17' event = 'TRUE'>
9  <transition from = 'if_16' to = 'if_18' event = 'FALSE'>
10 <transition from = 'if_18' to = 'return_19' event = 'TRUE'>
11 <transition from = 'if_18' to = 'if_20' event = 'FALSE'>
12 <transition from = 'if_20' to = 'return_21' event = 'TRUE'>
13 <transition from = 'if_20' to = 'return_22' event = 'FALSE'>
14 <transition from = 'return_11' to = 'if_1' event = 'FALSE'>
15 <transition from = 'return_13' to = 'if_1' event = 'FALSE'>
16 <transition from = 'return_15' to = 'if_1' event = 'FALSE'>
17 <transition from = 'return_17' to = 'if_1' event = 'FALSE'>
18 <transition from = 'return_19' to = 'if_1' event = 'FALSE'>
19 <transition from = 'return_21' to = 'if_1' event = 'FALSE'>
20 <transition from = 'return_22' to = 'if_1' event = 'TRUE'>
21 <transition from = 'if_1' to = 'if_2' event = 'TRUE'>
22 <transition from = 'if_2' to = 'return_3' event = 'TRUE'>
23 <transition from = 'if_2' to = 'else_if_4' event = 'FALSE'>
24 <transition from = 'else_if_4' to = 'return_5' event = 'TRUE'>
25 <transition from = 'else_if_4' to = 'else_6' event = 'FALSE'>
26 <transition from = 'else_6' to = 'return_7' event = 'TRUE'>
27 <transition from = 'if_1' to = 'else_8' event = 'FALSE'>
28 <transition from = 'else_8' to = 'return_9' event = 'TRUE'>
29 <transition from = 'return_3' to = 'final_-1' event = 'Equilateral'>
30 <transition from = 'return_5' to = 'final_-1' event = 'Scalene'>
31 <transition from = 'return_7' to = 'final_-1' event = 'Isosceles'>
32 <transition from = 'return_9' to = 'final_-1' event = 'Invalid'>
```

Fonte: O autor.

3.4 O componente gerador

O componente gerador é responsável por transformar o metadado gerado, representando um diagrama de transição de estados ou uma MEF, em um modelo SMV pronto para ser utilizado pelo verificador de modelo NuSMV. Essa etapa do método é feita pela variação do método HiMoST (DE SANTIAGO JÚNIOR; DA SILVA, 2017). O método HiMoST tem como entrada um modelo descrito por um *Statecharts* (HAREL, 1987), e o mesmo converte de Statecharts para uma estrutura geral baseada na linguagem do Verificador de Modelo NuSMV. Além disso, as propriedades são formalizadas em CTL por meio de Padrões de Especificação (DWYER et al., 1999) e um algoritmo de Teste de Interação Combinatória, chamado TTR (BALERA; SANTIAGO JÚNIOR, 2017; BALERA; DE SANTIAGO JÚNIOR, 2015). O componente gerador aqui proposto apresenta uma variação desse método, onde a entrada é o diagrama de transição de estados gerado pelo componente extrator, que se diferencia do Statecharts por não apresentar histórico nem paralelismo. Também é diferente a forma de geração das propriedades CTL que, ao invés de utilizarem os padrões de especificação de Dwyer combinados com os estados do diagrama pelo algoritmo TTR, aqui são gerados por propriedades armadilha encontradas no trabalho de Fraser et al. (2009). Diferentemente da abordagem HiMoST, que usa os padrões de especificação para gerar contraexemplos pelo verificador de modelo, as propriedades armadilha utilizadas fazem uma simples negação dos eventos, decisões e transições para geração de contraexemplos, na busca da geração de um grande número de contraexemplos por classe. O processo completo executado pelo componente gerador é dividido em cinco funções que podem ser vistas no Algoritmo 4.

Algoritmo 4 gerador

entrada: estados, transições
saída: modelo.smv

1: variáveis ← criarVariáveis(estados, transições)
2: iniciais ← criarEstadosIniciais(estados, transições)
3: próximos ← criarPróximos(iniciais, estados, transições)
4: propriedades ← criarPropriedades(variáveis, estados, transições)
5: **return** modelo.smv ← gerarSMV(variáveis, iniciais, próximos, propriedades)

O processo executado por cada função dentro do componente reflete a estrutura de processos sugerida pelo HiMoST, que segue a ordem dos elementos que aparecem em um arquivo SMV:

- Definição das variáveis a partir dos estados e transições;
- Definição dos estados iniciais de cada variável;
- Definição das regras de transição de cada estado para seu próximo.

Tendo esse modelo, as propriedades CTL a serem testadas são geradas e por fim o arquivo SMV final é criado.

3.4.1 Definição das variáveis

Em um arquivo SMV os estados do modelo a serem testados são definidos através de variáveis. Uma variável pode assumir diferentes valores dentro de um conjunto definido pelo usuário. Para descrever o modelo gerado pelo método, serão utilizadas três variáveis:

- **state**: Estados do diagrama marcados por um identificador único
- **events**: Eventos não booleanos disparados durante as transições de estados
- **decision**: Eventos booleanos disparados durante as transições de estados

Somente os estados marcados com um identificador único são considerados na criação das variáveis. Isso garante que o modelo testado pelo verificador de modelos NuSMV possa ser considerado um GFC. Observando os estados do diagrama descrito em XML na Figura 3.3, são recolhidos 22 estados que, no arquivo SMV ficam:

```
state : {if_1, if_2, return_3, else_if_4, return_5, else_6,
return_7, else_8, return_9, if_10, return_11, if_12, return_13,
if_14, return_15, if_16, return_17, if_18, return_19, if_20,
return_21, return_22};
```

Durante uma transição de estados, eventos podem ser disparados. Uma estrutura de decisão ou um laço de repetição disparam um evento booleano, isto é, "TRUE" ou "FALSE". Retornos de funções, saídas em tela, exceções e outros cenários do código podem disparar eventos não booleanos. Esses eventos são colocados na variável "events":

```
events : {null, Equilateral, Scalene, Isosceles, Invalid};
```

Um evento nulo ("null") é criado para ser o estado inicial da variável evento, isso diz ao verificador de modelo que no início da simulação nenhum evento significativo foi disparado. Para garantir que o modelo gerado seja um GFC, os eventos "lambda" não são considerados. Caso existam eventos booleanos no modelo em teste, é definida a variável "decision":

```
decision : boolean;
```

3.4.2 Definição dos estados iniciais

Cada uma das três variáveis tem um estado inicial. A variável "state" tem como estado inicial o primeiro estado visitado pelas transições que possui um identificador único positivo. A variável "events" sempre vai ser iniciada como um evento nulo ("null") e a variável "decision" não é iniciada, seu valor é não determinístico. Seguindo o exemplo do problema do triângulo, a inicialização das variáveis no arquivo SMV fica:

```
init(events) := null;
init(state) := if_10;
```

3.4.3 Definição das regras de transição

Assim como cada uma das três variáveis precisa ser inicializada, é preciso definir as regras de transição para cada uma delas. Para variável "state", todas as transições geradas pelo componente extrator são iteradas, para cada transição entre dois estados com identificado único positivo, uma regra de transição é gerada. Isso exclui transições entre os estados fictícios "initial" e "final". Transições entre estados que dependam de um evento booleano são acrescidos de uma condição de guarda de acordo com o estado da variável "decision". Em linguagem SMV, as transições entre estados ficam:

```
next(state) :=
    case
        (state = if_10 & decision = TRUE) : return_11;
        (state = if_10 & decision = FALSE) : if_12;
        (state = if_12 & decision = TRUE) : return_13;
        (state = if_12 & decision = FALSE) : if_14;
        (state = if_14 & decision = TRUE) : return_15;
```

```
        (state = if_14 & decision = FALSE) : if_16;
        (state = if_16 & decision = TRUE) : return_17;
        (state = if_16 & decision = FALSE) : if_18;
        (state = if_18 & decision = TRUE) : return_19;
        (state = if_18 & decision = FALSE) : if_20;
        (state = if_20 & decision = TRUE) : return_21;
        (state = if_20 & decision = FALSE) : return_22;
        (state = return_11) : if_1;
        (state = return_13) : if_1;
        (state = return_15) : if_1;
        (state = return_17) : if_1;
        (state = return_19) : if_1;
        (state = return_21) : if_1;
        (state = return_22) : if_1;
        (state = if_1 & decision = TRUE) : if_2;
        (state = if_2 & decision = TRUE) : return_3;
        (state = if_2 & decision = FALSE) : else_if_4;
        (state = else_if_4 & decision = TRUE) : return_5;
        (state = else_if_4 & decision = FALSE) : else_6;
        (state = else_6 & decision = TRUE) : return_7;
        (state = if_1 & decision = FALSE) : else_8;
        (state = else_8 & decision = TRUE) : return_9;
        TRUE : state;
    esac;
```

3.4.4 Criação das propriedades CTL

A geração das propriedades, mostrada no Algoritmo 5, tem como entrada os estados e transições. O trabalho de Fraser et al. (2009) sugere diversas propriedades para geração de casos de teste baseados na cobertura do sistema testado. Essas propriedades, chamadas propriedades armadilha, aplicam uma negação lógica para cada critério de um modelo, forçando a cobertura do caminho percorrido por esse critério no dado modelo. Diferente das propriedades usadas pelo método HiMoST (DE SANTIAGO JÚNIOR; DA SILVA, 2017) que buscam filtrar a geração de contraexemplos em busca do contraexemplo mais relevante, as propriedades aqui escolhidas procuram maximizar a geração de contraexemplos visando obter a maior quantidade possível por meio de negação simples de cada estado ou transição, forçando dessa forma uma

maior cobertura do programa sob teste. Das propriedades armadilha apresentadas, três delas foram escolhidas para cobertura do modelo gerado pelo método: A negação dos eventos, a negação das transições e a negação das condições de guarda. Essas propriedades são nomeadas como caso 1, caso 2 e caso 3:

- **Caso 1**: Negação dos eventos;
- **Caso 2**: Negação das transições booleanas;
- **Caso 3**: Negação das transições não booleanas (condições de guarda);

Algoritmo 5 criarPropriedades

entrada: variáveis, transições
saída: propriedades

```
1:  for ∀ eventos ∈ variáveis.event do
2:      propriedades ← gerarPropriedadeDePrimeiroCaso(evento)
3:  end for
4:  for ∀ transição ∈ transições do
5:      if transição é booleana then
6:          propriedades ← gerarPropriedadeDeSegundoCaso(transição)
7:      else
8:          propriedades ← gerarPropriedadeDeTerceiroCaso(transição)
9:      end if
10: end for
11: return propriedades
```

Para criação das propriedades do primeiro caso, a lista de eventos do modelo é iterada e para cada evento uma propriedade é gerada. Essa propriedade é dada pela Equação 3.1, que diz que "sempre, globalmente, o próximo evento é diferente de e_i, onde i pertence aos eventos".

$$\forall \Box \neg (evento = e_i), i \in eventos \tag{3.1}$$

Negar cada valor assumido pelos eventos durante a execução do código gera um contraexemplo para cada evento com um caminho que leva à confirmação desse evento. Cada contraexemplo aqui gerado se torna um caso abstrato de teste.

No exemplo do triângulo, as propriedades geradas são muito numerosas e não necessitam serem mostradas na íntegra aqui, apenas dois exemplos de cada caso serão

usados para fins ilustrativos. Para propriedade do caso 1, uma negação de cada evento é realizada, por exemplo, os eventos "Equilateral" e "Scalene" geram as propriedades:

```
CTLSPEC
   AG (events != Equilateral)
CTLSPEC
   AG (events != Scalene)
```

Após isso, todas as transições são visitadas e separadas em booleanas e não booleanas. Transições booleanas geram propriedades do segundo caso, mostrada na Equação 3.2, que diz que "sempre, globalmente, estando em um estado α e ocorrendo o valor booleano de transição γ, o próximo estado será diferente de β".

$$\forall \Box \alpha \wedge \gamma \rightarrow \exists \bigcirc \neg \beta \tag{3.2}$$

A propriedade do caso 2 usa os valores de uma transição booleana afirmando que, ao se encontrar em um determinado estado de origem e um determinado valor booleano de transição ocorrer, o próximo estado será diferente do estado de destino. Essa propriedade é escrita duas vezes para cada transição, uma vez com o valor de decisão "FALSE" e outra com o valor "TRUE". Com isso, pelo menos uma das duas propriedades irá gerar um contraexemplo para uma dada transição.

No exemplo do triângulo, observa-se o estado "if_10" que mediante um evento "TRUE" sofre uma transição para o estado "return_11", caso um evento "FALSE" sua transição leva ao estado "if_12". A propriedade do caso 2 irá negar essas informações dizendo que a partir do estado "if_10", não existe um próximo estado "return_11" diante um evento "TRUE" e não existe um próximo estado "if_12" diante um evento "FALSE":

```
CTLSPEC
    AG (state = if_10 & decision = TRUE -> EX state != return_11)
CTLSPEC
    AG (state = if_10 & decision = FALSE -> EX state != if_12)
```

Uma transição pode ser examinada sem se considerar o evento de forma booleana,

criando uma propriedade do caso 3. Essa propriedade contém duas sentenças: uma que nega o destino da transição e outra que nega o segundo termo na condição de guarda. Essa propriedade é mostrada na Equação 3.3, onde α é um estado de origem, β é um estado de destino e γ é um valor não necessariamente booleano de decisão.

$$\begin{cases} \forall \Box \alpha \wedge \gamma \rightarrow \exists \bigcirc \neg \beta \\ \forall \Box \alpha \wedge \neg \gamma \rightarrow \exists \bigcirc \beta \end{cases} \tag{3.3}$$

Nota-se que a primeira expressão da propriedade de caso 3 é igual à propriedade de caso 2, com a diferença que aqui ela é aplicado para qualquer valor de evento (não somente booleano) por meio de negação simples. A segunda equação, que diz que "sempre, globalmente, estando em um estado α e não ocorrendo o valor de transição γ, o próximo estado será igual a β", afirma que uma transição de um estado diferente da origem sempre irá levar ao destino, o que é imediatamente inválido para o modelo sob teste. Essa propriedade força pelo menos um contraexemplo para cada transição.

O estado "if_10" do exemplo do triângulo irá gerar quatro propriedades, duas negando o destino e e duas nengando a condição de guarda:

```
CTLSPEC
    AG (state = if_10 -> EX state != return_11)
CTLSPEC
   AG (state != if_10 -> EX state = return_11)
CTLSPEC
    AG (state = if_10 -> EX state != if_12)
CTLSPEC
   AG (state != if_10 -> EX state = if_12)
```

3.5 O componente construtor

A etapa de construção é responsável por executar o arquivo SMV gerado no componente anterior e obter os contraexemplos. É esperado que a execução do verificador de modelo gere contraexemplos, caso contrário não existirão casos de teste. A lista de contraexemplos é iterada e cada contraexemplo é um caso/dado de teste.

Foi estabelecido que somente contraexemplos com um número de estados maior que 2 serão considerados na geração de testes unitários, como sugerido pelo método

HiMoST (DE SANTIAGO JÚNIOR; DA SILVA, 2017). Cada contraexemplo é iterado quanto a seus estados. Esses estados descrevem um ponto no caminho a ser percorrido no código C++.

A ideia de utilizar o traço de execução gerado por um contraexemplo subentende que o programa a ser testado é conhecido por quem está executando os testes para que essa pessoa possa interpretar cada estado, identificar a linha de código à qual esse estado pertence, acessar essa linha e o conhecer cada parâmetro e cada condição de guarda necessária para reproduzir o caminho no código sob teste.

A partir do modelo produzido pelo método, foram criados 41 contraexemplos, sendo que 36 apresentam um numero de estados ≤ 2, o que os tornam imediatamente nulos. Dos contraexemplos restantes, um deles foi escolhido para ilustrar a geração de um teste unitário. Dado a propriedade CTL testada:

```
AG (events != Equilateral)
```

foi obtido o seguinte traço de execução como contraexemplo:

```
1
STATE = if_10
DECISION = false
2
STATE = if_12
DECISION = false
3
STATE = if_14
DECISION = false
4
STATE = if_16
DECISION = false
5
STATE = if_18
DECISION = false
6
STATE = if_20
DECISION = false
7
```

```
STATE = return_22
DECISION = false
8
STATE = if_1
DECISION = true
9
STATE = if_2
DECISION = true
10
STATE = return_3
DECISION = true
11
EVENT = Equilateral
DECISION = true
```

O traço de execução inicia com o estado "if_10", dentro da função privada "is_-valid" da classe "Tool". Essa função é chamada por uma instrução "if" dentro da função pública "classify", que recebe como parâmetro um objeto do tipo "Triangle". As duas primeiras instruções de um teste unitário baseado nesse contraexemplo precisam instanciar objetos dos tipos "Tool" e "triangle":

```
Tool tool;
Triangle triangle;
```

Como o valor da variável "DECISION" é falso nesse estado e a condição de guarda da estrutura "if" espera um valor menor ou igual à zero para o parâmetro "a" do objeto "Triangle", definimos que esse parâmetro terá um valor positivo arbitrário:

```
triangle.a = 2;
```

O segundo estado mostrado no contraexemplo pede um valor falso para variável "DECISION" em uma estrutura "if" onde a condição de guarda pede um valor menor ou igual à zero para o parâmetro "b" do objeto "Triangle", logo, esse parâmetro também recebe um valor positivo arbitrário:

```
triangle.b = 2;
```

O mesmo ocorre no terceiro estado do contraexemplo, porém para o parâmetro "c" do objeto "Triangle":

```
triangle.c = 2;
```

Os estados quatro, cinco e seis do contraexemplo pedem um valor falso para a variável "DECISION" onde as três condições de guarda correspondentes no código C++ pedem para que a soma de duas variáveis seja menor ou igual a terceira variável. Com os valores arbitrários escolhidos nas etapas anteriores essas condições já são satisfeitas sendo assim possível continuar iterando os estados do contraexemplo.

O sétimo estado do traço nos mostra uma instrução de retorno que muda o valor da variável "DECISION" para verdadeiro, o que já resolve o oitavo estado fazendo com que a primeira instrução "if" da função "classify" seja verdadeira.

O nono estado pede para que a segunda instrução "if" da função "classify" resulte em "true". A condição de guarda dessa condicional pede para que o valor do parâmetro "a" do objeto "Triangle" seja igual ao valor dos parâmetros "b" e "c", o que já esta satisfeito pelos valores escolhidos para esses parâmetros nos passos anteriores.

O décimo estado do contraexemplo leva a uma instrução de retorno que muda o valor da variável "EVENT" para "Equilateral", chegando ao estado de número onze do contraexemplo onde a propriedade CTL testada é violada.

O teste unitário em C++ baseado nesse contraexemplo fica:

```
Tool tool;
Triangle triangle;
triangle.a = 2;
triangle.b = 2;
triangle.c = 2;
```

A execução dessas instruções deve reproduzir o caminho previsto pelo contraexemplo e exercitar a parte do código coberta pelo mesmo. De acordo com a hipótese formulada, quanto mais contraexemplos forem executados, maior a chance de se obter uma boa cobertura do código sob teste.

3.6 Considerações finais

O capítulo mostrou a aplicação do método Singularity em um exemplo de programa em C++ denominado "O problema do triângulo". Cada passo de execução do método foi ilustrado com a computação linha a linha do exemplo escolhido. Foram apresentadas uma descrição detalhada da execução de cada um dos três componentes do método: Extrator, Gerador e Construtor. No primeiro componente, foi definido o metadado XML que representa um modelo do código fonte original como uma máquina de estados finitos, seguida da descrição minuciosa das sub-rotinas de extração de estados e transições. Em seguida, o componente gerador, baseado no método HiMoST (DE SANTIAGO JÚNIOR; DA SILVA, 2017) e na propriedades definidas por Fraser et al. (2009), foi descrito e exemplificado dês da definição dos estados e transições até a elaboração de cada propriedade armadilha. Por fim, o último componente traz o resultado final obtido pela execução do Verificado de Modelos exibindo um contraexemplo gerado a partir do Problema do Triângulo e ilustrando o uso desse caso de teste abstrato em um teste unitário.

4 AVALIAÇÃO EXPERIMENTAL DO MÉTODO SINGULARITY

O método proposto no presente trabalho foi implementado na forma de uma ferramenta Java, permitindo assim a realização de uma avaliação experimental de seu desempenho. Cada programa C++ aplicado à ferramenta gerou um conjunto de contraexemplos e dados estatísticos sobre o funcionamento do método. Foram utilizados para essa avaliação classes escolhidas do programa GeoDMA (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, a) e da plataforma TerraLib (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, b), descritos brevemente no Capítulo 1. Os dados obtidos foram comparados por meio de gráficos e tabelas para se obter uma visão do comportamento do método diante de programas do mundo real.

4.1 A ferramenta

Foi desenvolvida uma ferramenta Java que implementa o método Singularity como prova de conceito. Os resultados obtidos com essa ferramenta serão considerados aqui como uma avaliação do método em si. O código fonte da ferramenta está disponível no GitHub através do link `https://github.com/eduardoeras/Singularity`.

A ferramenta é composta por quatro módulos, que refletem o método proposto no capítulo 3, acrescido de um módulo de leitura:

- **Módulo Leitor**: Responsável pela leitura do arquivo C++ de entrada e conversão do mesmo em uma estrutura de dados do tipo árvore em Java. É utilizado a ferramenta Antlr (PARR, 2019) para essa transformação. A saída desse módulo é uma estrutura Java contendo os elementos do código C++ de entrada;

- **Módulo Extrator**: Responsável por extrair os estados e transições da estrutura Java que representa o código original. É o módulo mais complexo da ferramenta;

- **Módulo Gerador**: Processa os estados e transições obtidos para montar um GFC no formato de um arquivo SMV, pronto para ser utilizado pelo verificador de modelos NuSMV. Esse módulo também é responsável pela criação das propriedades armadilha utilizadas para obter contraexemplos.

- **Módulo Construtor**: Executa o verificador de modelo e obtém os contraexemplos. Aqui são feitas as filtragens dos contraexemplos válidos e a obtenção dos dados estatísticos sobre a execução do método.

A ferramenta apresenta algumas limitações: (i) o método não suporta sobrecarga de funções e operadores, arquivos que contém essa característica não são aceitos pela ferramenta; (ii) também não são suportados arquivos que contenham linhas de código fora do escopo de uma classe. Ao tentar executar um arquivo que viole essas limitações, a ferramenta não completa sua execução e exibe uma mensagem de erro.

4.2 Validação da medida de complexidade

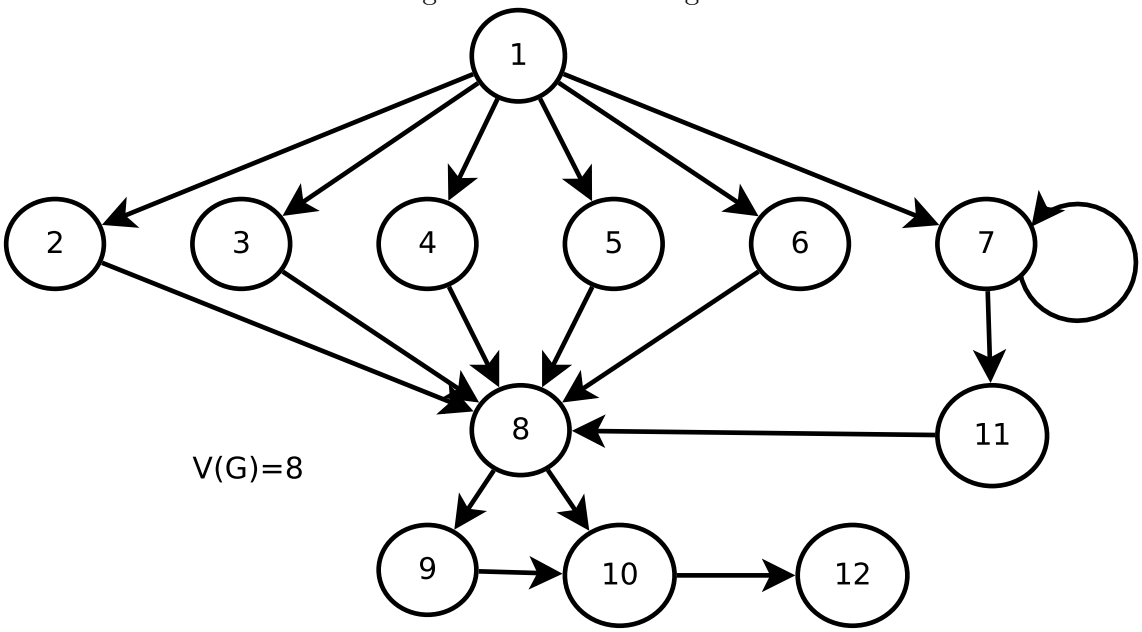
O critério de complexidade ciclomática visto na Sessão 2.5 dá o número mínimo de caminhos a serem percorrido para se obter uma cobertura completa de um GFC, onde cada um desses caminhos pode ser interpretado como um caso de teste. Esse critério pode ser utilizado como uma métrica importante na avaliação do método, para isso ele precisa ser implementado junto à ferramenta e validado.

Na posse dos valores de complexidade do código testado, é possível determinar se o número de casos de teste gerado foi pelo menos igual ao número de casos esperado mínimo para cobrir todos os caminhos. Mesmo que não existam garantias de que os testes gerados cubram caminhos diferentes, é possível verificar se o método está gerando um valor próximo ao número de complexidade, que seria suficiente para cobrir todo Programa Sob Teste (PST).

A publicação original de Mccabe (1976) sobre a complexidade ciclomática mostra treze grafos de fluxo de controle acompanhados pelo seu valor de complexidade $V(G)$ em sua sessão central. Esses grafos refletem o padrão de programação da época que fazia o uso da instrução `GO TO`, o que permitia saltos complexos no fluxo de execução. Para reconstruir um programa em C++ que reflita esses grafos sem o uso dessa instrução de salto, algumas modificações precisaram ser feitas nesses grafos: caminhos que levavam para fora de um laço ou para estados anteriores foram substituídos por uma transição para o próprio estado, que podem ser facilmente simuladas com um laço "enquanto". Essas modificações não alteraram o valor de complexidade ciclomática do grafo original. A publicação, de mais de quarenta anos atrás, também possui diversos erros de digitação, logo todos os valores de complexidade foram recalculados de acordo com os grafos para corrigir esses erros.

Por exemplo, o GFC visto na Figura 4.1, teve uma alteração no estado de número 7, onde em sua versão original continha uma transição para o estado 2, muito difícil de se reproduzir sem utilizar uma instrução `GO TO`. Essa transição foi substituída por uma transição para o próprio estado. Essa alteração não muda o número de complexidade ciclomática do grafo pois não altera o número de estados e transições.

Figura 4.1 - GFC corrigido.



Fonte: Adaptado de Mccabe (1976)

O grafo da Figura 4.1 pode ser reconstruído em linguagem C++ da seguinte forma:

```
#include<iostream>

class CaseSix {
public:
    void function (bool condition)
    {
        switch (condition) //one
        {
        case 2 :
            std::cout << "two" << std::endl;
            break;
        case 3 :
            std::cout << "three" << std::endl;
            break;
        case 4 :
            std::cout << "four" << std::endl;
            break;
        case 5 :
            std::cout << "five" << std::endl;
```

```
            break;
        case 6 :
            std::cout << "six" << std::endl;
            break;
        default :
            while(condition) //seven
            {
            }
            std::cout << "eleven" << std::endl;
            break;
        }
        if (condition) //eight
        {
            std::cout << "nine" << std::endl;
        }
        std::cout << "ten" << std::endl;
        std::cout << "twelve" << std::endl;
    }
};
```

Dessa forma, todos os treze exemplos do trabalho original de McCabe foram reconstruídos e testados pela ferramenta. Todos os valores de Complexidade Ciclomática foram reproduzidos com sucesso, validando a implementação proposta neste trabalho.

4.3 Avaliação experimental

Com o objetivo de avaliar o método Singularity, são definidas como bom desempenho as seguintes métricas:

- **Um bom número de contraexemplos**: Um maior número de contraexemplos significa uma maior possibilidade de criar testes unitários e consequentemente uma melhor chance de se obter uma boa cobertura do código;
- **Contraexemplos eficientes**: Um contraexemplo que mostra um caminho óbvio ou muito curto no código nem sempre tem muita utilidade. Um contraexemplo que descreve um caminho mais complexo é mais interessante para criação de um teste unitário. Esse tipo de contraexemplo é geralmente

associado à estruturas de decisão do código.

O método foi executado para dois cenários como forma de avaliação, GeoDMA e TerraLib. Para o primeiro caso de testes, algumas classes foram selecionadas manualmente, já para o segundo todas as classes compatíveis foram testadas. A descrição de cada cenário segue:

- **GeoDMA**: Foram escolhidas criteriosamente 24 classes com um tamanho entre 50 e 350 estados. Essas classes formam um "cenário otimizado": têm um número de estados grande suficiente para gerar bastante contraexemplos mas não tantos que a cobertura seja impraticável, além de terem muitas estruturas de decisão o que ajuda a gerar contraexemplos eficientes. Se espera encontrar um bom desempenho do método dessa forma.
- **TerraLib**: O pacote TerraLib é formado por 1466 classes, de onde 1063 foram compatíveis com o método, isso é, não continham sobrecarga de funções e operadores, nem nenhuma linha de código fora do escopo de uma classe. Esse teste forçou o método a processar todas as classes compatíveis sem nenhuma pré seleção e em grande quantidade. É um teste onde se espera encontrar fraquezas no desempenho no método, isso é, geração de muitos contraexemplos pequenos e óbvios ou mesmo uma quantidade muito pequena de contraexemplos por classe.

4.3.1 Descrição dos dados obtidos

Juntamente com os contraexemplos, a ferramenta também gera dezenove parâmetros que qualificam o PST e os resultados obtidos. Esses parâmetros serão avaliados nesse capítulo para se obter uma avaliação de desempenho do método.

Os três primeiros dados descrevem fisicamente o PST. Esses parâmetros dão o número de estados (instruções) do programa, o número de eventos disparados e o número de decisões:

- **Estados**: Número total de estados (nós do GFC) do PST;
- **Eventos**: Número de eventos não booleanos disparados pelo PST;
- **Decisões**: Número de decisões e laços (bifurcações no GFC) que disparam eventos booleanos.

Os próximos três parâmetros detalham as transições ocorridas no modelo, dividindo as transições entre transições de estado (principais) e transições causadas por eventos não booleanos (secundárias):

- **Trans. de Estado**: Número de transições entre estados (arestas do GFC);
- **Trans. de Evento**: Número de transições geradas por eventos disparados não booleanos, não faz parte do GFC;
- **Trans. Total**: Número total de transições do modelo gerado: soma das transições de estados com as transições de eventos.

O modelo do PST é um diagrama de estados e transições avaliado pelo verificador de modelos. Todos os estados e transições que aparecem no contraexemplo são considerados cobertos pelo caso de teste. Essas duas métricas podem ser consideradas como métricas de efetividade:

- **Cobertura de Estados**: Número de estados que compõe o modelo que também aparecem nos contraexemplos;
- **Cobertura de Transições**: Número de transições que compõe o modelo que também aparecem nos contraexemplos.

O número de complexidade ciclomática de cada programa depende do número de estados, transições e componentes que fazem parte desse programa. Nesse trabalho, é considerado um componente do código C++ todas as classes, estruturas, funções e operadores:

- **Componentes**: Número de componentes que compõem o PST;
- **Complexidade**: Número de complexidade ciclomática do PST.

Para se obter contraexemplos, um número de propriedades CTL é gerado de acordo com as regras definidas na Subseção 3.4.4:

- **Propriedades**: Número total de propriedades geradas (soma dos três casos);
- **Caso 1**: Número de propriedades do primeiro caso;

- **Caso 2**: Número de propriedades do segundo caso;
- **Caso 3**: Número de propriedades do terceiro caso.

Ao executar a ferramenta, é esperado que um número de contraexemplos seja gerado. Desses contraexemplos, somente os que possuem um número de estados maior que dois são considerados válidos, conforme definido na Sessão 3.5:

- **Total de Contraexemplos**: Número total de contraexemplos gerados;
- **Contraexemplos Válidos**: Número de contraexemplos válidos;
- **Contraexemplos Inválidos**: Número de contraexemplos inválidos;
- **Maior Contraexemplo**: Número de estados do maior contraexemplo válido gerado;
- **Menor Contraexemplo**: Número de estados do menor contraexemplo válido gerado.

4.3.2 Casos de teste do programa GeoDMA

As classes do GeoDMA foram escolhidas cuidadosamente para se obter um melhor desempenho do método. De acordo com a Tabela 4.1, é possível montar um retrato do cenário de teste. Das 24 classes escolhidas, existe uma média de $145,54$ estados por classe com um desvio padrão de $88,86$ estados e desvio médio de aproximadamente 73 estados. Uma amostra com o desvio padrão menor que a média e com o valor de média razoavelmente centralizado entre o menor e o maior número de estados. A quantidade de estados por classe pode ser vista na Figura 4.2. Os dados completos da execução de uma das classes do GeoDMA pode ser visto no Apendice A.

Tabela 4.1 - Características básicas das classes testadas do GeoDMA.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Estados	145,54	88,86	72,88	38	339
Eventos	6,50	3,46	2,33	0	17
Decisões	29,08	21,05	16,26	1	81
Trans. de Estado	179,50	108,32	88,42	42	415
Trans. de Evento	22,21	13,26	10,46	0	47
Trans. Total	201,71	118,68	97,24	50	459

Figura 4.2 - Número de Estados por classe da amostra do GeoDMA

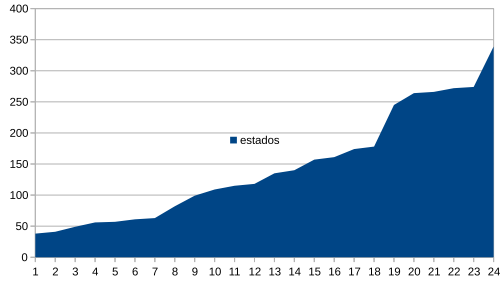


Figura 4.3 - Número de Eventos por classe da amostra do GeoDMA

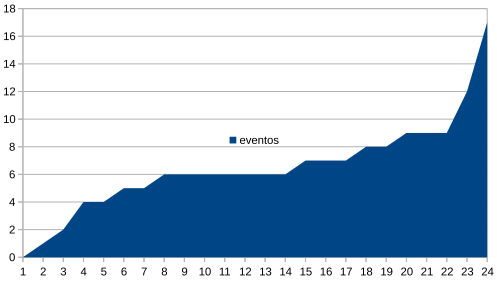
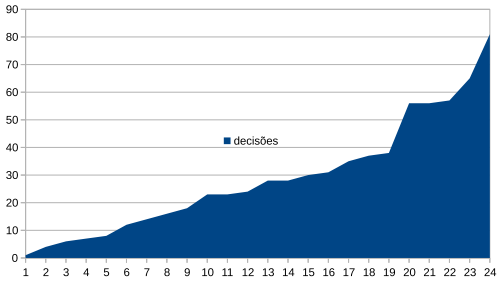


Figura 4.4 - Número de Decisões por classe da amostra do GeoDMA



A amostra tem um número máximo de 17 eventos não booleanos e média de $6,5$ eventos por classe, a quantidade de eventos é mostrada na Figura 4.3. Contendo entre 1 a 81 uma estruturas de decisão, com uma média de aproximadamente 29 decisões por classe e desvios padrão e médio abaixo dessa média, novamente um cenário equilibrado, com um número de decisões próximos a linha central de tendência. A quantidade de decisões por classe é vista na Figura 4.4. Da mesma forma, as transições de estados e eventos trazem uma média próxima a um valor central entre o mínimo e o máximo e desvios padrão e médio abaixo dessa média.

A cobertura de estados e transições, isto é, os nós e arestas do GFC que aparecem nos contraexemplos, segue uma média de aproximadamente 62% de acordo com a Tabela 4.2. Ao dividirmos o número de estados cobertos pelo número de estados totais por classe, obtemos uma porcentagem de cobertura por classe, mostrada na Figura 4.5. Para classes com até pouco mais de 100 estados, existe uma maioria de casos onde a cobertura é maior que 80%, para classes com mais estados no entanto a cobertura se torna inconsistente e em sua maioria não chega a 50%. O mesmo é observado para as transições de estado vistas na Figura 4.6.

Tabela 4.2 - Cobertura das amostras do GeoDMA.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Cobertura de Estados	62,88%	47,85	32,78	3	233
Cobertura de Transições	62,42%	48,17	33,24	2	232

Figura 4.5 - GeoDMA: Porcentagem de cobertura de estados de acordo com o número de estados por classe

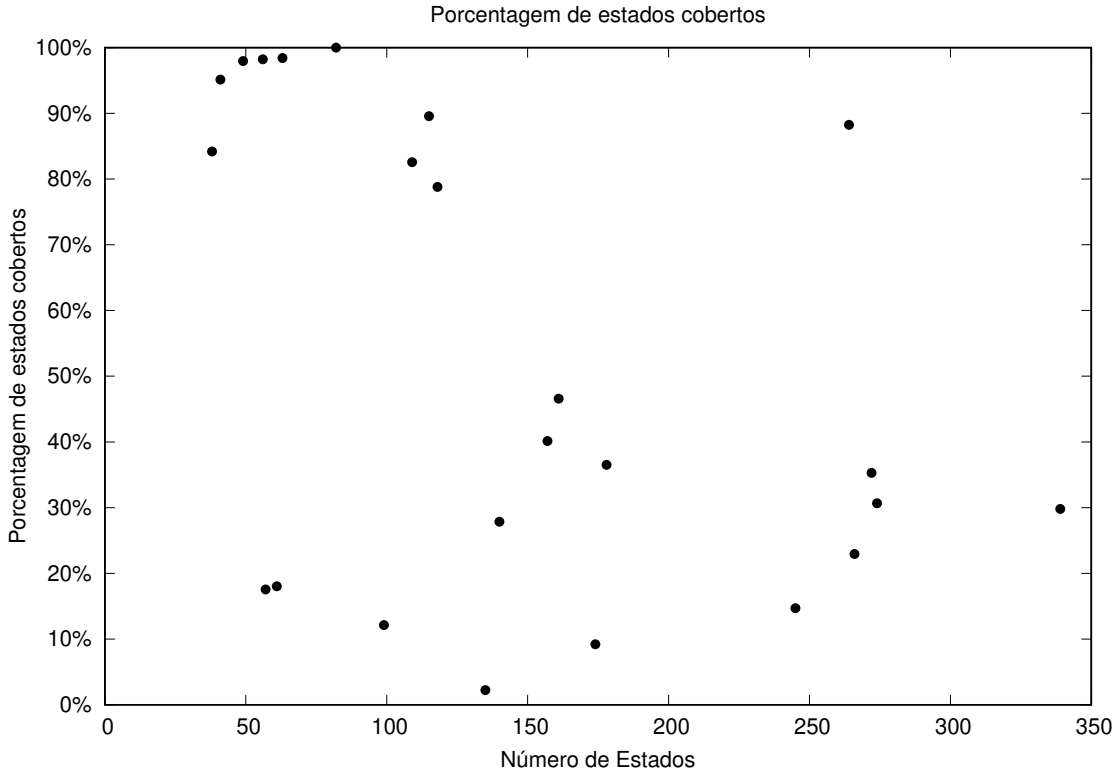
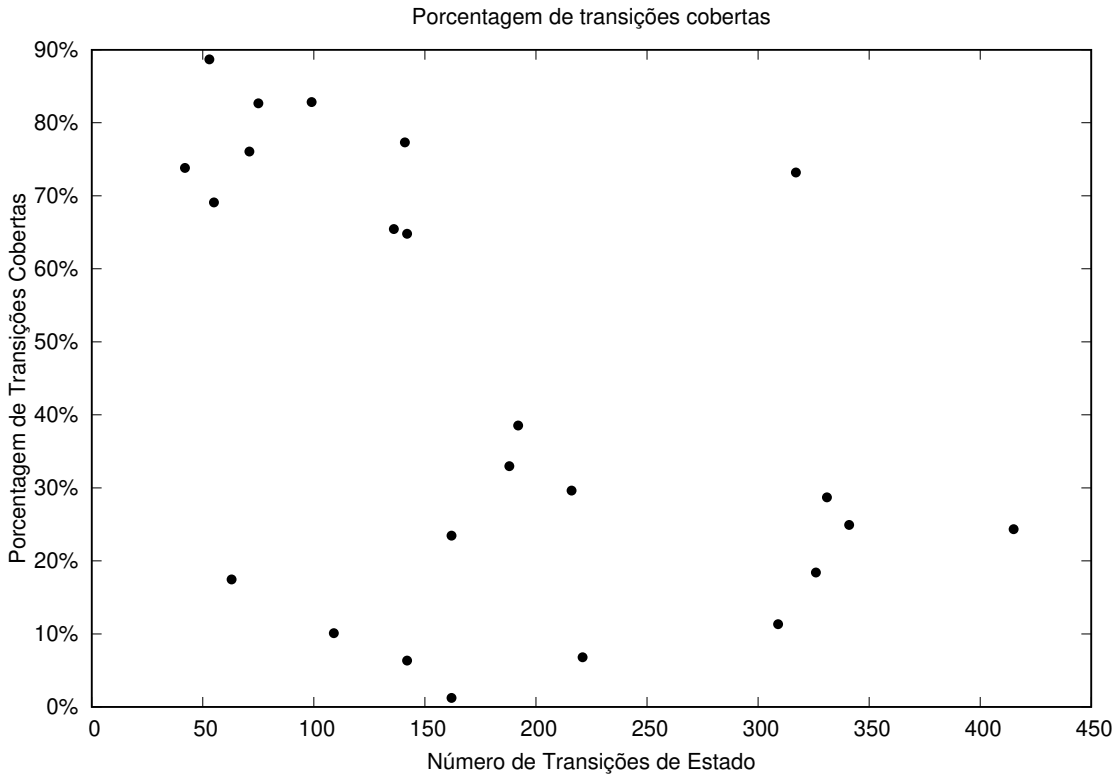


Figura 4.6 - GeoDMA: Porcentagem de cobertura das transições estados de acordo com o número de transições de estados por classe



O cálculo de complexidade ciclomática e número de componentes das amostras do GeoDMA mantém os desvios abaixo da média como mostrado na Tabela 4.3, variando a complexidade dos casos entre 17 e 125.

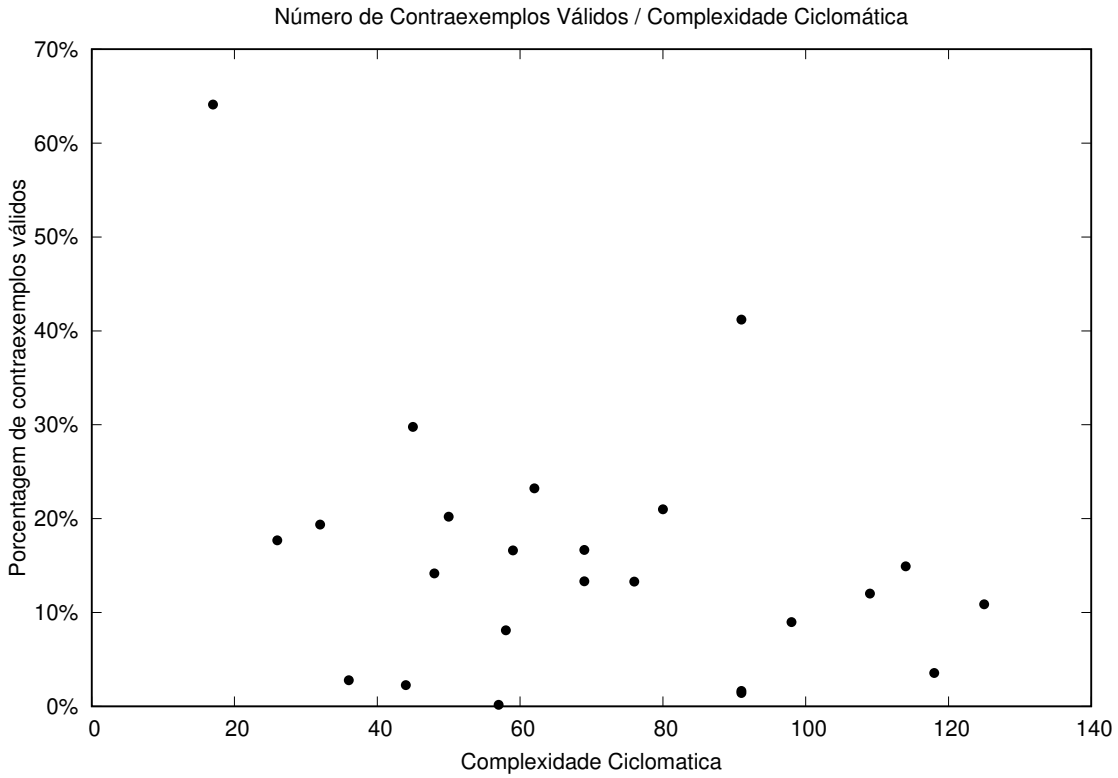
Tabela 4.3 - Compodentes e Complexidade das amostras do GeoDMA.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Componentes	17,71	6,94	4,70	1	33
Complexidade	69,38	30,25	24,94	17	125

É visto na Figura 4.7 que ao se dividir o número de contraexemplos válidos gerados por classe pela complexidade ciclomática de cada classe que na maioria dos casos a porcentagem de contraexemplos por classe não chega a 40% da complexidade ciclomática de cada classe, o que implica em uma baixa chance de cobertura dos caminhos possíveis no código mesmo diante da implementação de todos os contra -

exemplos gerados.

Figura 4.7 - GeoDMA: Porcentagem de contraexemplos válidos pela complexidade ciclomática



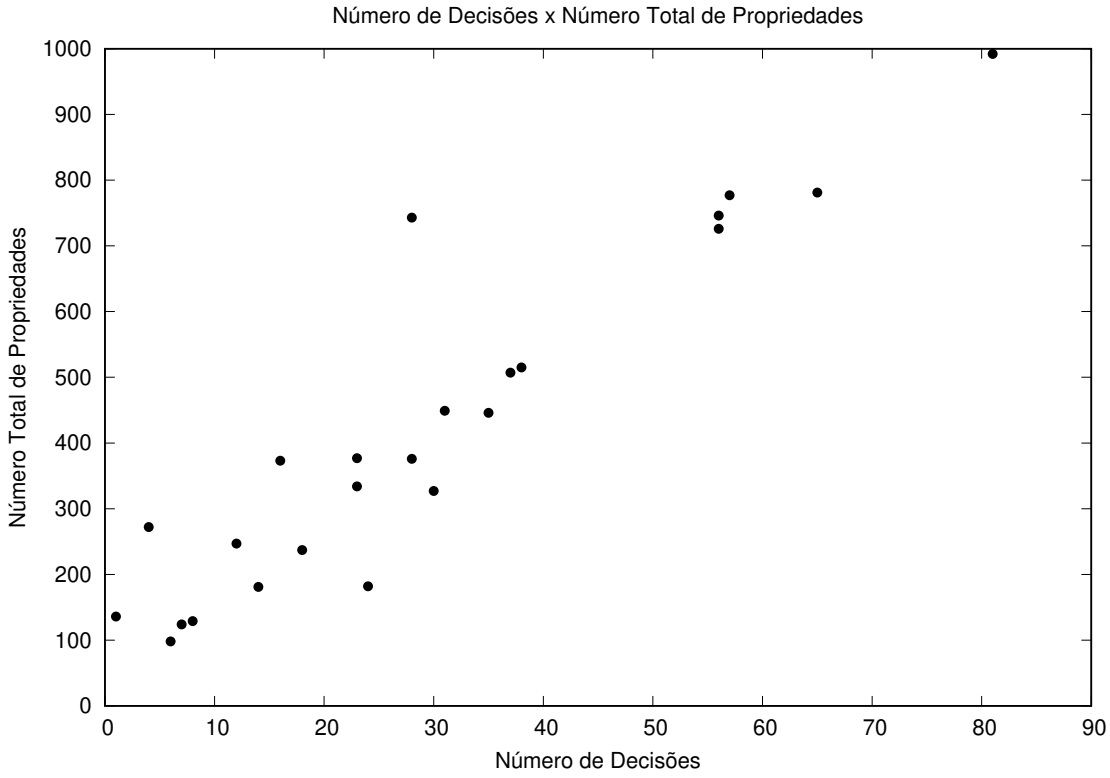
A Tabela 4.4 mostra informações sobre o número de propriedades geradas para as amostras do GeoDMA. Existe um número grande de propriedades onde a maioria delas parece pertencer ao chamado "Caso 3". Nota-se também que os desvios continuam abaixo da média para todos os parâmetros.

Tabela 4.4 - Dados das propriedades geradas pelo método a partir das amostras do GeoDMA.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Propriedades	419,79	254,10	207,01	98	992
Caso 1	6,50	3,46	2,33	0	17
Caso 2	54,13	39,98	31,15	2	155
Caso 3	359,17	215,29	174,36	84	830

Observamos na Figura 4.8 que o aumento do número de decisões parece influenciar diretamente no número de propriedades, o que é o correto a se pensar dado a geração das propriedades serem baseadas nas transições e eventos do arquivo original, como descrito na Subseção 3.4.4.

Figura 4.8 - GeoDMA: Geração de propriedades de acordo com o número de decisões por classe



Ao fazer a comparação da cobertura de estados e transições de cordo com o número de propriedades geradas obtém-se dois gráficos quase idênticos, como visto nas Figuras 4.9 e 4.10. Notamos uma leve relação entre a cobertura de estados e transições até pouco mais que 300 propriedades. Um número maior de propriedades não é consistente em sua cobertura, o que parece mostrar que o método seja mais eficiente em casos de teste menores.

Figura 4.9 - GeoDMA: Cobertura dos estados de acordo com o número de propriedades geradas por classe

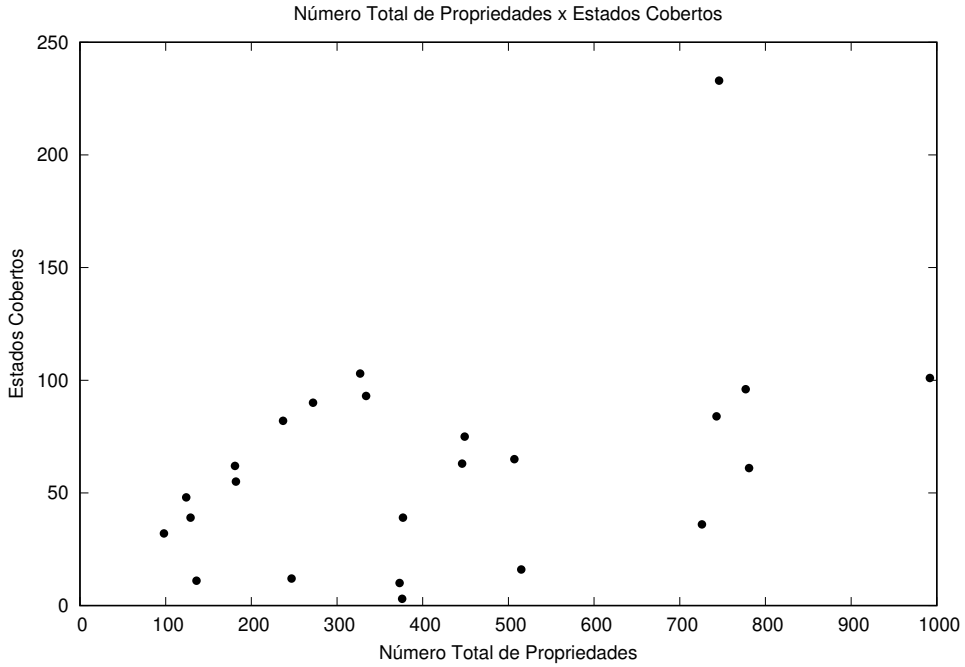
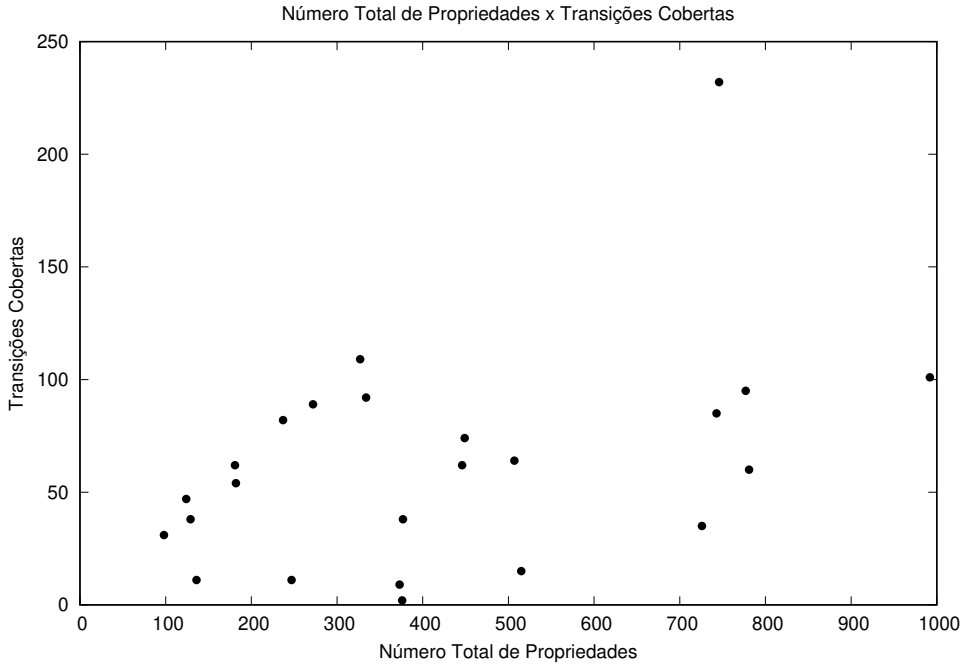


Figura 4.10 - GeoDMA: Cobertura das transições de acordo com o número de propriedades geradas por classe



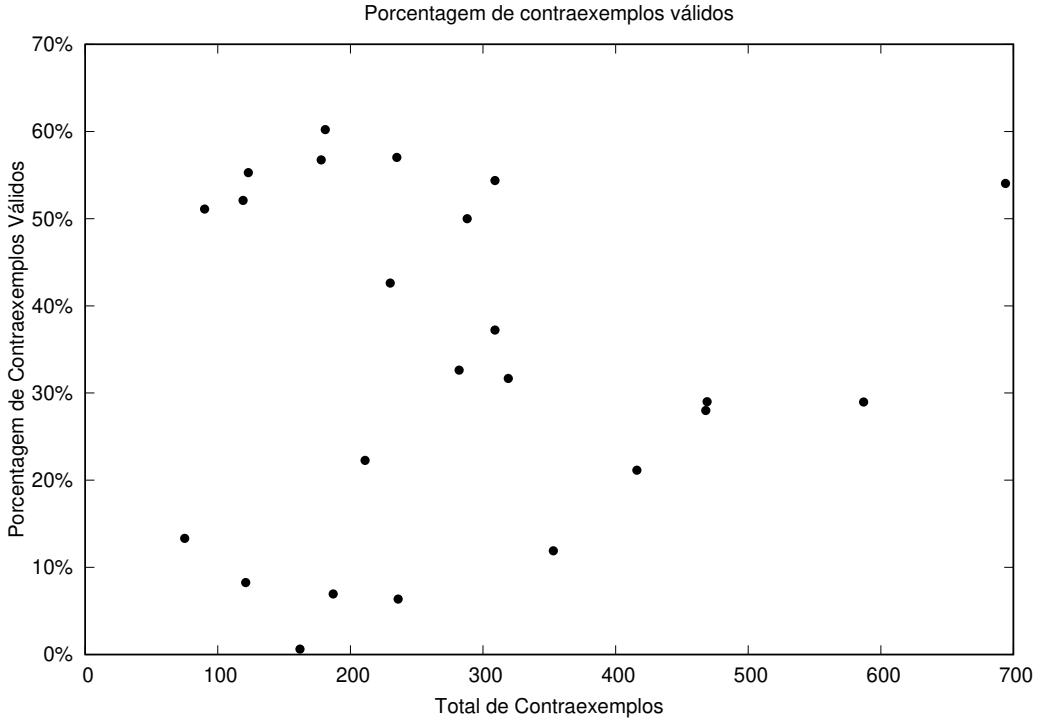
A execução das propriedades e do modelo criado geram contraexemplos. Os dados sobre a geração de contraexemplos são mostradas na Tabela 4.5. Observa-se todos os desvios menores que a média em todos os parâmetros.

Tabela 4.5 - Dados da geração de contraexemplos baseada nas amostras do GeoDMA.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Total de Contraexemplos	276,75	157,64	120,81	75	694
Contraexemplos Válidos	94,83	78,78	53,67	1	375
Contraexemplos Inválidos	181,92	107,84	87,40	44	417
Maior Contraexemplo	42,04	28,11	21,13	3	127
Menor Contraexemplo	3,00	0,00	0,00	3	3

O primeiro dado que chama a atenção no entanto é que a média de contraexemplos inválidos é quase o dobro da média de contraexemplos válidos. Na Figura 4.11 é visto que a porcentagem de contraexemplos válidos chega a ser próxima de 60% em até aproximadamente 300 contraexemplos. Um número maior de contraexemplos não garante a geração de contraexemplos válidos.

Figura 4.11 - GeoDMA: Porcentagem de contraexemplos válidos gerados



A figura 4.12 mostra uma relação linear entre o aumento no número de estados e o aumento no número de contraexemplos inválidos. Esse resultado sugere uma avaliação da forma de como os contraexemplos são gerados. Olhando as propriedades CTL responsáveis pela criação desses contraexemplos, observa-se na Figura 4.13 que um grande número de propriedades não gera necessariamente bons contraexemplos, uma tendencia linear é observada até aproximadamente 300 propriedades, depois desse valor a geração de contraexemplos válidos é inconsistente. A Figura 4.14 mostra uma relação bem clara entre a o aumento no número de propriedades e o aumento no número de contraexemplos inválidos.

Figura 4.12 - GeoDMA: Número de contraexemplos inválidos de acordo com o número de estados por classe

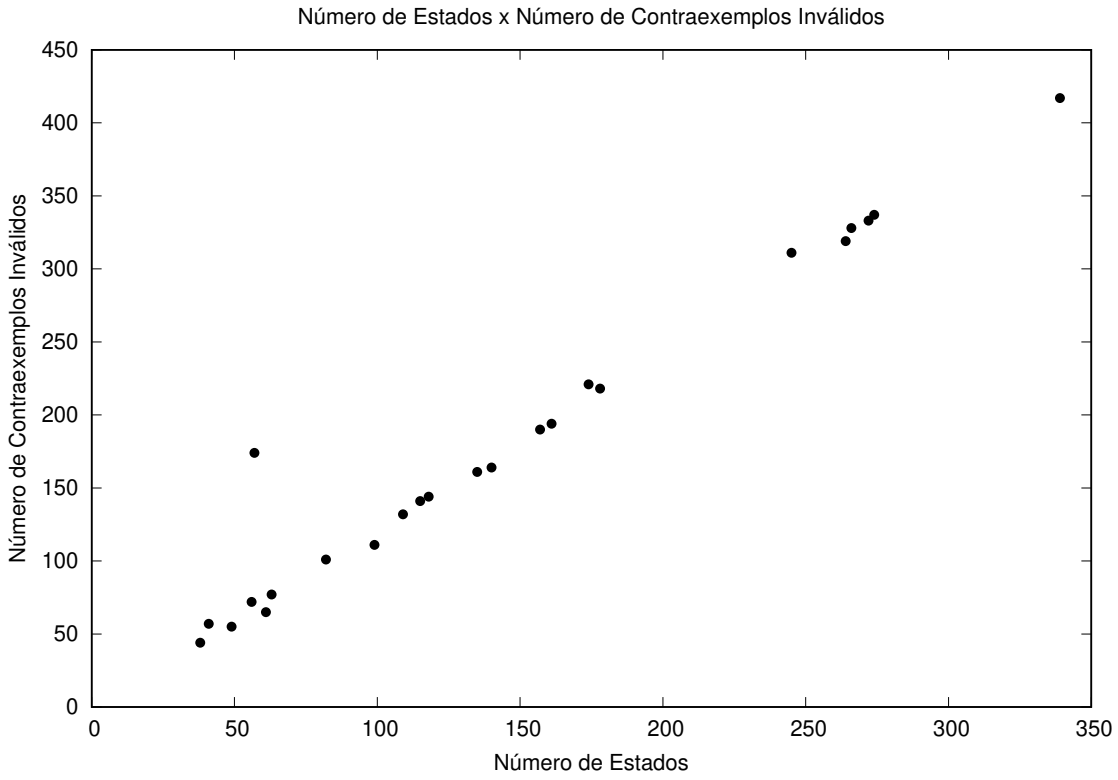


Figura 4.13 - GeoDMA: Número de contraexemplos válidos de acordo com o número de propriedades geradas por classe

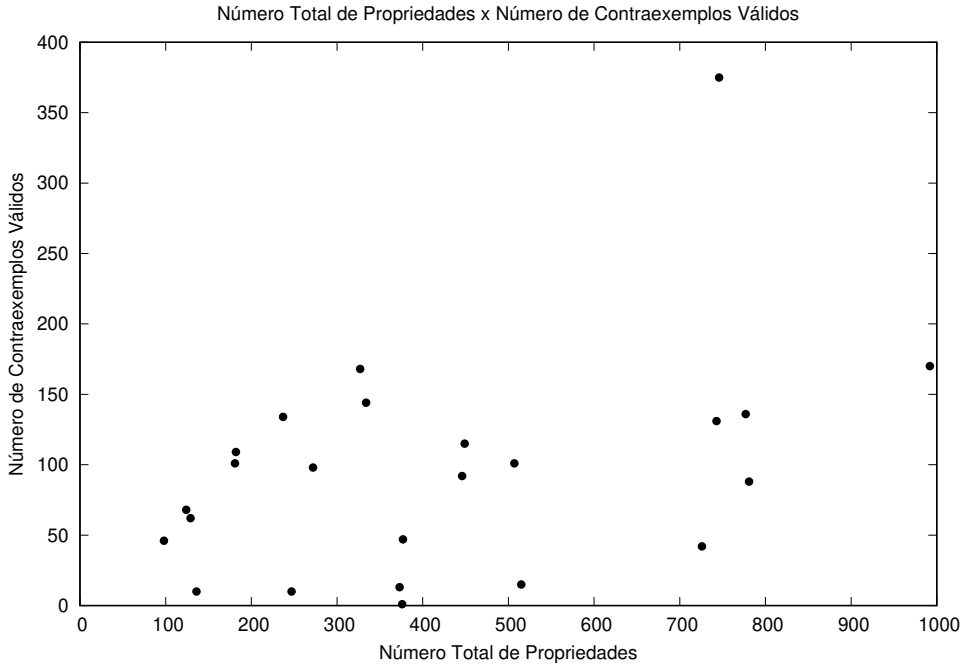


Figura 4.14 - GeoDMA: Número de contraexemplos inválidos de acordo com o número de propriedades geradas por classe

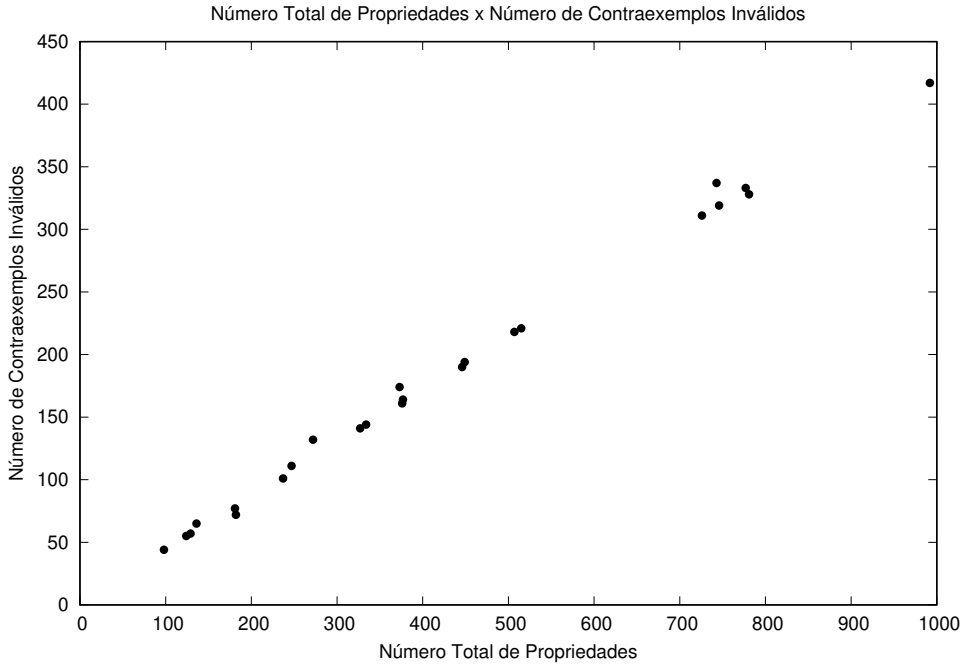
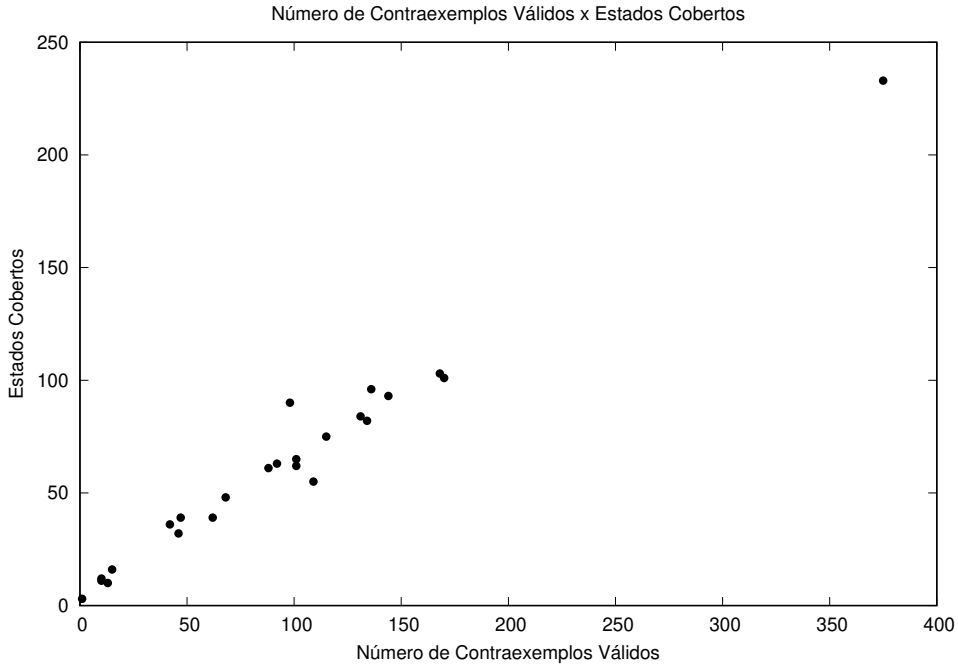


Figura 4.15 - GeoDMA: Cobertura de estados de acordo com o número de contraexemplos válidos geradas por classe



No entanto, ao considerarmos somente os contraexemplos válidos gerados, é obtido uma boa relação entre o número desses contraexemplos por classe e a quantidade de estados e transições cobertas, como visto nas Figuras 4.15 e 4.16. Esses dois últimos resultados são positivos para o método pois mostram uma bom desempenho dos casos de teste na cobertura dos nós e transições do GFC gerado.

Pode-se utilizar o argumento de que implementar testes unitários para todos os contraexemplos válidos gerados não seja prático. Para isso, vamos avaliar o desempenho do maior contraexemplo gerado para cada classe. A Figura 4.17 mostra que ao dividirmos o número de estados do maior contraexemplo pelo número de estados da classe testada, em 9 casos foram obtidas coberturas maiores ou iguais a 50% dos estados mesmo utilizando esse único contraexemplo (o maior).

4.3.3 Casos de teste da ferramenta TerraLib

As classes do TerraLib foram testadas em sua totalidade, sem nenhum critério de seleção. Todas as 1063 classes compatíveis com o método foram executadas. Essa amostra contém uma variedade muito grande de classes sem nenhuma pré seleção. É visto na Tabela 4.6 que os dados não são nada comportados, tendo todos os desvios

Figura 4.16 - GeoDMA: Cobertura das transições de acordo com o número de contraexemplos válidos geradas por classe

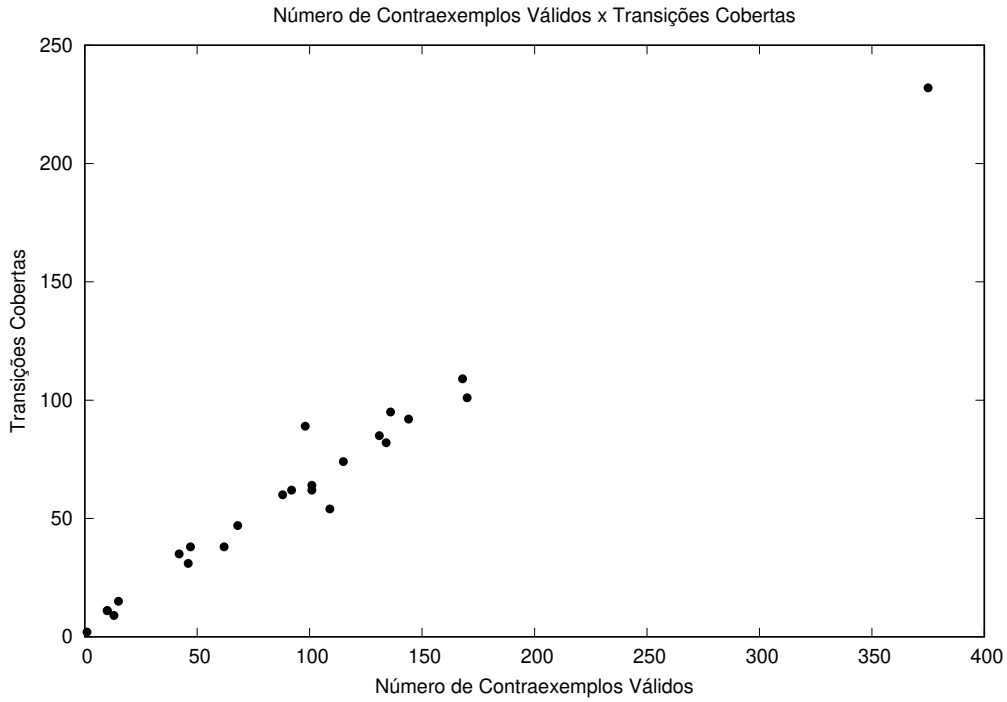
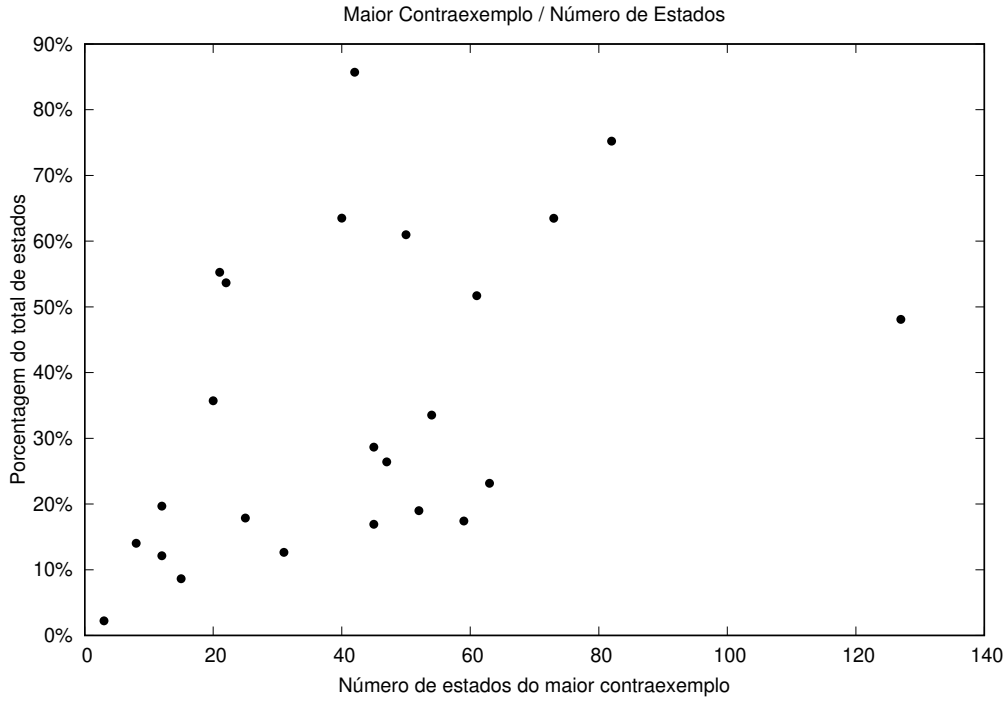


Figura 4.17 - GeoDMA: Porcentagem de cobertura de estados pelo maior contraexemplo



padrão maiores que a média, sendo alguns deles próximos do dobro. Classes com um único estado até classes com mais de oitocentos estados compõe o cenário de teste. A Figura 4.18 mostra a quantidade de estados por classe. Observa-se que a maioria das classes possuem até 100 estados, mas existe um rápido aumento no número de estados que leva a valores extremos em uma pequena amostra da população.

Tabela 4.6 - Características básicas das classes do TerraLib.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Estados	58.44	92,49	57.04	1	808
Eventos	2.45	2,82	1.91	0	26
Decisões	9.67	17,75	10.82	0	142
Trans. de Estado	78.21	131,73	81.72	0	982
Trans. de Evento	7.51	16,58	8.50	0	189
Trans. Total	85.72	142,47	88.57	0	1056

As Figuras 4.19 e 4.20 refletem o mesmo desequilíbrio, mostrando valores extremos em pequenas partes da amostra e mesmo valores nulos ou bem próximos de zero em outras partes da amostra. Nenhum atributo da Tabela 4.6 têm valor médio próximo de estar centralizado entre o maior e o menor valor da população.

Figura 4.18 - Número de Estados por classe da amostra do TerraLib

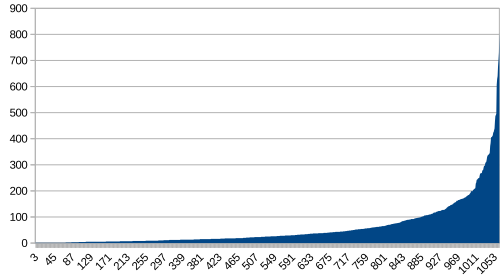


Figura 4.19 - Número de Eventos por classe da amostra do TerraLib

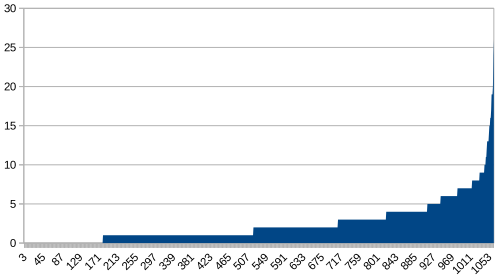
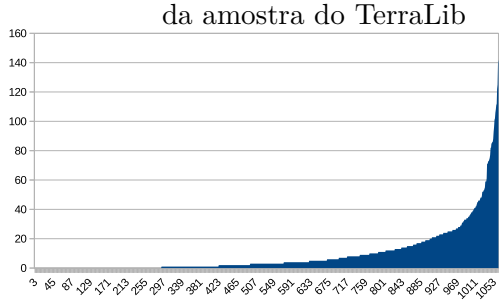


Tabela 4.7 - Cobertura das amostras do TerraLib.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Cobertura de Estados	28,29%	37,73	24,27	0	380
Cobertura de Transições	27,52%	37,74	24,28	0	379

Figura 4.20 - Número de Decisões por classe da amostra do TerraLib



Essa amostra não mostra um cenário onde é esperado um bom desempenho do método como era esperado no caso anterior. A Tabela 4.7 mostra os valores de cobertura de estados e transições pelo método. Novamente, os valores de desvio padrão são maiores que a média, e o valor de média é bem diferente do que seria um valor central entre os extremos da população. É visto na Figura 4.21 que, ao dividir o número de estados cobertos pelo número de estados totais por classe observa-se um grupo de classes com até pouco mais de 100 estados com uma porcentagem significativa de cobertura de estados pelo método. Classes com mais estados apresentam uma porcentagem de cobertura arbitrária.

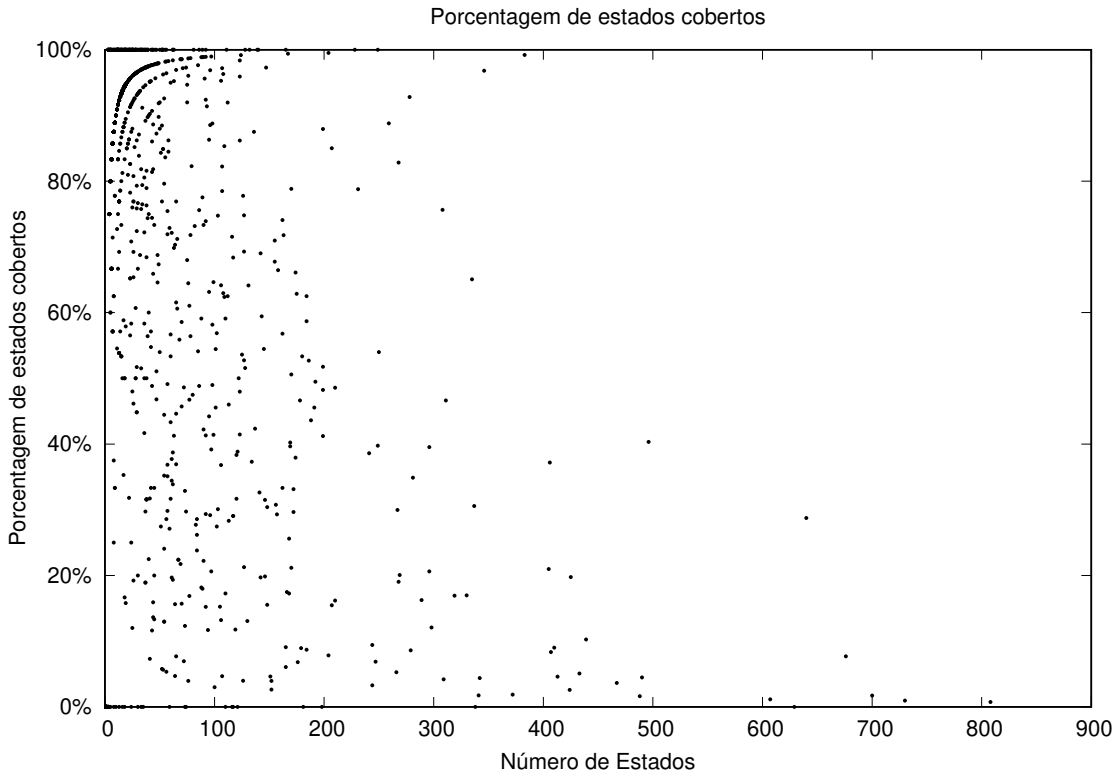
Um desempenho similar à cobertura de estados é vista na Figura 4.22 com a porcentagem de transições cobertas por classe. Aqui, o grupo de maior cobertura aparece em classes com até 100 transições de estado, se dispersando rapidamente após esse valor.

Os valores de complexidade vistos na Tabela 4.8 mostram níveis de complexidade ciclomática que variam de zero até quatrocentos e setenta de dois, juntamente a classes com centenas de componentes.

Tabela 4.8 - Complexidade das amostras do TerraLib.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Componentes	7,89	8,35	5,05	0	123
Complexidade	35,55	55,17	33,53	0	472

Figura 4.21 - TerraLib: Porcentagem de Estados cobertos de acordo com o número de estados por classe



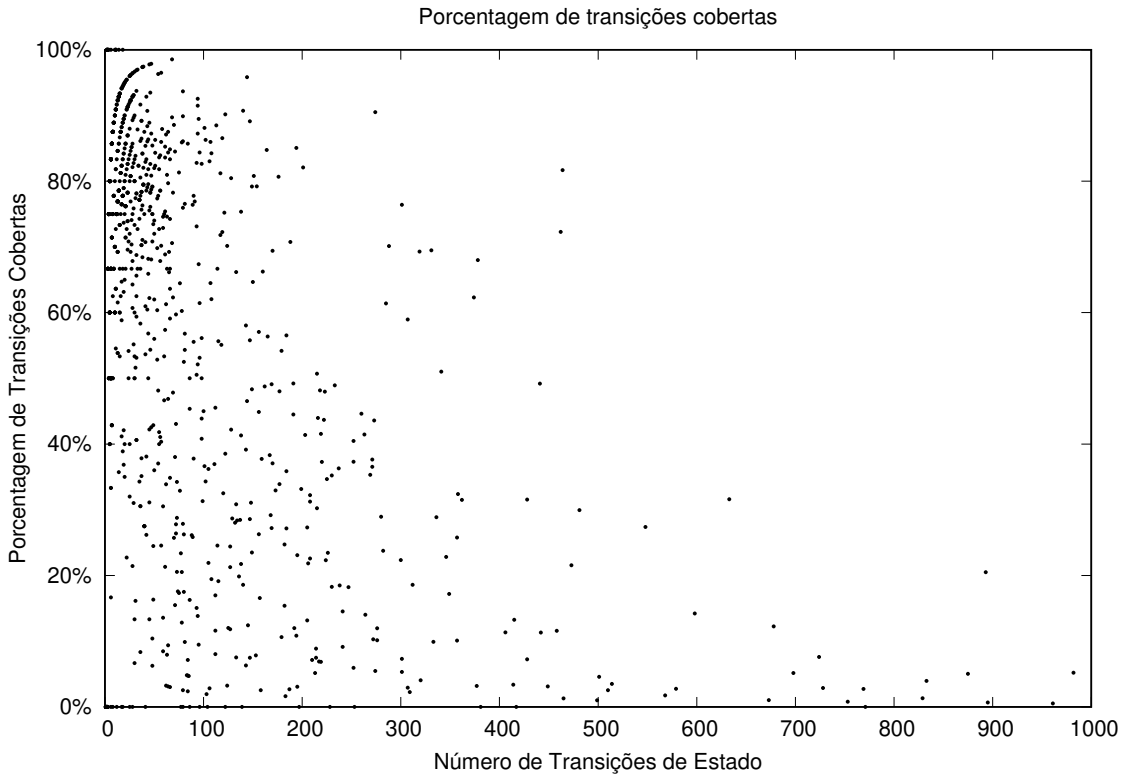
A quantidade de propriedades geradas, vista na Tabela 4.9 apresenta uma variação bastante expressiva. Com o valor mínimo de nenhuma propriedade até mais duas mil e duzentas, destaca que o conjunto de propriedades do caso 3 é de longe o maior, o que chama a atenção para uma investigação aprofundada sobre esses números.

Tabela 4.9 - Dados sobre as propriedades geradas a partir das amostras do TerraLib.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Propriedades	178,41	300,03	185,93	0	2210
Caso 1	2,45	2,82	1,91	0	26
Caso 2	19,35	35,88	21,85	0	279
Caso 3	156,62	265,36	164,21	0	1964

Ao compararmos o total de propriedades geradas por classe com o número de estados e transições cobertos, obtém-se dois gráficos virtualmente idênticos, vistos nas Figuras 4.23 e 4.24. Ambos os cenários conseguem ter alguma cobertura até apro-

Figura 4.22 - TerraLib: Porcentagem de Transições cobertas de acordo com o número de estados por classe



ximadamente 100 estados ou 100 transições. A geração de um grande número de propriedades novamente não parece ser eficiente para cobertura do código testado.

Os dados sobre os contraexemplos gerados, produtos finais do método, são mostrados na Tabela 4.10. Novamente os dados apresentam um desvio padrão maior que a média, com a única exceção do "Menor Contraexemplo". Os valores de média novamente estão longe do valor central entre o máximo e mínimo.

Tabela 4.10 - TerraLib: Dados sobre os contraexemplos gerados a partir das amostras do TerraLib.

	Média	Desvio Padrão	Desvio Médio	Mínimo	Máximo
Total de Contraexemplos	118,82	168,44	112,70	0	1208
Contraexemplos Válidos	40,15	60,43	38,12	0	617
Contraexemplos Inválidos	78,66	130,45	80,42	0	984
Maior Contraexemplo	19,55	21,42	14,52	0	227
Menor Contraexemplo	2,67	0,94	0,59	0	4

Figura 4.23 - TerraLib: Estados cobertos de acordo com o número de propriedades geradas por classe

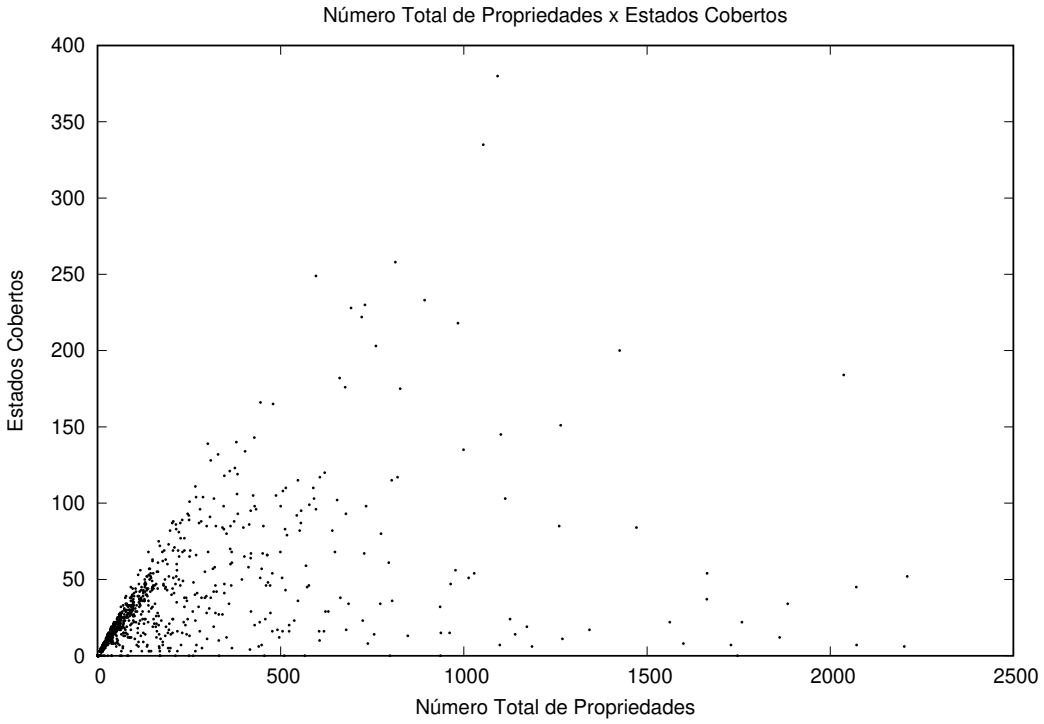
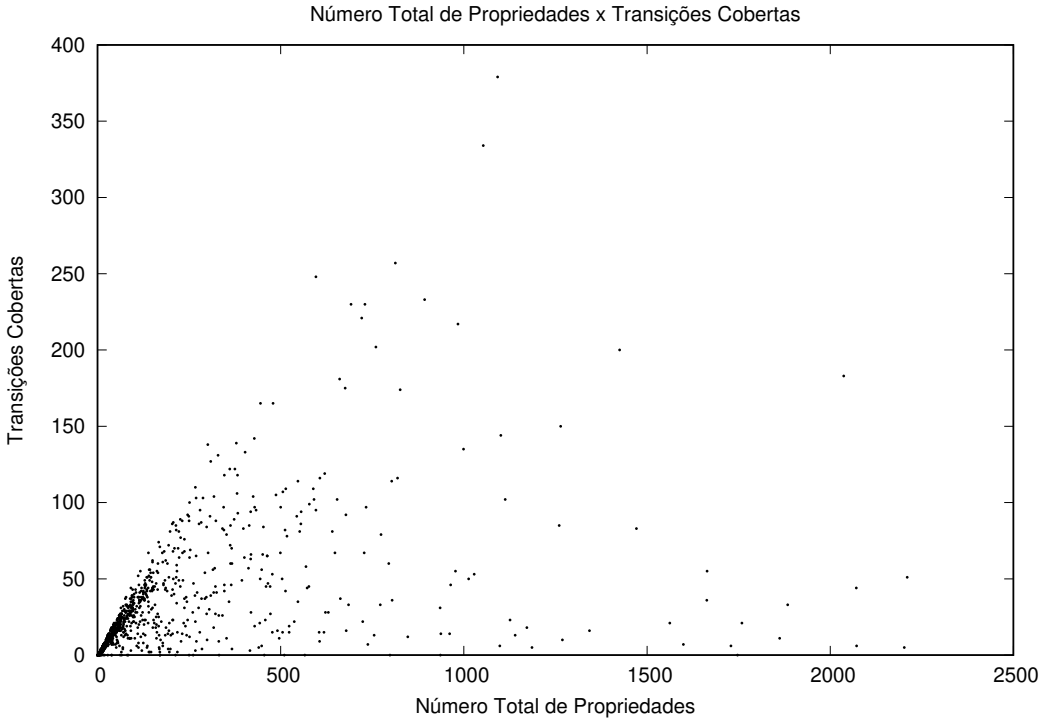
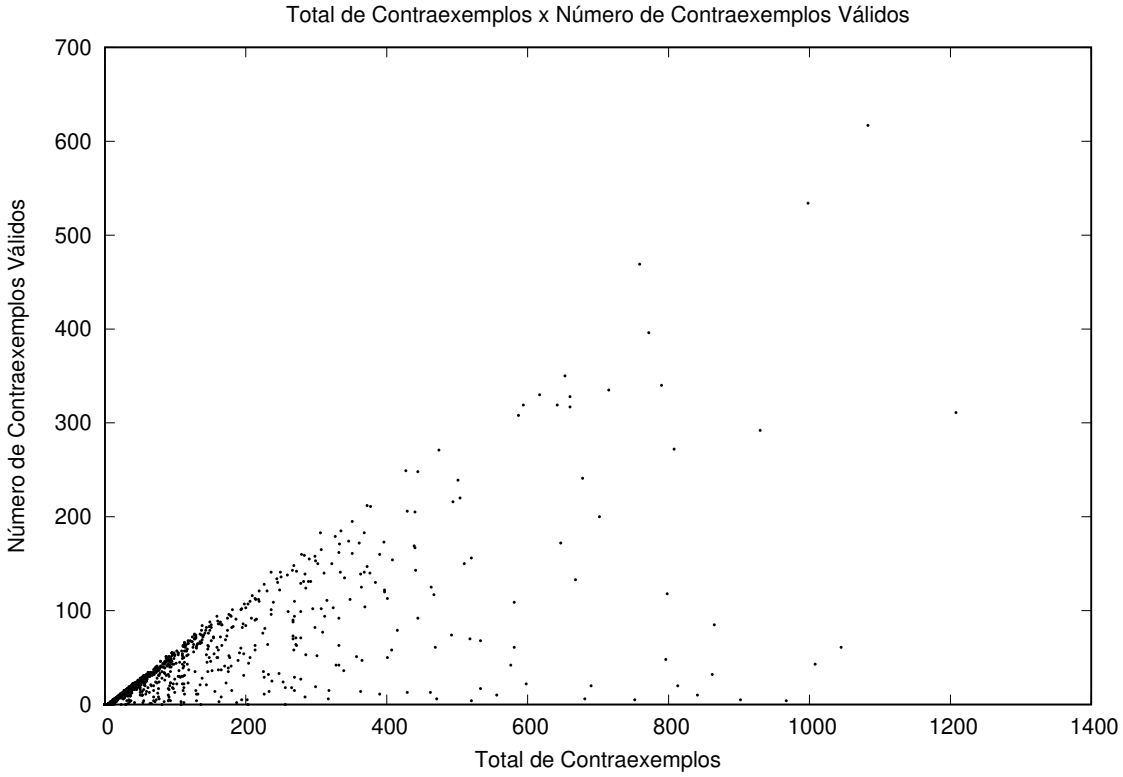


Figura 4.24 - TerraLib: Transições cobertas de acordo com o número de propriedades geradas por classe



A Figura 4.25 mostra que a correspondência entre contraexemplos válidos diante do total gerado não vai muito além da primeira centena. É tentador culpar o mau comportamento da amostra pela limitação do desempenho à cenários pequenos, mas é preciso lembrar que um desempenho não muito diferente foi observada no exemplo anterior, mesmo usando classes selecionadas à dedo. O método mostrou uma limitação em lidar com cenários grandes.

Figura 4.25 - TerraLib: Contraexemplos válidos de acordo com o número total de contraexemplos geradas por classe



Observando o gráfico da Figura 4.28 é visto que existe uma leve relação entre o aumento da complexidade ciclomática da classe com o aumento do número de contraexemplos inválidos gerados por classe. Ainda mais acentuado é a relação entre o aumento no número de estados e transições por classe e o aumento no número de contraexemplos inválidos visto nos gráficos das Figuras 4.26 e 4.27.

Figura 4.26 - TerraLib: Contraexemplos inválidos de acordo com o número de estados por classe

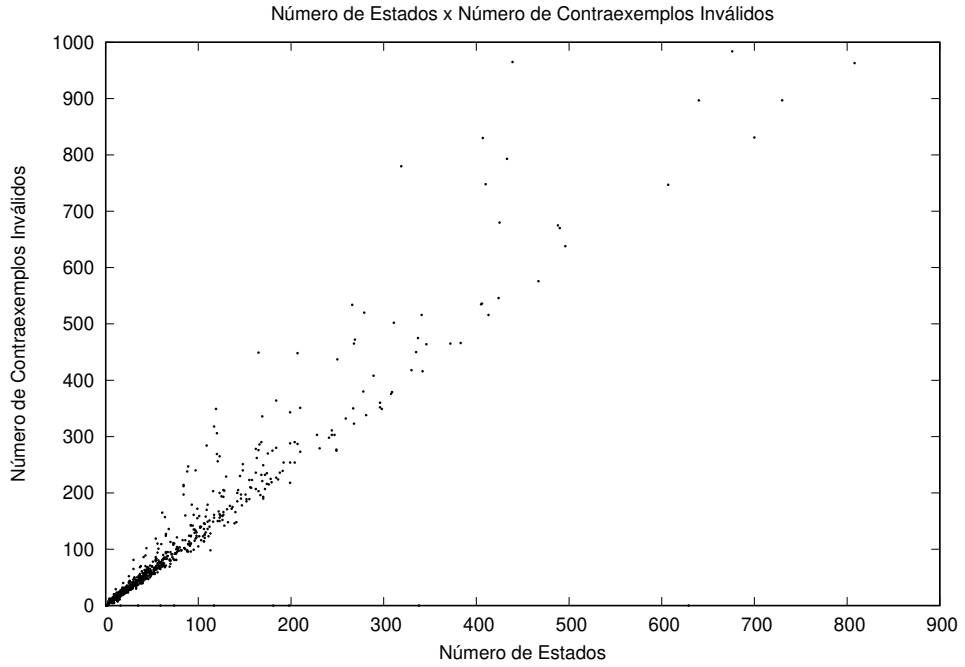


Figura 4.27 - TerraLib: Contraexemplos inválidos de acordo com o número de transições de estados por classe

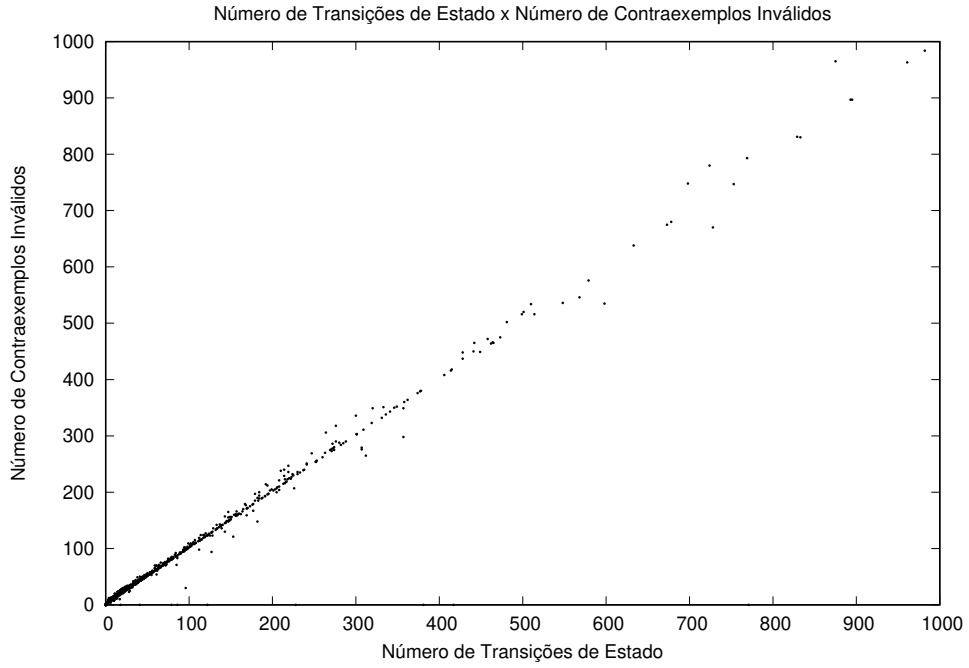
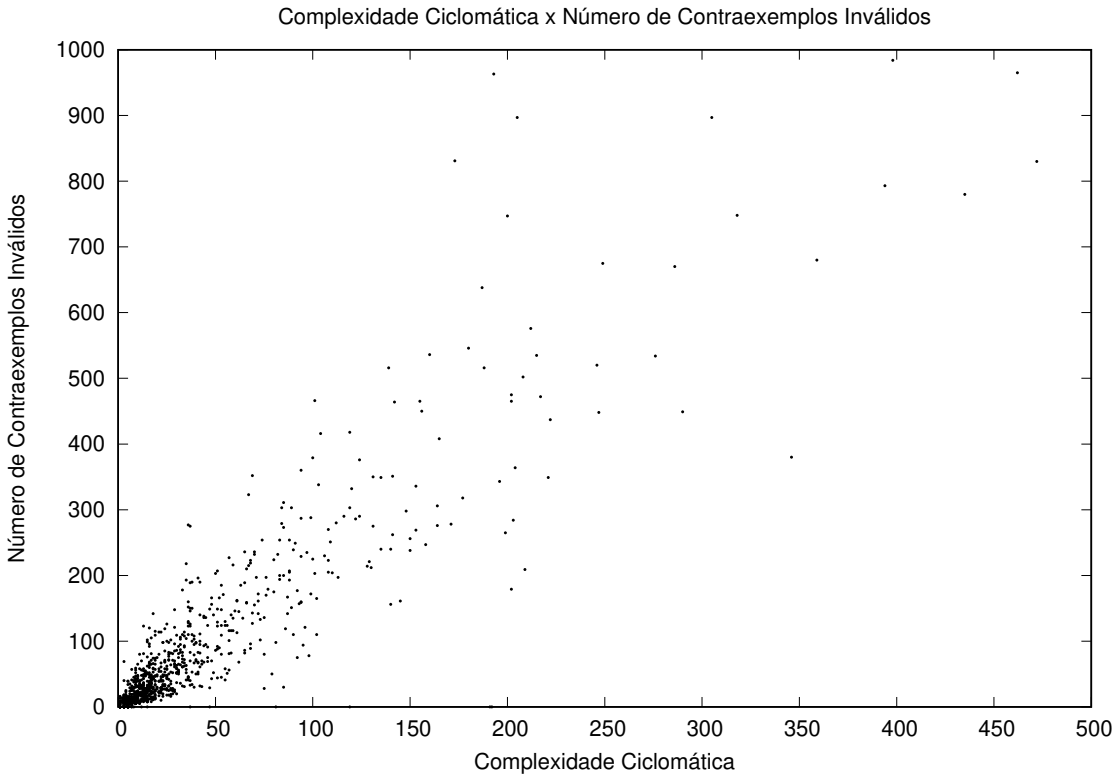


Figura 4.28 - TerraLib: Contraexemplos inválidos de acordo com o número de complexidade ciclomática por classe



Porém o item que mais demanda atenção é de fato o número de propriedades geradas, em especial as propriedades do caso 3, onde o aumento do número dessas propriedades parece estar diretamente relacionado com a geração de contraexemplos inválidos, como visto nos gráficos das Figuras 4.29 e 4.30 que parecem funções identidade.

Figura 4.29 - TerraLib: Contraexemplos inválidos de acordo com o número de propriedades geradas por classe

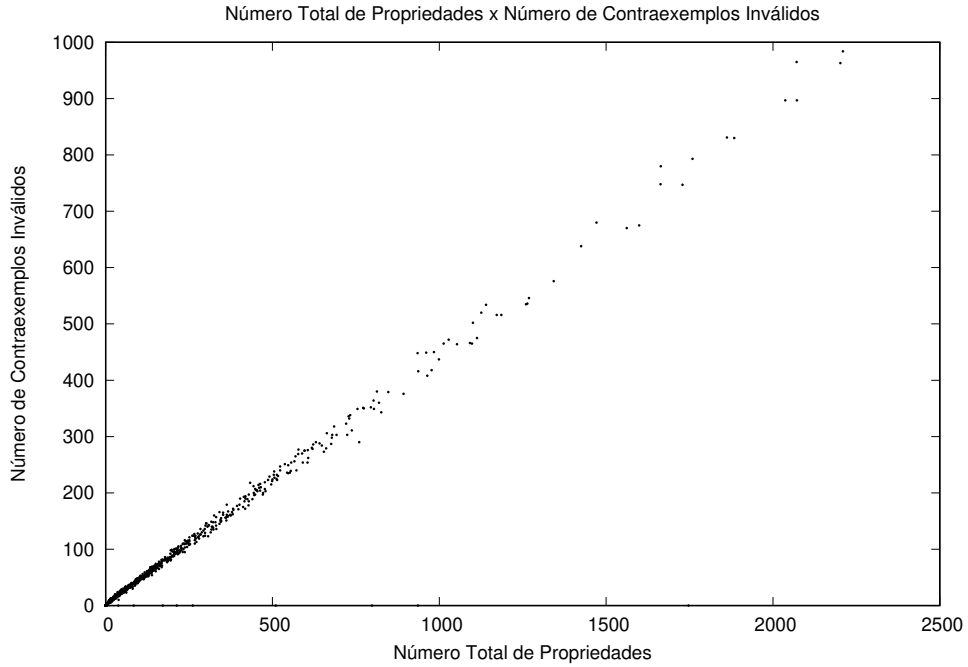
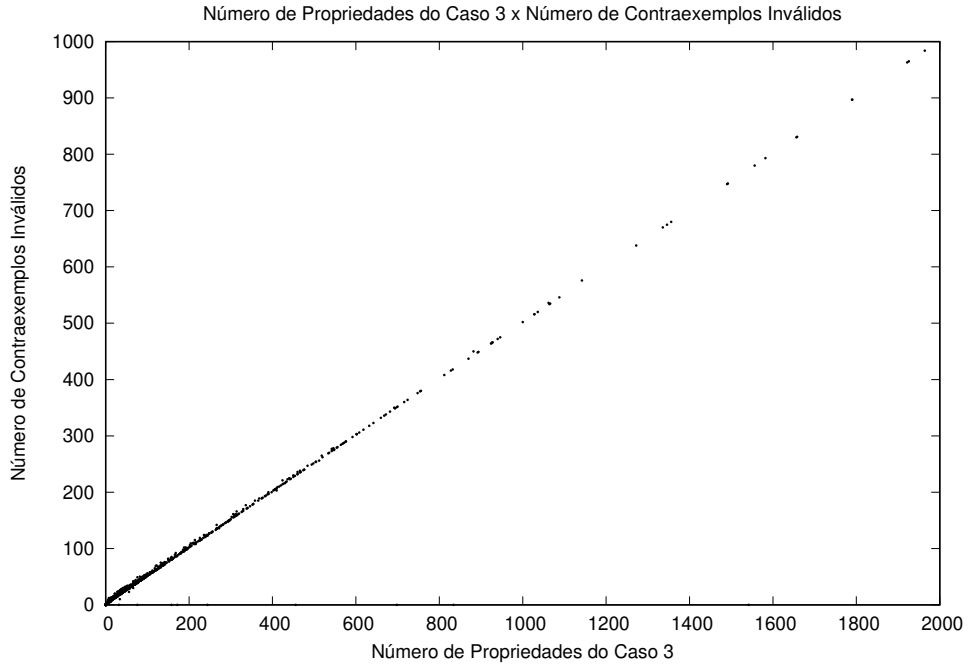


Figura 4.30 - TerraLib: Contraexemplos inválidos de acordo com o número de propriedades do caso 3 geradas por classe



No entanto, ao considerar apenas os contraexemplos válidos obtém-se resultados positivos, mesmo considerando a natureza turbulenta da amostra sob teste. O gráfico da Figura 4.32 mostra uma relação muito favorável entre o número de contraexemplos válidos e o número de estados cobertos, assim como a cobertura de transições pelos contraexemplos válidos, vista no gráfico da Figura 4.33.

Ao dividir o número de estados do maior contraexemplo pelo número de estados de sua respectiva classe, obtemos a porcentagem de cobertura desse único contraexemplo, vista na Figura 4.31, onde um grupo de cobertura significativa do código é visto para contraexemplos com até pouco mais de 50 estados.

Figura 4.31 - TerraLib: Cobertura de estados de acordo com o número de estados do maior contraexemplos de cada classe

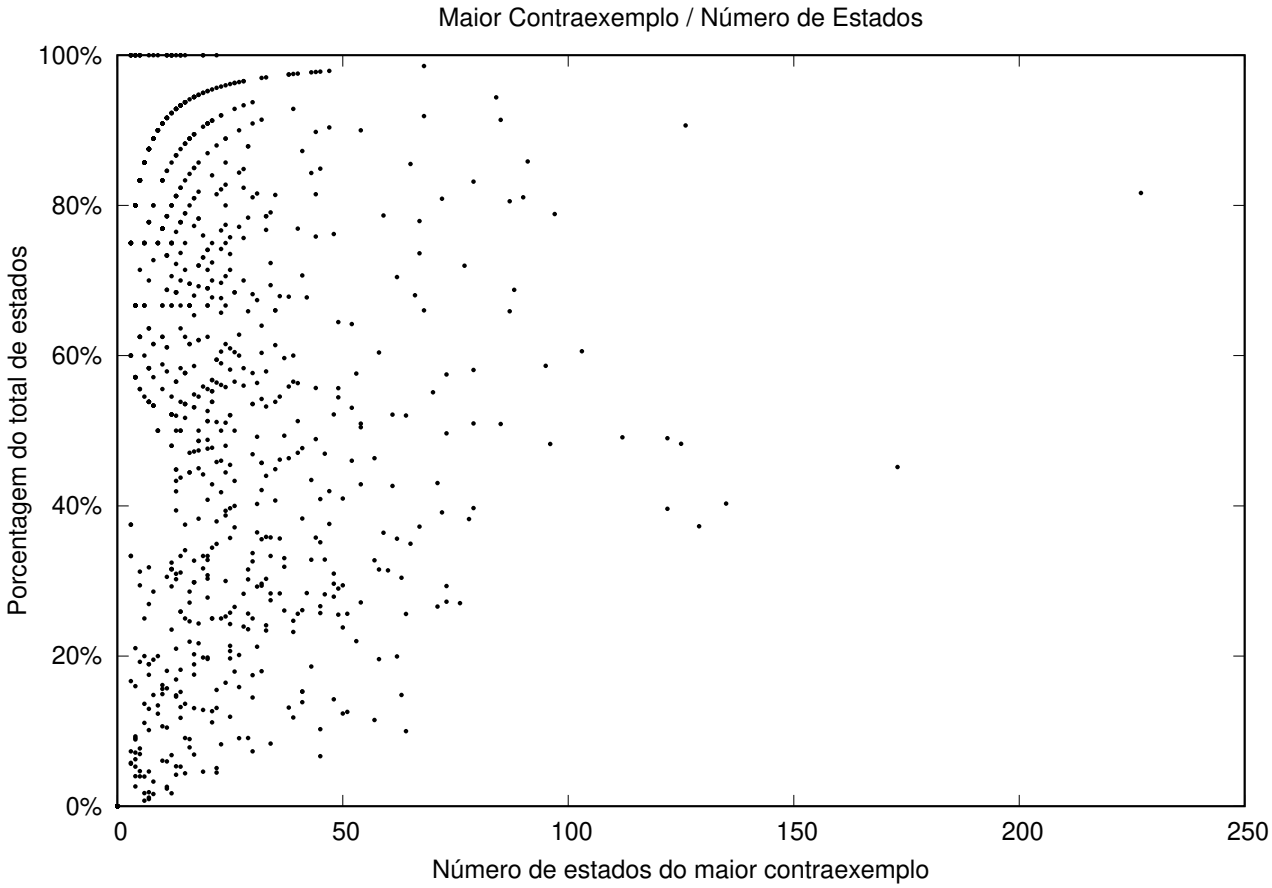


Figura 4.32 - TerraLib: Contraexemplos válidos de acordo com o número de estados por classe

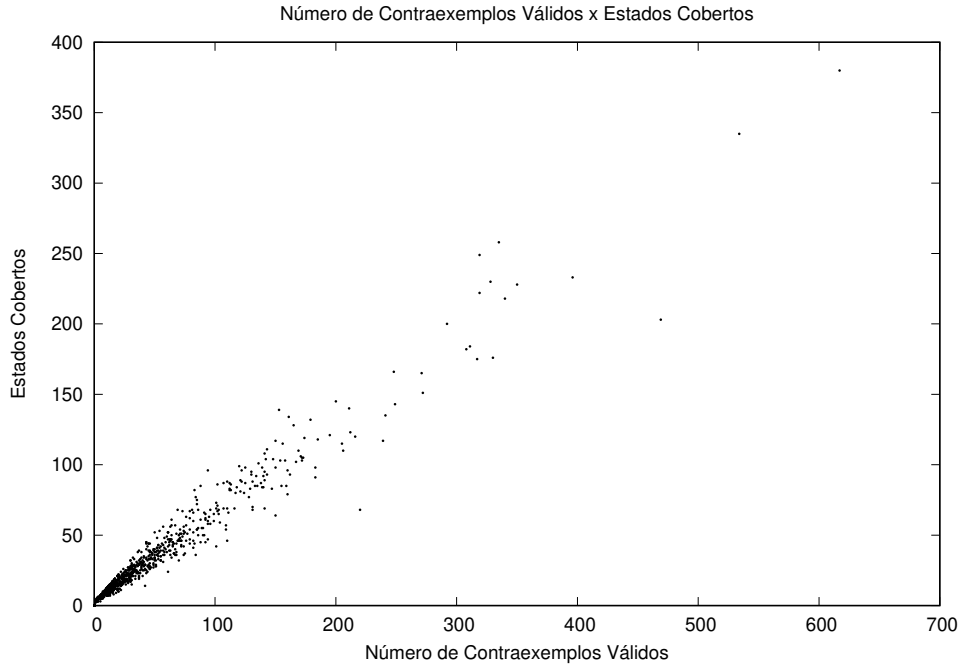
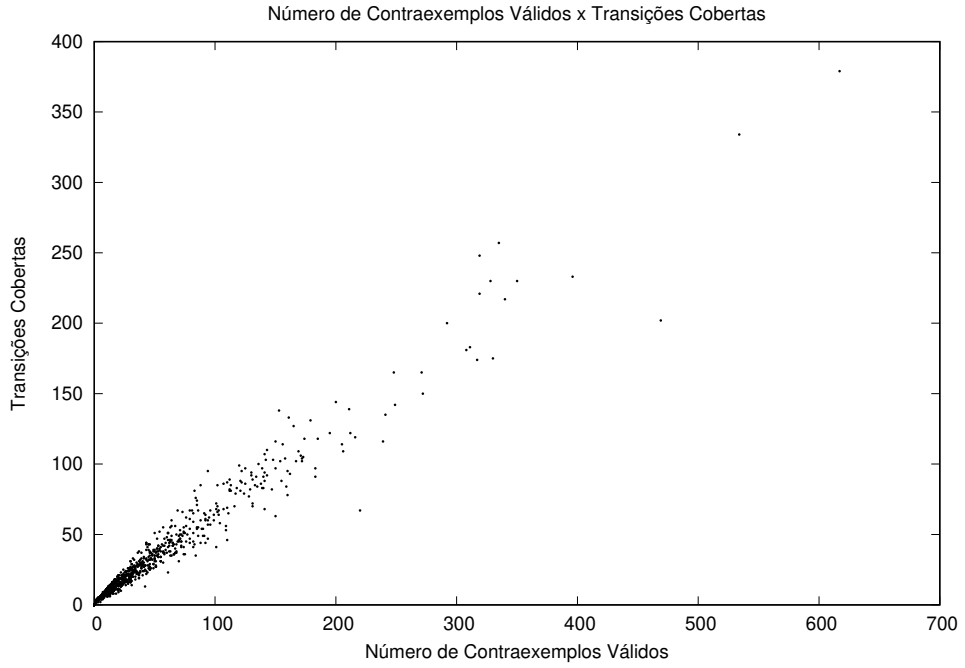


Figura 4.33 - TerraLib: Contraexemplos válidos de acordo com o número de transições de estados por classe



4.4 Considerações finais

O método Singularity foi testado em dois cenários, o primeiro deles consiste em 24 classes C++ escolhidas cuidadosamente entre as classes que compõe o programa GeoDMA, o segundo é composto por 1063 classes pertencentes ao programa TerraLib escolhidas sem nenhuma pré-seleção, todas as classes suportadas pelo método foram executadas. Em ambos os cenários o método se comportou de forma bastante similar, independente da pré-seleção, conseguindo gerar "um bom número de contraexemplos"denominados "eficientes", em sua grande maioria em classes com até aproximadamente, entre 150 e 200 estados. Houve uma grande produção de contraexemplos "inválidos", isso é, com dois ou menos estados, para classes com um número de estados superior a 200. Grande parte desses contraexemplos são gerados a partir das propriedades do chamado "caso 3". Executando apenas um único contraexemplo por classe, o maior contraexemplo gerado, é possível obter coberturas expressivas principalmente em classes com menos de 100 estados.

Em resposta à hipótese vista no Capítulo 1, um grande número de contraexemplos não parece ser uma garantia de boa cobertura do código, porém nem tão pouco atrapalham a cobertura. Existe sim, uma cobertura interessante do código em muitos cenários (classes com entre 100 e 200 estados) mas não são utilizados todos os contraexemplos gerados para essa cobertura. Parece existir um excesso de contraexemplos em alguns casos, especialmente em classes com um grande número de estados, sugerindo o uso de um filtro ou mesmo diferentes propriedades em busca de um conjunto mais seleto de contraexemplos.

5 CONCLUSÃO

O presente trabalho mostrou um método, denominado Singularity, para geração de casos de teste a partir de um código fonte C++ usando contraexemplos de um Verificador de Modelos (ERAS et al., 2019). Um contraexemplo do Verificador de Modelos é considerado um caso de teste abstrato no contexto desse trabalho. Esse método é dividido em três componentes (Extrator, Gerador e Construtor) que refletem três etapas para obtenção dos contraexemplos. Foi feita uma avaliação do desempenho desse método com a execução de classes C++ provenientes das aplicações de software para geoprocessamento desenvolvidas no INPE, GeoDMA (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, a) e TerraLib (INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE, b), sendo que as classes utilizadas do GeoDMA foram escolhidas de forma a otimizar o desempenho do método, enquanto todas as classes compatíveis do TerraLib foram utilizadas, sem nenhum critério de seleção. No total, foram analisadas 24 classes da aplicação GeoDMA e 1063 classes da TerraLib na geração de casos de teste unitários. O sucesso em gerar casos de teste abstratos para uma aplicação do porte da TerraLib demonstra a viabilidade da abordagem para lidar com sistemas de software não triviais.

O método foi implementado na forma de uma ferramenta Java, disponível em `https://github.com/eduardoeras/Singularity`, para fins de prova de conceito. Para isso, fez-se uso do gerador de *parser* ANTLR (PARR, 2019) e a geração de casos de teste é via o Verificador de Modelos NuSMV (CIMATTI et al., 1999).

O método mostrado nesse trabalho pareceu promissor na cobertura dos estados e transições do modelo gerado e, consequentemente na cobertura do código em C++, mesmo que de forma exclusivamente teórica, utilizando os contraexemplos válidos gerados. Ainda que considerada apenas a utilização do maior contraexemplo, esse único contraexemplo já traz uma cobertura teórica interessante do código para classes com até aproximadamente 150 estados. É pertinente notar que esse desempenho se manteve mesmo nos casos de teste gerados a partir das classes do TerraLib, um cenário bastante complexo.

5.1 Trabalhos futuros

Existem muitas melhorias a serem exploradas. As duas maiores fraquezas do método são a não geração de casos de teste executáveis e baixa eficiência em cenários de grande porte. A conversão dos casos de teste abstratos gerados pelo método Singularity é o primeiro e mais importante trabalho futuro. Essa não é uma tarefa trivial.

Para esse fim, é preciso considerar, por exemplo, um escopo maior do que o de uma classe de forma a minimizar a sobrecarga (*overhead*), comum para testes unitários principalmente em cenários de maior complexidade. Outro ponto a ser levado em conta é a própria linguagem de programação C++, que é muito abrangente.

O segundo ponto a se destacar é a necessidade do método de melhor apoiar cenários de grande porte. É louvável a ideia de que um código fonte, não somente em C++, seja organizado, otimizado, modular, conciso e minimalista. Entretanto, na realidade tais códigos são mais exceção do que regra. Códigos fonte costumam ser muito complexos e a escrita de classes com milhares de linhas acaba sendo uma prática comum. Muitas vezes a otimização demanda tempo e habilidades não disponíveis em um projeto submetido a prazos curtos, recursos limitados e equipe em processo de aprendizado. Um método preparado para geração de casos de testes a partir de arquivos grandes, complexos e redundantes é necessário. Um apoio robusto às praticas de programação não ortodoxas, porém comuns, é vital para maior utilidade do método. Mesmo dentro de um escopo de programação mais formal, o método não apoia diversas características da linguagem C++ como, por exemplo, sobrecarga de funções e métodos.

Outro ponto importante a ser trabalhado é a melhor elaboração das propriedades armadilha. Existe uma forte relação entre a geração de propriedades e a geração de contraexemplos inválidos e o aumento do número de contraexemplos não parece melhorar a qualidade da cobertura do código. Um destaque fica para as propriedades do caso 3 que são as mais numerosas e menos eficientes. O trabalho de Fraser *et al* (FRASER et al., 2009) cita outras formas de gerar essas propriedades, o que pode ser explorado pelo método. O uso do método HiMoST (DE SANTIAGO JÚNIOR; DA SILVA, 2017) em sua integridade, e não somente como base para geração de propriedades armadilhas, é também uma proposta de estudos muito válida na busca pela maior qualidade dos contraexemplos gerados. Alguma nova forma de geração dessas propriedades também pode ser proposta, como por exemplo o uso de aprendizado de máquina para geração de propriedades que busquem uma melhor cobertura de estados e transições do modelo.

É importante destacar, também, a realização de experimentos controlados no futuro, e comparações com outros métodos. Por exemplo, nos trabalhos relacionados, vistos no Capítulo 2, é notada a semelhança de proposta do método Singularity com a KLOVER (PRASAD et al.,), que é de uso exclusivo da Fujitsu. Outra importante contribuição futura seria a aplicação do método Singularity para uma diferente gama

de programas em C++ como, por exemplo, para softwares embarcados em computadores de satélites e outros sistemas da área aeroespacial. Esse tipo de software é de grande importância para as atividades desenvolvidas dentro do INPE, e requerem rigorosas etapas de teste durante seu desenvolvimento, um cenário promissor para aplicação de um método de geração de casos de teste automáticos.

REFERÊNCIAS BIBLIOGRÁFICAS

AMMANN, P. E.; BLACK, P. E.; MAJURSKI, W. Using model checking to generate tests from specifications. In: **INTERNATIONAL CONFERENCE ON FORMAL ENGINEERING METHODS, 2., 1998. Proceedings...** [S.l.]: IEEE, 1998. p. 46–54. ISBN 0818691980. 3, 18

BAIER, C.; KATOEN, J. P. **Principles of model checking**. [S.l.]: MIT Press, 2008. 984 p. ISSN 0010-4620. ISBN 9780262026499. 1, 8, 12, 13, 14

BALERA, J. M.; DE SANTIAGO JÚNIOR, V. A. T-tuple reallocation: an algorithm to create mixed-level covering arrays to support software test case generation. **Lecture Notes in Computer Science**, v. 9158, p. 503–517, 2015. ISSN 16113349. 34

BALERA, J. M.; SANTIAGO JÚNIOR, V. A. An algorithm for combinatorial interaction testing: definitions and rigorous evaluations. **Journal of Software Engineering Research and Development**, v. 5, n. 1, p. 10, Dec 2017. ISSN 2195-1721. Disponível em: <https://doi.org/10.1186/s40411-017-0043-z>. 17, 34

BASHIR, I.; GOEL, A. L. Testing c++ classes. In: IEEE. **INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, RELIABILITY AND QUALITY ASSURANCE, 1., 1994. Proceedings...** [S.l.], 1994. p. 43–48. 3, 19

BEYER, D.; CHLIPALA, A. J.; HENZINGER, T. A.; JHALA, R.; MAJUMDAR, R. Generating tests from counterexamples. In: **International Conference on Software Engineering, Proceedings...** [S.l.: s.n.], 2004. v. 26, p. 326–335. ISSN 02705257. 18

BEYER, D. H.; HENZINGER, T. A.; JHALA, R.; MAJUMDAR, R. The software model checker b last. **International Journal on Software Tools for Technology Transfer**, v. 9, n. 5-6, p. 505–525, 2007. 3, 18

BINDER, R. **Testing object-oriented systems: models, patterns, and tools**. [S.l.]: Addison-Wesley Professional, 2000. 7

BONFANTI, S.; GARGANTINI, A.; MASHKOOR, A. Generation of c++ unit tests from abstract state machines specifications. In: **INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND**

VALIDATION WORKSHOP, 11., 2018. Proceedings... [S.l.]: IEEE, 2018. p. 185–193. ISBN 9781538663523. 3, 19

BRIAND, L. C. Software documentation: how much is enough? In: **EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 7., 2003. Proceedings...** [S.l.: s.n.], 2003. p. 13–15. ISSN 1534-5351. 1

CIMATTI, A.; CLARKE, E.; GIUNCHIGLIA, F.; ROVERI, M. Nusmv: a new symbolic model verifier. In: SPRINGER. **INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION, 1999. Proceedings...** [S.l.], 1999. p. 495–499. 3, 75

CREPANI, E.; MEDEIROS, J. S. Imagens cbers+ imagens srtm+ mosaicos geocover landsat em ambiente spring e terraview: sensoriamento remoto e geoprocessamento gratuitos aplicados ao desenvolvimento sustentável. In: **SIMPÓSIO BRASILEIRO DE SENSORIAMENTO REMOTO, 12., 2005. Anais... São José dos Campos: INPE**. [S.l.: s.n.], 2005. 2

DE SANTIAGO JÚNIOR, V. A.; DA SILVA, F. E. C. From Statecharts into Model Checking : a hierarchy-based translation and specification patterns properties to generate test cases. In: **BRAZILIAN SYMPOSIUM ON SYSTEMATIC AND AUTOMATED SOFTWARE, 2., 2017. Proceedings...** [S.l.: s.n.], 2017. ISBN 9781450353021. 1, 4, 16, 17, 21, 34, 37, 41, 44, 76

de Santiago Júnior, V. A. **SOLIMVA: a methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications**. Tese (Doutorado em Computaçã Aplicada) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2011., São José dos Campos, 2011. 16

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2017. 7, 10, 12, 14, 15

DELAMARO, M. E. **Proteum - um ambiente de teste baseado na análise de mutantes**. Tese (Doutorado em Computaçã e Matemática Aplicada) — Universidade de São Paulo, São Paulo, 1993. 3

DEUTSCH, M. S. **Software verification and validation: realistic project approaches**. [S.l.]: Prentice Hall PTR, 1981. 2, 9

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specifications for finite-state verification. In: **INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 21., 1999. Proceedings...** [s.n.], 1999. p. 411–420. ISBN 1581130740. ISSN 0270-5257. Disponível em: <http://portal.acm.org/citation.cfm?doid=302405.302672>. 17, 34

ERAS, E. R.; DE SANTIAGO JÚNIOR, V. A.; DOS SANTOS, L. B. R. Singularity: a methodology for automatic unit test data generation for c++ applications based on model checking counterexamples. In: **BRAZILIAN SYMPOSIUM ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING, 4., 2019. Proceedings...** [S.l.: s.n.], 2019. p. 72–79. 3, 21, 75

FRASER, G.; ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In: **ACM SIGSOFT SYMPOSIUM, 19., 2011, and EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING, 13., 2011. Proceedings...** [S.l.: s.n.], 2011. p. 416–419. 3

FRASER, G.; WOTAWA, F. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In: ICSEA. **INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES, 2., 2007. Proceedings...** [S.l.], 2007. ISBN 0769529372. 1, 21

FRASER, G.; WOTAWA, F.; AMMANN, P. E. Testing with model checkers: a survey. **Software Testing Verification and Reliability**, v. 19, n. 3, p. 215–261, 2009. ISSN 09600833. 7, 34, 37, 44, 76

GARG, P.; IVANCIC, F.; BALAKRISHNAN, G.; MAEDA, N.; GUPTA, A. Feedback-directed unit test generation for c/c++ using concolic execution. In: **INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013. Proceedings... 2013.** [S.l.: s.n.], 2013. p. 132–141. ISBN 9781467330763. ISSN 02705257. 3, 19

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, n. 3, p. 231–274, 1987. ISSN 01676423. 16, 17, 34

HOWDEN, W. E. Completeness criteria for testing elementary program functions. In: **INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 5., 1981. Proceedings...** [S.l.]: IEEE, 1981. p. 235–243. 11

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - INPE. **GeoDMA - Geographic Data Mining Analyst**. Disponível em:

<http://wiki.dpi.inpe.br/doku.php?id=geodma>. Acesso em: 14 Fev. 2020. 2, 4, 45, 75

_____. **TerraLib and TerraView Wiki Page**. Disponível em: <http://www.dpi.inpe.br/terralib5/wiki/doku.php?id=start>. Acesso em: 14 Fev. 2020. 2, 4, 45, 75

JORGENSEN, P. C. **Software testing: a craftsman's approach**. [S.l.]: Auerbach Publications, 2013. 7, 9

LAKHOTIA, K.; HARMAN, M.; GROSS, H. Austin: an open source tool for search based software testing of c programs. **Information and Software Technology**, Elsevier, v. 55, n. 1, p. 112–125, 2013. 3

LI, G.; LI, P.; SAWAYA, G.; GOPALAKRISHNAN, G.; GHOSH, I.; RAJAN, S. P. Gklee: concolic verification and test generation for gpus. In: **ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 17., 2012. Proceedings...** [S.l.: s.n.], 2012. p. 215–224. 17

MAO, C.; LU, Y. Cpptest: a prototype tool for testing c/c++ programs. In: IEEE. **INTERNATIONAL CONFERENCE ON AVAILABILITY, RELIABILITY AND SECURITY, 2., 2007. Proceedings...** [S.l.], 2007. p. 1066–1073. 3

MCCABE, T. J. A complexity measure. **Ieee Transactions on Software Engineering**, SE 2, n. 4, p. 308–320, 1976. 15, 46, 47

MCMILLAN, K. L. Symbolic model checking. In: HARTONAS-GARMHAUSEN, V. (Ed.). **Symbolic model checking**. Boston, MA: Springer, 1993. p. 25–60. 1

MYERS, G. J.; BADGETT, T.; THOMAS, T. M.; SANDLER, C. **The art of software testing**. [S.l.]: Wiley Online Library, 2004. 9

PARR, T. **ANTLR 4 documentation**. 2019. Disponível em: <https://github.com/antlr/antlr4/blob/master/doc/index.md>. Acesso em: 17 Abr. 2019. 45, 75

PRASAD, M.; YOSHIDA, H.; LI, G. **Utilization and evolution of KLEE-based technologies for embedded software testing at fujitsu**. Disponível em: <https://srg.doc.ic.ac.uk/klee18/talks/Ghosh-Keynote.pdf>. Acesso em: 18 Fev. 2020. 3, 17, 76

PRESSMAN, R.; MAXIM, B. **Engenharia de software**. 8. ed. [S.l.]: McGraw Hill Brasil, 2016. 11

RAYADURGAM, S.; HEIMDAHL, M. P. E. Coverage based test-case generation using model checkers. In: **INTERNATIONAL CONFERENCE AND WORKSHOP ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS, 8., 2001. Proceedings...** [S.l.: s.n.], 2001. p. 83–91. ISBN 0769510868. 3, 18

SANTIAGO, V.; VIJAYKUMAR, N. L.; AES, D. G.; AMARAL, A. S.; FERREIRA, E. An environment for automated test case generation from Statechart-based and Finite State Machine-based behavioral models. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST) - WORKSHOP ON ADVANCES IN MODEL BASED TESTING (A-MOST), 1., 2008, Lillehammer, Norway. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 63–72. 16

TIOBE. **TIOBE index for February 2020**. Victory House II, Company Number 2089, Esp 401, 5633 AJ Eindhoven, The Netherlands: [s.n.], 2020. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 14 Fev. 2020. 2

VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. Jabuti: a coverage analysis tool for java programs. In: **SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 17., 2003. Anais...** [S.l.: s.n.], 2003. p. 79–84. 3

WESTLEY, A. E. **Infotech state of the art report: software testing, volume 1: analysis and bibliography**. Maidenhead, England: Infotech International Limited, 1979. 11

YOSHIDA, H.; LI, G.; KAMIYA, T.; GHOSH, I.; RAJAN, S.; TOKUMOTO, S.; MUNAKATA, K.; UEHARA, T. Klover: automatic test generation for c and c++ programs, using symbolic execution. **IEEE Software**, v. 34, n. 5, p. 30–37, 2017. ISSN 07407459. 2, 3, 17

YOSHIDA, H.; TOKUMOTO, S.; PRASAD, M. R.; GHOSH, I.; UEHARA, T. Fsx: fine-grained incremental unit test generation for c/c++ programs. In: **INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 25., 2016. Proceedings...** [S.l.: s.n.], 2016. p. 106–117. 17

APÊNDICE A - Dados de execução de uma classe do GeoDMA

A.1 Código C++

Código C++ original da classe "analysis.cpp"do programa GeoDMA:

```
/*  Copyright (C) 2012 National Institute For Space Research (INPE) - Brazil.

This file is part of the GeoDMA - a Toolbox that integrates Data Mining Techniques with object-based
and multi-temporal analysis of satellite remotely sensed imagery.

GeoDMA is free software: you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation, either version 3 of the License,
or (at your option) any later version.

GeoDMA is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with GeoDMA. See COPYING. If not, write to
GeoDMA Team at <thales@dpi.inpe.br, raian@dpi.inpe.br, castejon@dpi.inpe.br>.
*/

#include <boost/filesystem.hpp>

#include <string>

// GeoDMA Includes
#include "analysis.hpp"

namespace gdma {
namespace sa {

  Analysis::Analysis()
  {
    reset();
  }

  Analysis::Analysis( const Analysis& other )
  {
    operator=( other );
  }

  Analysis::~Analysis()
  {
  }

  const std::string& Analysis::getName() const
  {
    std::lock_guard<std::mutex> lock( m_mutex );
    return m_name;
  }

  Analysis& Analysis::operator=( const Analysis& other )
  {
    reset();

    std::lock_guard<std::mutex> lock( m_mutex );
    std::lock_guard<std::mutex> lock2( other.m_mutex );

    m_enableCache = other.m_enableCache;
    m_projectPtr = other.m_projectPtr;
    m_name = other.m_name;
    m_outputDirectoryName = other.m_outputDirectoryName;

    m_graph = other.m_graph;
    m_graph.setAnalysis( *this );

    m_context = other.m_context;
```

```
  return *this;
}

void Analysis::setName( const std::string& newName )
{
  std::lock_guard<std::mutex> lock( m_mutex );
  m_name = newName;
}

const Graph& Analysis::getGraph() const
{
  std::lock_guard<std::mutex> lock( m_mutex );
  return m_graph;
}

Graph& Analysis::getGraph()
{
  std::lock_guard<std::mutex> lock( m_mutex );
  return m_graph;
}

bool Analysis::setGraph( const Graph& newGraph )
{
  std::lock_guard<std::mutex> lock( m_mutex );

  boost::filesystem::path graphOutDir( m_outputDirectoryName );
  graphOutDir /= newGraph.getName();

  m_graph = newGraph;

  if( ! m_graph.setOutputDirectoryName( graphOutDir.string() ) )
  {
    return false;
  }

  m_graph.enableCache( m_enableCache );
  m_graph.setAnalysis( *this );

  return true;
}

const Context& Analysis::getContext() const
{
  std::lock_guard<std::mutex> lock( m_mutex );
  return m_context;
}

void Analysis::setContext( const Context& newContext )
{
  std::lock_guard<std::mutex> lock( m_mutex );
  m_context = newContext;
}

bool Analysis::setOutputDirectoryName( const std::string& outDirName )
{
  boost::filesystem::path completePath;
  try
  {
    completePath = boost::filesystem::system_complete( outDirName );
  }
  catch(...)
  {
    return false;
  }

  std::lock_guard<std::mutex> lock( m_mutex );

  m_outputDirectoryName = completePath.string();

  boost::filesystem::path graphOutDir( m_outputDirectoryName );
  graphOutDir /= m_graph.getName();

  return m_graph.setOutputDirectoryName( graphOutDir.string() );
}

const std::string& Analysis::getOutputDirectoryName() const
{
```

```
    std::lock_guard<std::mutex> lock( m_mutex );
    return m_outputDirectoryName;
  }

  void Analysis::enableCache( const bool enabled )
  {
    std::lock_guard<std::mutex> lock( m_mutex );

    m_enableCache = enabled;
    m_graph.enableCache( enabled );
  }

  bool Analysis::isCacheEnabled() const
  {
    std::lock_guard<std::mutex> lock( m_mutex );
    return m_enableCache;
  }

  void Analysis::reset()
  {
    std::lock_guard<std::mutex> lock( m_mutex );

    m_enableCache = false;
    m_projectPtr = 0;
    m_name.clear();
    m_outputDirectoryName.clear();
    m_graph.reset();
    m_context.reset();
  }

  void Analysis::setProject( const Project& project )
  {
    std::lock_guard<std::mutex> lock( m_mutex );
    m_projectPtr = &project;
  }

  Project const* Analysis::getProject() const
  {
    std::lock_guard<std::mutex> lock( m_mutex );
    return m_projectPtr;
  }

}  // end namespace sa
}  // end namespace gdma
```

A.2 Diagrama de transição de estados em XML

Diagrama de transição de estados gerado pelo método Singularity a partir da classe "analysis.cpp":

```
<level label='gdma' element='NAMESPACE'>
    <level label='sa' element='NAMESPACE'>
        <level label='Analysis' element='CONSTRUCTOR'>
            <state label='reset' element='STATEMENT' id='1'>
        </level>
        <level label='Analysis' element='CONSTRUCTOR'>
            <state label='operator_other' element='ATTRIBUTION' id='2'>
        </level>
        <level label='Analysis' element='DESTRUCTOR'>
        </level>
        <level label='getName' element='FUNCTION'>
            <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='3'>
            <state label='return' element='JUMP' id='4'>
        </level>
        <level label='=' element='OPERATOR'>
            <state label='reset' element='STATEMENT' id='5'>
            <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='6'>
            <state label='lock_guard_mutex_lock2_other_m_mutex' element='STATEMENT' id='7'>
            <state label='m_enableCache_other_m_enableCache' element='ATTRIBUTION' id='8'>
            <state label='m_projectPtr_other_m_projectPtr' element='ATTRIBUTION' id='9'>
            <state label='m_name_other_m_name' element='ATTRIBUTION' id='10'>
```

```
    <state label='m_outputDirectoryName_other_m_outputDirectoryName' element='ATTRIBUTION' id='11'>
    <state label='m_graph_other_m_graph' element='ATTRIBUTION' id='12'>
    <state label='m_graph_setAnalysis_this' element='STATEMENT' id='13'>
    <state label='m_context_other_m_context' element='ATTRIBUTION' id='14'>
    <state label='return' element='JUMP' id='15'>
</level>
<level label='setName' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='16'>
    <state label='m_name_newName' element='ATTRIBUTION' id='17'>
</level>
<level label='getGraph' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='18'>
    <state label='return' element='JUMP' id='19'>
</level>
<level label='getGraph' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='20'>
    <state label='return' element='JUMP' id='21'>
</level>
<level label='setGraph' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='22'>
    <state label='boost_filesystem_path_graphOutDir_m_outputDirectoryName' element='STATEMENT' id='23'>
    <state label='graphOutDir_newGraph_getName' element='STATEMENT' id='24'>
    <state label='m_graph_newGraph' element='ATTRIBUTION' id='25'>
    <level label='if' element='DECISION' id='26'>
        <state label='return' element='JUMP' id='27'>
    </level>
    <state label='m_graph_enableCache_m_enableCache' element='STATEMENT' id='28'>
    <state label='m_graph_setAnalysis_this' element='STATEMENT' id='29'>
    <state label='return' element='JUMP' id='30'>
</level>
<level label='getContext' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='31'>
    <state label='return' element='JUMP' id='32'>
</level>
<level label='setContext' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='33'>
    <state label='m_context_newContext' element='ATTRIBUTION' id='34'>
</level>
<level label='setOutputDirectoryName' element='FUNCTION'>
    <state label='boost_filesystem_path_completePath' element='STATEMENT' id='35'>
    <level label='try' element='EXCEPTION'>
        <state label='completePath_boost_filesystem_system_complete_outDirName' element='ATTRIBUTION' id='36'>
    </level>
    <level label='catch' element='EXCEPTION' id='37'>
        <state label='return' element='JUMP' id='38'>
    </level>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='39'>
    <state label='m_outputDirectoryName_completePath_string' element='ATTRIBUTION' id='40'>
    <state label='boost_filesystem_path_graphOutDir_m_outputDirectoryName' element='STATEMENT' id='41'>
    <state label='graphOutDir_m_graph_getName' element='STATEMENT' id='42'>
    <state label='return' element='JUMP' id='43'>
</level>
<level label='getOutputDirectoryName' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='44'>
    <state label='return' element='JUMP' id='45'>
</level>
<level label='enableCache' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='46'>
    <state label='m_enableCache_enabled' element='ATTRIBUTION' id='47'>
    <state label='m_graph_enableCache_enabled' element='STATEMENT' id='48'>
</level>
<level label='isCacheEnabled' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='49'>
    <state label='return' element='JUMP' id='50'>
</level>
<level label='reset' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='51'>
    <state label='m_enableCache_false' element='ATTRIBUTION' id='52'>
    <state label='m_projectPtr_0' element='ATTRIBUTION' id='53'>
    <state label='m_name_clear' element='STATEMENT' id='54'>
    <state label='m_outputDirectoryName_clear' element='STATEMENT' id='55'>
    <state label='m_graph_reset' element='STATEMENT' id='56'>
    <state label='m_context_reset' element='STATEMENT' id='57'>
</level>
<level label='setProject' element='FUNCTION'>
    <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='58'>
    <state label='m_projectPtr_project' element='ATTRIBUTION' id='59'>
```

```
        </level>
        <level label='getProject' element='FUNCTION'>
            <state label='lock_guard_mutex_lock_m_mutex' element='STATEMENT' id='60'>
            <state label='return' element='JUMP' id='61'>
        </level>
    </level>
</level>

<transition from = 'initial_-1' to = 'lock_guard_mutex_lock_m_mutex_51' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_51' to = 'm_enableCache_false_52' event = 'lambda'>
<transition from = 'm_enableCache_false_52' to = 'm_projectPtr_0_53' event = 'lambda'>
<transition from = 'm_projectPtr_0_53' to = 'm_name_clear_54' event = 'lambda'>
<transition from = 'm_name_clear_54' to = 'm_outputDirectoryName_clear_55' event = 'lambda'>
<transition from = 'm_outputDirectoryName_clear_55' to = 'm_graph_reset_56' event = 'lambda'>
<transition from = 'm_graph_reset_56' to = 'm_context_reset_57' event = 'lambda'>
<transition from = 'm_context_reset_57' to = 'reset_1' event = 'lambda'>
<transition from = 'reset_1' to = 'operator_other_2' event = 'lambda'>
<transition from = 'operator_other_2' to = 'lock_guard_mutex_lock_m_mutex_3' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_3' to = 'return_4' event = 'lambda'>
<transition from = 'return_4' to = 'lock_guard_mutex_lock_m_mutex_51' event = 'm_name'>
<transition from = 'm_context_reset_57' to = 'reset_5' event = 'lambda'>
<transition from = 'reset_5' to = 'lock_guard_mutex_lock_m_mutex_6' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_6' to = 'lock_guard_mutex_lock2_other_m_mutex_7' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock2_other_m_mutex_7' to = 'm_enableCache_other_m_enableCache_8' event = 'lambda'>
<transition from = 'm_enableCache_other_m_enableCache_8' to = 'm_projectPtr_other_m_projectPtr_9' event = 'lambda'>
<transition from = 'm_projectPtr_other_m_projectPtr_9' to = 'm_name_other_m_name_10' event = 'lambda'>
<transition from = 'm_name_other_m_name_10' to = 'm_outputDirectoryName_other_m_outputDirectoryName_11' event = 'lambda'>
<transition from = 'm_outputDirectoryName_other_m_outputDirectoryName_11' to = 'm_graph_other_m_graph_12' event = 'lambda'>
<transition from = 'm_graph_other_m_graph_12' to = 'm_graph_setAnalysis_this_13' event = 'lambda'>
<transition from = 'm_graph_setAnalysis_this_13' to = 'm_context_other_m_context_14' event = 'lambda'>
<transition from = 'm_context_other_m_context_14' to = 'return_15' event = 'lambda'>
<transition from = 'return_15' to = 'lock_guard_mutex_lock_m_mutex_16' event = 'return'>
<transition from = 'lock_guard_mutex_lock_m_mutex_16' to = 'm_name_newName_17' event = 'lambda'>
<transition from = 'm_name_newName_17' to = 'lock_guard_mutex_lock_m_mutex_18' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_18' to = 'return_19' event = 'lambda'>
<transition from = 'return_19' to = 'lock_guard_mutex_lock_m_mutex_20' event = 'm_graph'>
<transition from = 'lock_guard_mutex_lock_m_mutex_20' to = 'return_21' event = 'lambda'>
<transition from = 'return_21' to = 'lock_guard_mutex_lock_m_mutex_22' event = 'm_graph'>
<transition from = 'lock_guard_mutex_lock_m_mutex_22' to = 'boost_filesystem_path_graphOutDir_m_outputDirectoryName_23' event = 'lambda'>
<transition from = 'boost_filesystem_path_graphOutDir_m_outputDirectoryName_23' to = 'graphOutDir_newGraph_getName_24' event = 'lambda'>
<transition from = 'graphOutDir_newGraph_getName_24' to = 'm_graph_newGraph_25' event = 'lambda'>
<transition from = 'm_graph_newGraph_25' to = 'if_26' event = 'lambda'>
<transition from = 'if_26' to = 'return_27' event = 'TRUE'>
<transition from = 'if_26' to = 'm_graph_enableCache_m_enableCache_28' event = 'FALSE'>
<transition from = 'm_graph_enableCache_m_enableCache_28' to = 'm_graph_setAnalysis_this_29' event = 'lambda'>
<transition from = 'm_graph_setAnalysis_this_29' to = 'return_30' event = 'lambda'>
<transition from = 'return_27' to = 'lock_guard_mutex_lock_m_mutex_31' event = 'FALSE'>
<transition from = 'return_30' to = 'lock_guard_mutex_lock_m_mutex_31' event = 'TRUE'>
<transition from = 'lock_guard_mutex_lock_m_mutex_31' to = 'return_32' event = 'lambda'>
<transition from = 'return_32' to = 'lock_guard_mutex_lock_m_mutex_33' event = 'm_context'>
<transition from = 'lock_guard_mutex_lock_m_mutex_33' to = 'm_context_newContext_34' event = 'lambda'>
<transition from = 'm_context_newContext_34' to = 'boost_filesystem_path_completePath_35' event = 'lambda'>
<transition from = 'boost_filesystem_path_completePath_35' to = 'try_-1' event = 'lambda'>
<transition from = 'try_-1' to = 'completePath_boost_filesystem_system_complete_outDirName_36' event = 'lambda'>
<transition from = 'completePath_boost_filesystem_system_complete_outDirName_36' to = 'catch_37' event = 'exception'>
<transition from = 'catch_37' to = 'return_38' event = 'lambda'>
<transition from = 'completePath_boost_filesystem_system_complete_outDirName_36' to = 'lock_guard_mutex_lock_m_mutex_39' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_39' to = 'm_outputDirectoryName_completePath_string_40' event = 'lambda'>
<transition from = 'm_outputDirectoryName_completePath_string_40' to = 'boost_filesystem_path_graphOutDir_m_outputDirectoryName_41' event = 'lambda'>
<transition from = 'boost_filesystem_path_graphOutDir_m_outputDirectoryName_41' to = 'graphOutDir_m_graph_getName_42' event = 'lambda'>
<transition from = 'graphOutDir_m_graph_getName_42' to = 'return_43' event = 'lambda'>
<transition from = 'return_38' to = 'lock_guard_mutex_lock_m_mutex_44' event = 'FALSE'>
<transition from = 'return_43' to = 'lock_guard_mutex_lock_m_mutex_44' event = 'm_graph'>
<transition from = 'lock_guard_mutex_lock_m_mutex_44' to = 'return_45' event = 'lambda'>
<transition from = 'return_45' to = 'lock_guard_mutex_lock_m_mutex_46' event = 'm_outputDirectoryName'>
<transition from = 'lock_guard_mutex_lock_m_mutex_46' to = 'm_enableCache_enabled_47' event = 'lambda'>
<transition from = 'm_enableCache_enabled_47' to = 'm_graph_enableCache_enabled_48' event = 'lambda'>
<transition from = 'm_graph_enableCache_enabled_48' to = 'lock_guard_mutex_lock_m_mutex_49' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_49' to = 'return_50' event = 'lambda'>
<transition from = 'return_50' to = 'lock_guard_mutex_lock_m_mutex_51' event = 'm_enableCache'>
<transition from = 'm_context_reset_57' to = 'lock_guard_mutex_lock_m_mutex_58' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_58' to = 'm_projectPtr_project_59' event = 'lambda'>
<transition from = 'm_projectPtr_project_59' to = 'lock_guard_mutex_lock_m_mutex_60' event = 'lambda'>
<transition from = 'lock_guard_mutex_lock_m_mutex_60' to = 'return_61' event = 'lambda'>
<transition from = 'return_61' to = 'final_-1' event = 'm_projectPtr'>
```

A.3 Arquivo SMV

Código em SMV para o verificador de modelo NuSMV gerado pelo método Singularity a partir da classe "analysis.cpp":

```
MODULE main

    VAR
        state :
        {
            reset_1,
            operator_other_2,
            lock_guard_mutex_lock_m_mutex_3,
            return_4,
            reset_5,
            lock_guard_mutex_lock_m_mutex_6,
            lock_guard_mutex_lock2_other_m_mutex_7,
            m_enableCache_other_m_enableCache_8,
            m_projectPtr_other_m_projectPtr_9,
            m_name_other_m_name_10,
            m_outputDirectoryName_other_m_outputDirectoryName_11,
            m_graph_other_m_graph_12,
            m_graph_setAnalysis_this_13,
            m_context_other_m_context_14,
            return_15,
            lock_guard_mutex_lock_m_mutex_16,
            m_name_newName_17,
            lock_guard_mutex_lock_m_mutex_18,
            return_19,
            lock_guard_mutex_lock_m_mutex_20,
            return_21,
            lock_guard_mutex_lock_m_mutex_22,
            boost_filesystem_path_graphOutDir_m_outputDirectoryName_23,
            graphOutDir_newGraph_getName_24,
            m_graph_newGraph_25,
            if_26,
            return_27,
            m_graph_enableCache_m_enableCache_28,
            m_graph_setAnalysis_this_29,
            return_30,
            lock_guard_mutex_lock_m_mutex_31,
            return_32,
            lock_guard_mutex_lock_m_mutex_33,
            m_context_newContext_34,
            boost_filesystem_path_completePath_35,
            completePath_boost_filesystem_system_complete_outDirName_36,
            catch_37,
            return_38,
            lock_guard_mutex_lock_m_mutex_39,
            m_outputDirectoryName_completePath_string_40,
            boost_filesystem_path_graphOutDir_m_outputDirectoryName_41,
            graphOutDir_m_graph_getName_42,
            return_43,
            lock_guard_mutex_lock_m_mutex_44,
            return_45,
            lock_guard_mutex_lock_m_mutex_46,
            m_enableCache_enabled_47,
            m_graph_enableCache_enabled_48,
            lock_guard_mutex_lock_m_mutex_49,
            return_50,
            lock_guard_mutex_lock_m_mutex_51,
            m_enableCache_false_52,
            m_projectPtr_0_53,
            m_name_clear_54,
            m_outputDirectoryName_clear_55,
            m_graph_reset_56,
            m_context_reset_57,
            lock_guard_mutex_lock_m_mutex_58,
            m_projectPtr_project_59,
            lock_guard_mutex_lock_m_mutex_60,
            return_61
        };

        events :
```

```
    {
        null,
        m_name,
        return,
        m_graph,
        m_context,
        exception,
        m_outputDirectoryName,
        m_enableCache,
        m_projectPtr
    };

    decision : boolean;

ASSIGN

    init(events) := null;
    init(state) := lock_guard_mutex_lock_m_mutex_51;

    next(state) :=
        case
            (state = lock_guard_mutex_lock_m_mutex_51) : m_enableCache_false_52;
            (state = m_enableCache_false_52) : m_projectPtr_0_53;
            (state = m_projectPtr_0_53) : m_name_clear_54;
            (state = m_name_clear_54) : m_outputDirectoryName_clear_55;
            (state = m_outputDirectoryName_clear_55) : m_graph_reset_56;
            (state = m_graph_reset_56) : m_context_reset_57;
            (state = m_context_reset_57) : reset_1;
            (state = reset_1) : operator_other_2;
            (state = operator_other_2) : lock_guard_mutex_lock_m_mutex_3;
            (state = lock_guard_mutex_lock_m_mutex_3) : return_4;
            (state = return_4) : lock_guard_mutex_lock_m_mutex_51;
            (state = m_context_reset_57) : reset_5;
            (state = reset_5) : lock_guard_mutex_lock_m_mutex_6;
            (state = lock_guard_mutex_lock_m_mutex_6) : lock_guard_mutex_lock2_other_m_mutex_7;
            (state = lock_guard_mutex_lock2_other_m_mutex_7) : m_enableCache_other_m_enableCache_8;
            (state = m_enableCache_other_m_enableCache_8) : m_projectPtr_other_m_projectPtr_9;
            (state = m_projectPtr_other_m_projectPtr_9) : m_name_other_m_name_10;
            (state = m_name_other_m_name_10) : m_outputDirectoryName_other_m_outputDirectoryName_11;
            (state = m_outputDirectoryName_other_m_outputDirectoryName_11) : m_graph_other_m_graph_12;
            (state = m_graph_other_m_graph_12) : m_graph_setAnalysis_this_13;
            (state = m_graph_setAnalysis_this_13) : m_context_other_m_context_14;
            (state = m_context_other_m_context_14) : return_15;
            (state = return_15) : lock_guard_mutex_lock_m_mutex_16;
            (state = lock_guard_mutex_lock_m_mutex_16) : m_name_newName_17;
            (state = m_name_newName_17) : lock_guard_mutex_lock_m_mutex_18;
            (state = lock_guard_mutex_lock_m_mutex_18) : return_19;
            (state = return_19) : lock_guard_mutex_lock_m_mutex_20;
            (state = lock_guard_mutex_lock_m_mutex_20) : return_21;
            (state = return_21) : lock_guard_mutex_lock_m_mutex_22;
            (state = lock_guard_mutex_lock_m_mutex_22) : boost_filesystem_path_graphOutDir_m_outputDirectoryName_23;
            (state = boost_filesystem_path_graphOutDir_m_outputDirectoryName_23) : graphOutDir_newGraph_getName_24;
            (state = graphOutDir_newGraph_getName_24) : m_graph_newGraph_25;
            (state = m_graph_newGraph_25) : if_26;
            (state = if_26 & decision = TRUE) : return_27;
            (state = if_26 & decision = FALSE) : m_graph_enableCache_m_enableCache_28;
            (state = m_graph_enableCache_m_enableCache_28) : m_graph_setAnalysis_this_29;
            (state = m_graph_setAnalysis_this_29) : return_30;
            (state = return_27) : lock_guard_mutex_lock_m_mutex_31;
            (state = return_30) : lock_guard_mutex_lock_m_mutex_31;
            (state = lock_guard_mutex_lock_m_mutex_31) : return_32;
            (state = return_32) : lock_guard_mutex_lock_m_mutex_33;
            (state = lock_guard_mutex_lock_m_mutex_33) : m_context_newContext_34;
            (state = m_context_newContext_34) : boost_filesystem_path_completePath_35;
            (state = completePath_boost_filesystem_system_complete_outDirName_36) : catch_37;
            (state = catch_37) : return_38;
            (state = completePath_boost_filesystem_system_complete_outDirName_36) : lock_guard_mutex_lock_m_mutex_39;
            (state = lock_guard_mutex_lock_m_mutex_39) : m_outputDirectoryName_completePath_string_40;
            (state = m_outputDirectoryName_completePath_string_40) : boost_filesystem_path_graphOutDir_m_outputDirectoryName_41;
            (state = boost_filesystem_path_graphOutDir_m_outputDirectoryName_41) : graphOutDir_m_graph_getName_42;
            (state = graphOutDir_m_graph_getName_42) : return_43;
            (state = return_38) : lock_guard_mutex_lock_m_mutex_44;
            (state = return_43) : lock_guard_mutex_lock_m_mutex_44;
            (state = lock_guard_mutex_lock_m_mutex_44) : return_45;
            (state = return_45) : lock_guard_mutex_lock_m_mutex_46;
            (state = lock_guard_mutex_lock_m_mutex_46) : m_enableCache_enabled_47;
            (state = m_enableCache_enabled_47) : m_graph_enableCache_enabled_48;
```

```
                (state = m_graph_enableCache_enabled_48) : lock_guard_mutex_lock_m_mutex_49;
                (state = lock_guard_mutex_lock_m_mutex_49) : return_50;
                (state = return_50) : lock_guard_mutex_lock_m_mutex_51;
                (state = m_context_reset_57) : lock_guard_mutex_lock_m_mutex_58;
                (state = lock_guard_mutex_lock_m_mutex_58) : m_projectPtr_project_59;
                (state = m_projectPtr_project_59) : lock_guard_mutex_lock_m_mutex_60;
                (state = lock_guard_mutex_lock_m_mutex_60) : return_61;
                TRUE : state;
            esac;

        next(events) :=
            case
                (state = return_4) : m_name;
                (state = return_15) : return;
                (state = return_19) : m_graph;
                (state = return_21) : m_graph;
                (state = return_32) : m_context;
                (state = completePath_boost_filesystem_system_complete_outDirName_36) : exception;
                (state = return_43) : m_graph;
                (state = return_45) : m_outputDirectoryName;
                (state = return_50) : m_enableCache;
                (state = return_61) : m_projectPtr;
                TRUE : events;
            esac;

        next(decision) :=
            case
                (state = return_27) : FALSE;
                (state = return_30) : TRUE;
                (state = return_38) : FALSE;
                TRUE : {TRUE, FALSE};
            esac;

CTLSPEC
   AG (events != m_name)
CTLSPEC
   AG (events != return)
CTLSPEC
   AG (events != m_graph)
CTLSPEC
   AG (events != m_context)
CTLSPEC
   AG (events != exception)
CTLSPEC
   AG (events != m_outputDirectoryName)
CTLSPEC
   AG (events != m_enableCache)
CTLSPEC
   AG (events != m_projectPtr)

CTLSPEC
    AG (state = if_26 & decision = TRUE -> EX state != return_27)
CTLSPEC
    AG (state = if_26 & decision = FALSE -> EX state != m_graph_enableCache_m_enableCache_28)

CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_51 -> EX state != m_enableCache_false_52)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_51 -> EX state = m_enableCache_false_52)
CTLSPEC
    AG (state = m_enableCache_false_52 -> EX state != m_projectPtr_0_53)
CTLSPEC
   AG (state != m_enableCache_false_52 -> EX state = m_projectPtr_0_53)
CTLSPEC
    AG (state = m_projectPtr_0_53 -> EX state != m_name_clear_54)
CTLSPEC
   AG (state != m_projectPtr_0_53 -> EX state = m_name_clear_54)
CTLSPEC
    AG (state = m_name_clear_54 -> EX state != m_outputDirectoryName_clear_55)
CTLSPEC
   AG (state != m_name_clear_54 -> EX state = m_outputDirectoryName_clear_55)
CTLSPEC
    AG (state = m_outputDirectoryName_clear_55 -> EX state != m_graph_reset_56)
CTLSPEC
   AG (state != m_outputDirectoryName_clear_55 -> EX state = m_graph_reset_56)
CTLSPEC
    AG (state = m_graph_reset_56 -> EX state != m_context_reset_57)
```

```
CTLSPEC
   AG (state != m_graph_reset_56 -> EX state = m_context_reset_57)
CTLSPEC
    AG (state = m_context_reset_57 -> EX state != reset_1)
CTLSPEC
   AG (state != m_context_reset_57 -> EX state = reset_1)
CTLSPEC
    AG (state = reset_1 -> EX state != operator_other_2)
CTLSPEC
   AG (state != reset_1 -> EX state = operator_other_2)
CTLSPEC
    AG (state = operator_other_2 -> EX state != lock_guard_mutex_lock_m_mutex_3)
CTLSPEC
   AG (state != operator_other_2 -> EX state = lock_guard_mutex_lock_m_mutex_3)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_3 -> EX state != return_4)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_3 -> EX state = return_4)
CTLSPEC
    AG (state = return_4 -> EX state != lock_guard_mutex_lock_m_mutex_51)
CTLSPEC
   AG (state != return_4 -> EX state = lock_guard_mutex_lock_m_mutex_51)
CTLSPEC
    AG (state = m_context_reset_57 -> EX state != reset_5)
CTLSPEC
   AG (state != m_context_reset_57 -> EX state = reset_5)
CTLSPEC
    AG (state = reset_5 -> EX state != lock_guard_mutex_lock_m_mutex_6)
CTLSPEC
   AG (state != reset_5 -> EX state = lock_guard_mutex_lock_m_mutex_6)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_6 -> EX state != lock_guard_mutex_lock2_other_m_mutex_7)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_6 -> EX state = lock_guard_mutex_lock2_other_m_mutex_7)
CTLSPEC
    AG (state = lock_guard_mutex_lock2_other_m_mutex_7 -> EX state != m_enableCache_other_m_enableCache_8)
CTLSPEC
   AG (state != lock_guard_mutex_lock2_other_m_mutex_7 -> EX state = m_enableCache_other_m_enableCache_8)
CTLSPEC
    AG (state = m_enableCache_other_m_enableCache_8 -> EX state != m_projectPtr_other_m_projectPtr_9)
CTLSPEC
   AG (state != m_enableCache_other_m_enableCache_8 -> EX state = m_projectPtr_other_m_projectPtr_9)
CTLSPEC
    AG (state = m_projectPtr_other_m_projectPtr_9 -> EX state != m_name_other_m_name_10)
CTLSPEC
   AG (state != m_projectPtr_other_m_projectPtr_9 -> EX state = m_name_other_m_name_10)
CTLSPEC
    AG (state = m_name_other_m_name_10 -> EX state != m_outputDirectoryName_other_m_outputDirectoryName_11)
CTLSPEC
   AG (state != m_name_other_m_name_10 -> EX state = m_outputDirectoryName_other_m_outputDirectoryName_11)
CTLSPEC
    AG (state = m_outputDirectoryName_other_m_outputDirectoryName_11 -> EX state != m_graph_other_m_graph_12)
CTLSPEC
   AG (state != m_outputDirectoryName_other_m_outputDirectoryName_11 -> EX state = m_graph_other_m_graph_12)
CTLSPEC
    AG (state = m_graph_other_m_graph_12 -> EX state != m_graph_setAnalysis_this_13)
CTLSPEC
   AG (state != m_graph_other_m_graph_12 -> EX state = m_graph_setAnalysis_this_13)
CTLSPEC
    AG (state = m_graph_setAnalysis_this_13 -> EX state != m_context_other_m_context_14)
CTLSPEC
   AG (state != m_graph_setAnalysis_this_13 -> EX state = m_context_other_m_context_14)
CTLSPEC
    AG (state = m_context_other_m_context_14 -> EX state != return_15)
CTLSPEC
   AG (state != m_context_other_m_context_14 -> EX state = return_15)
CTLSPEC
    AG (state = return_15 -> EX state != lock_guard_mutex_lock_m_mutex_16)
CTLSPEC
   AG (state != return_15 -> EX state = lock_guard_mutex_lock_m_mutex_16)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_16 -> EX state != m_name_newName_17)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_16 -> EX state = m_name_newName_17)
CTLSPEC
    AG (state = m_name_newName_17 -> EX state != lock_guard_mutex_lock_m_mutex_18)
CTLSPEC
```

```
   AG (state != m_name_newName_17 -> EX state = lock_guard_mutex_lock_m_mutex_18)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_18 -> EX state != return_19)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_18 -> EX state = return_19)
CTLSPEC
    AG (state = return_19 -> EX state != lock_guard_mutex_lock_m_mutex_20)
CTLSPEC
   AG (state != return_19 -> EX state = lock_guard_mutex_lock_m_mutex_20)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_20 -> EX state != return_21)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_20 -> EX state = return_21)
CTLSPEC
    AG (state = return_21 -> EX state != lock_guard_mutex_lock_m_mutex_22)
CTLSPEC
   AG (state != return_21 -> EX state = lock_guard_mutex_lock_m_mutex_22)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_22 -> EX state != boost_filesystem_path_graphOutDir_m_outputDirectoryName_23)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_22 -> EX state = boost_filesystem_path_graphOutDir_m_outputDirectoryName_23)
CTLSPEC
    AG (state = boost_filesystem_path_graphOutDir_m_outputDirectoryName_23 -> EX state != graphOutDir_newGraph_getName_24)
CTLSPEC
   AG (state != boost_filesystem_path_graphOutDir_m_outputDirectoryName_23 -> EX state = graphOutDir_newGraph_getName_24)
CTLSPEC
    AG (state = graphOutDir_newGraph_getName_24 -> EX state != m_graph_newGraph_25)
CTLSPEC
   AG (state != graphOutDir_newGraph_getName_24 -> EX state = m_graph_newGraph_25)
CTLSPEC
    AG (state = m_graph_newGraph_25 -> EX state != if_26)
CTLSPEC
   AG (state != m_graph_newGraph_25 -> EX state = if_26)
CTLSPEC
    AG (state = if_26 -> EX state != return_27)
CTLSPEC
   AG (state != if_26 -> EX state = return_27)
CTLSPEC
    AG (state = if_26 -> EX state != m_graph_enableCache_m_enableCache_28)
CTLSPEC
   AG (state != if_26 -> EX state = m_graph_enableCache_m_enableCache_28)
CTLSPEC
    AG (state = m_graph_enableCache_m_enableCache_28 -> EX state != m_graph_setAnalysis_this_29)
CTLSPEC
   AG (state != m_graph_enableCache_m_enableCache_28 -> EX state = m_graph_setAnalysis_this_29)
CTLSPEC
    AG (state = m_graph_setAnalysis_this_29 -> EX state != return_30)
CTLSPEC
   AG (state != m_graph_setAnalysis_this_29 -> EX state = return_30)
CTLSPEC
    AG (state = return_27 -> EX state != lock_guard_mutex_lock_m_mutex_31)
CTLSPEC
   AG (state != return_27 -> EX state = lock_guard_mutex_lock_m_mutex_31)
CTLSPEC
    AG (state = return_30 -> EX state != lock_guard_mutex_lock_m_mutex_31)
CTLSPEC
   AG (state != return_30 -> EX state = lock_guard_mutex_lock_m_mutex_31)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_31 -> EX state != return_32)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_31 -> EX state = return_32)
CTLSPEC
    AG (state = return_32 -> EX state != lock_guard_mutex_lock_m_mutex_33)
CTLSPEC
   AG (state != return_32 -> EX state = lock_guard_mutex_lock_m_mutex_33)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_33 -> EX state != m_context_newContext_34)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_33 -> EX state = m_context_newContext_34)
CTLSPEC
    AG (state = m_context_newContext_34 -> EX state != boost_filesystem_path_completePath_35)
CTLSPEC
   AG (state != m_context_newContext_34 -> EX state = boost_filesystem_path_completePath_35)
CTLSPEC
    AG (state = completePath_boost_filesystem_system_complete_outDirName_36 -> EX state != catch_37)
CTLSPEC
   AG (state != completePath_boost_filesystem_system_complete_outDirName_36 -> EX state = catch_37)
```

```
CTLSPEC
    AG (state = catch_37 -> EX state != return_38)
CTLSPEC
   AG (state != catch_37 -> EX state = return_38)
CTLSPEC
    AG (state = completePath_boost_filesystem_system_complete_outDirName_36 -> EX state != lock_guard_mutex_lock_m_mutex_39)
CTLSPEC
   AG (state != completePath_boost_filesystem_system_complete_outDirName_36 -> EX state = lock_guard_mutex_lock_m_mutex_39)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_39 -> EX state != m_outputDirectoryName_completePath_string_40)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_39 -> EX state = m_outputDirectoryName_completePath_string_40)
CTLSPEC
    AG (state = m_outputDirectoryName_completePath_string_40 -> EX state != boost_filesystem_path_graphOutDir_m_outputDirectoryName_41)
CTLSPEC
   AG (state != m_outputDirectoryName_completePath_string_40 -> EX state = boost_filesystem_path_graphOutDir_m_outputDirectoryName_41)
CTLSPEC
    AG (state = boost_filesystem_path_graphOutDir_m_outputDirectoryName_41 -> EX state != graphOutDir_m_graph_getName_42)
CTLSPEC
   AG (state != boost_filesystem_path_graphOutDir_m_outputDirectoryName_41 -> EX state = graphOutDir_m_graph_getName_42)
CTLSPEC
    AG (state = graphOutDir_m_graph_getName_42 -> EX state != return_43)
CTLSPEC
   AG (state != graphOutDir_m_graph_getName_42 -> EX state = return_43)
CTLSPEC
    AG (state = return_38 -> EX state != lock_guard_mutex_lock_m_mutex_44)
CTLSPEC
   AG (state != return_38 -> EX state = lock_guard_mutex_lock_m_mutex_44)
CTLSPEC
    AG (state = return_43 -> EX state != lock_guard_mutex_lock_m_mutex_44)
CTLSPEC
   AG (state != return_43 -> EX state = lock_guard_mutex_lock_m_mutex_44)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_44 -> EX state != return_45)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_44 -> EX state = return_45)
CTLSPEC
    AG (state = return_45 -> EX state != lock_guard_mutex_lock_m_mutex_46)
CTLSPEC
   AG (state != return_45 -> EX state = lock_guard_mutex_lock_m_mutex_46)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_46 -> EX state != m_enableCache_enabled_47)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_46 -> EX state = m_enableCache_enabled_47)
CTLSPEC
    AG (state = m_enableCache_enabled_47 -> EX state != m_graph_enableCache_enabled_48)
CTLSPEC
   AG (state != m_enableCache_enabled_47 -> EX state = m_graph_enableCache_enabled_48)
CTLSPEC
    AG (state = m_graph_enableCache_enabled_48 -> EX state != lock_guard_mutex_lock_m_mutex_49)
CTLSPEC
   AG (state != m_graph_enableCache_enabled_48 -> EX state = lock_guard_mutex_lock_m_mutex_49)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_49 -> EX state != return_50)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_49 -> EX state = return_50)
CTLSPEC
    AG (state = return_50 -> EX state != lock_guard_mutex_lock_m_mutex_51)
CTLSPEC
   AG (state != return_50 -> EX state = lock_guard_mutex_lock_m_mutex_51)
CTLSPEC
    AG (state = m_context_reset_57 -> EX state != lock_guard_mutex_lock_m_mutex_58)
CTLSPEC
   AG (state != m_context_reset_57 -> EX state = lock_guard_mutex_lock_m_mutex_58)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_58 -> EX state != m_projectPtr_project_59)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_58 -> EX state = m_projectPtr_project_59)
CTLSPEC
    AG (state = m_projectPtr_project_59 -> EX state != lock_guard_mutex_lock_m_mutex_60)
CTLSPEC
   AG (state != m_projectPtr_project_59 -> EX state = lock_guard_mutex_lock_m_mutex_60)
CTLSPEC
    AG (state = lock_guard_mutex_lock_m_mutex_60 -> EX state != return_61)
CTLSPEC
   AG (state != lock_guard_mutex_lock_m_mutex_60 -> EX state = return_61)
```

A.4 Contraexemplos

Contraexemplos gerados pela execução do método Singularity a partir da classe "analysis.cpp":

```
Name of the input C++ file: test/GeoDMA_Selection/analysis
Number of states: 61
Number of events: 8
Number of decisions: 1
Number of state transitions: 63
Number of event transitions: 13
Number of total transitions: 76
Number of components: 17
..............................
Cyclomatic Complexity: 36
..............................
Number of Case One properties: 8
Number of Case Two properties: 2
Number of Case Three properties: 126
Total Number of properties: 136
..............................
Number of Counterexamples: 75
Number of Valid Counterexamples: 10
Number of Invalid Counterexamples: 65
Biggest number of states in a counterexample: 12
Smallest number of states in a counterexample: 3
Number of states used by valid counterexamples: 11
Number of transitions used by valid counterexamples: 11

---------------------- 1
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
---------------------- 2
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
---------------------- 3
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
---------------------- 4
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
```

```
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
STATE = m_graph_reset_56
DECISION = false
---------------------- 5
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
STATE = m_graph_reset_56
DECISION = false
7
STATE = m_context_reset_57
DECISION = false
---------------------- 6
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
STATE = m_graph_reset_56
DECISION = false
7
STATE = m_context_reset_57
DECISION = false
8
STATE = reset_1
DECISION = false
---------------------- 7
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
```

```
STATE = m_graph_reset_56
DECISION = false
7
STATE = m_context_reset_57
DECISION = false
8
STATE = reset_1
DECISION = false
9
STATE = operator_other_2
DECISION = false
---------------------- 8
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
STATE = m_graph_reset_56
DECISION = false
7
STATE = m_context_reset_57
DECISION = false
8
STATE = reset_1
DECISION = false
9
STATE = operator_other_2
DECISION = false
10
STATE = lock_guard_mutex_lock_m_mutex_3
DECISION = false
---------------------- 9
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
STATE = m_graph_reset_56
DECISION = false
7
STATE = m_context_reset_57
DECISION = false
8
STATE = reset_1
DECISION = false
9
STATE = operator_other_2
DECISION = false
10
STATE = lock_guard_mutex_lock_m_mutex_3
DECISION = false
11
STATE = return_4
DECISION = false
---------------------- 10
```

```
1
STATE = lock_guard_mutex_lock_m_mutex_51
DECISION = false
2
STATE = m_enableCache_false_52
DECISION = false
3
STATE = m_projectPtr_0_53
DECISION = false
4
STATE = m_name_clear_54
DECISION = false
5
STATE = m_outputDirectoryName_clear_55
DECISION = false
6
STATE = m_graph_reset_56
DECISION = false
7
STATE = m_context_reset_57
DECISION = false
8
STATE = reset_1
DECISION = false
9
STATE = operator_other_2
DECISION = false
10
STATE = lock_guard_mutex_lock_m_mutex_3
DECISION = false
11
STATE = return_4
DECISION = false
12
STATE = lock_guard_mutex_lock_m_mutex_51
EVENT = m_name
DECISION = false
```

PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

Teses e Dissertações (TDI)

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

Manuais Técnicos (MAN)

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

Notas Técnico-Científicas (NTC)

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programas de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

Relatórios de Pesquisa (RPQ)

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

Propostas e Relatórios de Projetos (PRP)

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

Publicações Didáticas (PUD)

Incluem apostilas, notas de aula e manuais didáticos.

Publicações Seriadas

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

Programas de Computador (PDC)

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. Aceitam-se tanto programas fonte quanto os executáveis.

Pré-publicações (PRE)

Todos os artigos publicados em periódicos, anais e como capítulos de livros.