

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE E INOVAÇÃO

LUCAS SANTOS COPETTI

**Análise Comparativa entre Ferramentas
para Auxílio na Modelagem de Testes a
partir da Metodologia de Partição dos
Domínios de Entrada**

Monografia de Conclusão de Curso apresentada
como requisito parcial para a obtenção do grau
de Especialista em Engenharia de Software e
Inovação

Orientador: Prof^ª. Dr^ª. Érika Fernandes Cota

Porto Alegre
2021

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Copetti, Lucas Santos

Análise Comparativa entre Ferramentas para Auxílio na Modelagem de Testes a partir da Metodologia de Partição dos Domínios de Entrada / Lucas Santos Copetti. – Porto Alegre: PPGC da UFRGS, 2021.

42 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Curso de Especialização em Engenharia de Software e Inovação, Porto Alegre, BR-RS, 2021. Orientador: Érika Fernandes Cota.

1. Partição dos domínios de entrada. 2. Ferramentas de automação de testes. 3. Testes combinatórios. I. Cota, Érika Fernandes. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos André Bulhões Mendes

Vice-Reitora: Prof^ª. Patricia Pranke

Pró-Reitora de Pós-Graduação: Prof. Júlio Otávio Jardim Barcellos

Diretora do Instituto de Informática: Prof^ª. Carla Maria Dal Sasso Freitas

Coordenador do PPGC: Prof^ª. Karin Becker

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“He who thinks a tool can solve all problems, has a new problem.”

— FELIPE KNORR KUHN

AGRADECIMENTOS

Agradeço, primeiramente, a minha família que sempre me incentivou a estudar e sempre me mostrou que a educação é a chave para um mundo melhor. Agradeço também a minha namorada, amiga e companheira Maria Cecília que sempre me anima e me motiva.

RESUMO

Um indicativo de qualidade de uma suíte de testes pode ser a quantidade de defeitos que ela é capaz de ajudar a detectar. Um desenvolvedor de testes experiente e com bom domínio do sistema pode ser capaz de criar testes com qualidade pois tende a ter um melhor discernimento sobre quais são os testes mais eficazes. Por outro lado, testadores menos experientes podem demorar a desenvolver essa expertise e os seus testes podem ter uma grande probabilidade de não detectar defeitos que possam vir a causar prejuízo ao usuário final e à própria organização desenvolvedora do *software*. Uma forma de amenizar esse problema, agregando qualidade aos testes (e por consequência ao *software*), é partir de uma metodologia com fundamentação científica para modelar os testes, como a Partição dos Domínios de Entrada. A proposta deste trabalho é analisar comparativamente diversas ferramentas que são voltadas à modelagem de testes baseada na técnica de partição de domínios. Uma busca inicial retornou um total de 22 ferramentas. Após uma análise preliminar, quatro ferramentas foram selecionadas para uma análise mais detalhada. A análise comparativa das ferramentas selecionadas foi feita com base em um conjunto de critérios técnicos e através da implementação de dois estudos de caso. Espera-se com esse trabalho ajudar a difundir essa técnica de teste e evidenciar ferramentas para o projeto e desenvolvimento de testes que a suportem, com isso, auxiliar testadores a produzir testes com maior qualidade.

Palavras-chave: Partição dos domínios de entrada. Ferramentas de automação de testes. Testes combinatórios.

Comparative Analysis Between Tools for Helping in Tests Modeling Through the Input Domain Partitioning Methodology.

ABSTRACT

An indicative of quality of a test suite can be the amount of defects which it is capable to help to detect. A test developer with expertise and with a good knowledge of the system can be able to create quality tests because he tends to have a good understanding about which are the more effective tests. On the other hand, less experienced testers may need time to acquire this expertise and his tests may have a high probability of not detecting defects that could come to cause losses to the end user and to the software developer company itself. A way to soften this issue, adding quality to the tests (and consequently to the software) is starting from a methodology with scientific basis to develop the tests, like the Input Domain Partitioning. The purpose of this work is to analyse comparatively some tools that are aimed to the test modeling based in this technique of domain partitioning. An initial search returned a total of 22 tools. After a preliminary analysis, four tools were selected for a more detailed analysis. The comparative analysis of the selected tools was based in a set of technical criteria and through the implementation of two case studies. It is expected with this work to help to spread this methodology and highlight tools for the project and development of tests based on it and, with this, help testers to produce tests with better quality.

Keywords: Input domain partitioning. Test automation tools. Combinatorial tests.

LISTA DE ABREVIATURAS E SIGLAS

ACTS	Automated Combinatorial Testing for Software
BC	Basic Choice
CSRC	Computer Security Resource Center
NIST	National Institute of Standards and Technology
TSL	Test Specification Language
XML	eXtensible Markup Language

LISTA DE FIGURAS

Figura 4.1 Restrições criadas no ACTS.	30
Figura 4.2 Modelagem do exemplo findElement descrito no Tcases.	30
Figura 4.3 Testes para o método findElement gerados com a ACTS usando restrições e todas as combinações.	31
Figura 4.4 Testes para o método findElement gerados com a ACTS usando <i>Basic Choice</i>	32
Figura 4.5 Restrições criadas para a ACTS para o método findElement	33
Figura 4.6 Testes gerados para o exemplo da sorveteria com ACTS.	34

LISTA DE TABELAS

Tabela 2.1	Resumo do esquema de partição para o método findElement	17
Tabela 2.2	Testes resultantes com o critério BC para o método findElement	18
Tabela 2.3	Testes resultantes com o critério BC para o método findElement após análise.	19
Tabela 2.4	Definição de valores de entrada e resultados esperados para o método findElement	20
Tabela 2.5	Instruções para o atendente da soverteria.	22
Tabela 3.1	Ferramentas gratuitas e sites onde elas podem ser encontradas.	26
Tabela 3.2	Conformidade com os critérios preliminares. As ferramentas que estão de acordo com os critérios receberam S (sim) enquanto as que não estão de acordo estão marcadas com N (não).	27
Tabela 4.1	Resumo das características das ferramentas em relação aos critérios comparativos.	36

SUMÁRIO

1 INTRODUÇÃO	11
2 FUNDAMENTAÇÃO TEÓRICA	13
2.1 Primeiro Exemplo de Aplicação da Técnica: Método findElement.	15
2.1.1 Discussão Sobre a Abordagem em que Considera-se as Restrições das Características Antes de Fazer as Combinações	20
2.2 Segundo Exemplo de Aplicação da Técnica: Exemplo da Soverteria.....	22
2.3 Trabalhos Relacionados.....	23
3 PROPOSTA	25
3.1 Levantamento e Análise Preliminar	25
3.2 Ferramentas Seleccionadas para Estudo Após a Análise Preliminar	27
4 AVALIAÇÃO EXPERIMENTAL	28
4.1 Definição de Critérios Comparativos	28
4.2 Geração de Testes com as Ferramentas para o Método findElement	29
4.3 Geração de Testes com as Ferramentas para o Exemplo da Sorveteria	31
4.4 Observações Sobre as Ferramentas a Respeito dos Critérios Comparativos....	35
5 CONCLUSÃO	38
REFERÊNCIAS	41

1 INTRODUÇÃO

Um indicativo de qualidade de uma suíte de testes pode ser a quantidade de defeitos que ela é capaz de ajudar a identificar. Um desenvolvedor de testes experiente e com bom domínio do sistema pode ser capaz de criar testes com qualidade pois tende a ter um melhor discernimento do meio mais eficiente de testar. Isso é, ele pode ser capaz de ativar uma boa quantidade de possíveis falhas de um determinado *software* com um tamanho razoável de cenários de teste. Por outro lado, testadores menos experientes, com menor conhecimento sobre um sistema podem demorar a desenvolver essa perícia de saber como testar o *software*. Na pior das hipóteses, até que ele tenha um bom domínio do sistema, os seus testes podem ter uma grande probabilidade de não detectar defeitos que possam vir a causar prejuízo ao usuário final e à própria organização desenvolvedora do *software*.

Para amenizar esse problema, agregando qualidade aos testes (e por consequência ao *software*), é possível partir de uma metodologia com fundamentação científica para modelar os testes. O uso de uma abordagem metódica pode auxiliar os projetistas de teste (dos menos experientes aos mais experientes) a identificar a variabilidade dos elementos envolvidos na execução de uma dada funcionalidade, e com isso selecionar combinações de valores de entrada relevantes para os casos de teste (PEZZÈ; YOUNG, 2008).

Um estudo mostra que em torno da metade das falhas de *software* tendem a ser causadas pela interação entre dois ou mais fatores (KUHN; WALLACE; GALLO, 2004). Uma forma de encontrar os defeitos que geram essas falhas é testando o comportamento do sistema quando estimulado por diferentes combinações entre dois, ou mais, parâmetros de entrada. Em sistemas críticos, em que falhas podem resultar em acidentes catastróficos, seria interessante testar todas as possibilidades a fim de encontrar todos os defeitos, mas ao mesmo tempo inviável devido à grande quantidade de possíveis combinações de parâmetros que os sistemas geralmente possuem. Então é preciso selecionar um conjunto de combinações de alguma forma. Em (HAGAR et al., 2015) é destacado que gerar uma suíte de testes com combinações de quatro a seis parâmetros pode ser tão eficiente quanto testar todas as combinações possíveis de entrada do sistema.

Elaborar os casos de teste para detectar falhas envolvendo múltiplos parâmetros pode ser difícil, ou até impraticável, se elaborado manualmente. O uso de Teste Combinatório nos últimos anos vem sendo estudado e aplicado como solução para viabilizar essa tarefa de forma automática, ou pelo menos semi-automática. Estudos empíricos sobre a aplicação da técnica de Teste Combinatório e na comparação em relação a outros métodos

indicam resultados positivos obtidos na aplicação da técnica tais como a diminuição no esforço (menor quantidade de casos de teste que precisam ser criados, revisados, executados e analisados) e a capacidade da suíte de testes em detectar defeitos que se mostrou semelhante ou, em alguns casos, superior ao se comparar com outros métodos (LI et al., 2016), (BHARGAVI et al., 2016), (OZCAN, 2019) e (HU et al., 2020).

Para a aplicação da técnica de Teste Combinatórios ser efetiva, é importante que os parâmetros combinados sejam bem escolhidos. Isso porque os diferentes algoritmos de Teste Combinatório recebem como entrada os parâmetros e fazem as combinações, sejam esses parâmetros relevantes ou não. A efetividade da suíte de teste produzida com essas combinações é dependente da qualidade dos parâmetros escolhidos. Por isso, mostra-se importante o conhecimento e a aplicação da técnica de Particionamento do Espaço de Entrada (AMMANN; OFFUTT, 2016), que é uma metodologia para escolher parâmetros representativos de forma a cobrir todo o domínio de entrada do sistema.

A proposta deste trabalho é analisar comparativamente diversas ferramentas (principalmente as gratuitas) que são voltadas à automação no processo de modelagem de Teste Combinatório, nas quais podem ser aplicados os conceitos de Particionamento do Espaço de Entrada para se definir os parâmetros que serão combinados. Espera-se com esse trabalho ajudar a difundir esse método e evidenciar ferramentas que possam ser úteis às companhias que buscam incluir o processo de modelagem de Testes Combinatórios no ciclo de vida do *software*.

O texto está organizado da seguinte maneira: no Capítulo 2 são revisados os conceitos fundamentais além de trabalhos anteriores relacionados ao tema. No Capítulo 3 é apresentada a proposta de trabalho e é feita uma triagem a fim selecionar ferramentas para a análise comparativa. No Capítulo 4 são definidos critérios de comparação e é realizada a análise comparativa entre as ferramentas. São demonstrados dois exemplos de aplicação da técnica com as ferramentas selecionadas. Nos dois exemplos as ferramentas selecionadas por critérios técnicos são usadas. Contudo, no primeiro exemplo a técnica é seguida enquanto que no segundo exemplo não, a fim de demonstrar a importância da técnica na utilização das ferramentas. No Capítulo 5 conclui-se o trabalho resumindo as observações da análise e sugerindo trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Teste baseado na funcionalidade – também chamado de Teste Caixa Preta, ou Teste Funcional – é o tipo de teste baseado somente na especificação funcional do software, ou do sistema. Isso significa que nesse tipo de teste não são levados em conta aspectos da implementação como arquitetura e códigos-fonte. O desenvolvimento desses testes pode ser completamente manual ou com automatização. A abordagem padrão para gerar esses testes é particionar o espaço de entrada de uma função sendo testada e escolher entradas conforme cada bloco de partição.

Ostrand e Balcer (1988) definiram um método para, de forma sistemática, especificar e gerar Testes Funcionais chamado Método Partição-Categoria (*Category-Partition Method*) e apresentaram uma linguagem para implementar esse método (*Test Specification Language, TSL*). O primeiro passo da abordagem consiste em analisar a especificação para identificar, para cada unidade funcional, parâmetros e características desses parâmetros. Ainda na primeira etapa deve-se classificar esses itens em categorias conforme o impacto delas no comportamento da unidade funcional. Na próxima etapa é feita a partição das categorias em escolhas que são determinadas pelo testador. Em seguida, o testador deve determinar restrições entre as escolhas, pois é ele que determina como a ocorrência das escolhas se afetam. Depois deve-se escrever e processar a especificação de teste, escrita em TSL, com uma ferramenta para fazer as combinações das escolhas e gerar conjuntos de *test frames* – que são basicamente as combinações das escolhas que podem ser usadas para gerar os casos de teste. Então o testador avalia a saída gerada pela ferramenta e determina se é preciso mais ou menos *test frames*. Caso a saída gerada pela ferramenta não seja satisfatória é recomendado voltar para a etapa de determinar restrições entre as escolhas. Caso o testador esteja satisfeito com os *test frames* gerados ele pode transformar em *scripts* de teste os casos de teste resultantes a partir dos *test frames*.

PEZZÈ e YOUNG (2008) e AMMANN e OFFUTT (2016) apresentam as técnicas Teste Combinatório (*Combinatorial Testing*) e Particionamento do Espaço de Entrada (*Input Space Partitioning*), respectivamente, que separam as atividades de análise e elaboração de testes em etapas que podem ser quantificadas e parcialmente suportadas por ferramentas. Ambos discorrem sobre como particionar o domínio de entrada e encontrar combinações de valores de forma a gerar casos de teste significativos e com o tamanho previsível e adequado. No entanto, AMMANN e OFFUTT (2016) apresentam de forma mais detalhada e sistemática a parte da modelagem do espaço de entrada e apon-

tam mais opções de métodos para realizar as combinações, se comparado ao apresentado em PEZZÈ e YOUNG (2008) (estes últimos focam em *Pair-Wise* enquanto que AMMANN e OFFUTT (2016) apresentam mais tipos de combinações). Apesar de utilizarem terminologias diferentes e focarem em detalhes distintos, eles e Ostrand e Balcer (1988) podem ser vistos como complementares entre si na fundamentação teórica da modelagem de teste a partir da combinação de parâmetros do espaço de entrada e servem como base para compreensão do funcionamento das ferramentas e para o entendimento dos artigos científicos estudados durante o desenvolvimento deste trabalho de conclusão de curso.

O particionamento dos domínios de entrada pode ser entendido e resumido em cinco passos para geração de testes. São eles:

1. **Primeiro Passo: Identificar funções testáveis.**
2. **Segundo Passo: Encontrar todos os parâmetros.**
3. **Terceiro Passo: Definir o domínio de entrada.**
4. **Quarto Passo: Aplicar um critério de teste para escolher combinações de valores.**
5. **Quinto Passo: Refinar as combinações de blocos em entradas de teste.**

As funções testáveis, no **primeiro passo**, podem ser métodos de classes, funcionalidades de programas ou até funcionalidades de sistemas compostos por *hardware* e *software*.

No **segundo passo** é importante que sejam identificados todos os parâmetros que podem influenciar no comportamento da funcionalidade testada. Em métodos e componentes de *software* os parâmetros podem ser os parâmetros dos métodos e variáveis de estado. Para sistemas pode se considerar qualquer entrada no sistema que afeta o seu comportamento.

No **terceiro passo** a modelagem é feita com os parâmetros levantados no passo anterior. Para esses parâmetros devem-se identificar as suas características. Cada característica levará a uma partição do domínio de entrada daquele parâmetro. Essas partições são compostas por conjuntos de blocos que são conjuntos de valores interessantes que cada parâmetro pode assumir durante o teste. O conjunto desses blocos devem cobrir todo o domínio de entrada. Outra propriedade que eles devem satisfazer é que eles devem ser disjuntos dois a dois, ou seja, cada entrada deve pertencer claramente a um único bloco de uma característica. A modelagem pode seguir a abordagem baseada em interface ou em funcionalidade. A primeira considera cada parâmetro da interface separadamente

enquanto a segunda se baseia na identificação de características relacionadas às funcionalidades.

O **quarto passo** consiste em decidir um tipo de combinação entre valores de parâmetros para resultar em testes interessantes ao invés de gerar todos os testes possíveis. Essa etapa é muito importante pois é preciso ponderar os custos e os riscos de escolher critérios de combinação que gerem mais ou menos testes.

Fazer as combinações de forma manual pode ser impraticável (ou muito caro) dependendo da quantidade de combinações possíveis. Então pode ser mais interessante fazer uso de ferramentas para automatizar esse processo. Grande parte das ferramentas para isso se baseiam nos critérios de teste **Pair-Wise** e **T-Wise** (AMMANN; OFFUTT, 2016). Uma das ferramentas possibilita também o uso do critério **Basic Choice**.

No critério de teste **Basic Choice** é escolhido um bloco para cada característica para formar um teste básico. Outros testes são gerados mantendo todos os blocos iguais à escolha básica com a exceção de um bloco que deve mudar. Para cada teste um bloco diferente é mudado até que todos os blocos tenham sido mudados uma vez. Já no **T-Wise**, não existe valores base, nele é feita a combinação entre todos valores de blocos divididos em grupos de **T** partições. Se o **T** for igual a 2, o critério é equivalente ao **Pair-Wise** em que um valor de cada bloco de característica é combinado com um valor de cada bloco de cada outra característica. As ferramentas geralmente denominam o **T** como sendo a força da combinação. Quando a força for 2 equivale ao critério **Pair-Wise** e quando a força for igual ao número total de partições equivale ao critério **All Combinations** que seria a combinação "força bruta" entre todos os blocos de todas as características.

No **quinto passo** é feita a escolha de valores concretos para os parâmetros. Depois disso os testes já podem ser implementados.

2.1 Primeiro Exemplo de Aplicação da Técnica: Método findElement.

Um exemplo de aplicação da técnica de **Particionamento do Espaço de Entrada** pode demonstrar de forma clara essas etapas da técnica. Ele consiste em aplicar as etapas com a finalidade de gerar testes para o método **findElement**¹. Esse método está descrito no código a seguir:

¹O exemplo resumido nesta seção foi apresentado e discutido durante as aulas da disciplina de Teste de *Software* deste curso de Especialização em Engenharia de *Software* e Inovação

```
public boolean findElement (List list, Object element)
// Effects:
//     if list or element is null throw
//     NullPointerException
//     else return true if elements is in the list,
//     false otherwise
```

1. Identificar Funções Testáveis. Obviamente, a função testável do exemplo é o método `findElement`.

2. Encontrar Todos os Parâmetros. Os parâmetros são os próprios parâmetros da assinatura do método (`list` e `element`).

3. Definir o Domínio de Entrada. As possíveis características dos parâmetros foram identificadas da seguinte forma:

- Característica A: `list` é `null`?
- Característica B: `list` é vazia?
- Característica C: `element` é `null`?

Cada uma das características leva a uma partição do domínio com dois blocos (B1: Verdadeiro ou B2: Falso). Importante verificar que o esquema de partição contempla todo o domínio de entrada. Em outras palavras, todas as possíveis entradas para cada parâmetro estão representadas pelos blocos identificados.

Outra possível característica (que não será usada nesse exemplo) seria relacionada a homogeneidade da `list`. Como não está definido o comportamento para essa característica, em uma aplicação real seria o caso em que o desenvolvedor do teste precisa interagir com o desenvolvedor do *software* ou do sistema para decidir se essa característica é relevante para o teste. Talvez essa característica realmente não importe, ou ela importa e talvez tenha faltado requisito especificando esse comportamento. Por isso é importante que exista interação entre o testador e o resto da equipe para que esse tipo de decisão seja feita considerando os riscos associados.

As características e os blocos identificados até aqui foram encontrados seguindo a **Modelagem do Espaço de Entrada Baseado na Interface**. A seguir serão identificadas características e blocos conforme a **Modelagem do Espaço de Entrada Baseada em Funcionalidade**. Somando às características identificadas anteriormente, podem-se adicionar as seguintes:

- Característica D: **element** pertence a **list**?
- Característica E: número de ocorrências de **element** em **list**.
- Característica F: **element** é o primeiro?
- Característica G: **element** é o último?

Assim como as características A, B e C, as características D, F e G tem apenas dois blocos (B1: Verdadeiro ou B2: Falso). A característica E possui três blocos (B1: 0, B2: 1, B3: + de 1). Esses blocos foram escolhidos pois foram considerados relevantes para representar o espaço de entrada. Todavia, outros blocos poderiam ter sido escolhidos se fossem considerados interessantes. A Tabela 2.1 resume as características e seus respectivos blocos. O *alias* será utilizado como forma de simplificar as explicações dos passos a seguir.

Tabela 2.1: Resumo do esquema de partição para o método **findElement**.

Característica	Alias	B1	B2	B3
list é null ?	A	'V'	'F'	
list é vazia?	B	'V'	'F'	
element é null ?	C	'V'	'F'	
element pertence a list ?	D	'V'	'F'	
número de ocorrências de element em list .	E	'0'	'1'	'+ de 1'
element é o primeiro?	F	'V'	'F'	
element é o último?	G	'V'	'F'	

Fonte: O autor.

Os testes baseados nessa tabela serão descritos por um conjunto de blocos representados por dois caracteres. O primeiro indica o *Alias* de uma característica e o segundo é o número de um bloco da mesma característica. Como por exemplo, pode haver um teste com os seguintes blocos A2, B2, C2, D1, E3, F2 e G2. Isso pode ser traduzido da seguinte forma: um teste que recebe como entrada uma lista não nula e não vazia, um elemento não nulo que pertence à lista, ocorre mais de uma vez mas não ocorre na primeira nem na última posição.

4. Aplicar um Critério de Teste para Escolher Combinações de Valores. A escolha do critério de teste para escolher as combinações dos valores pode partir de duas abordagens. Em uma delas se considera primeiro as combinações das características e depois as restrições entre elas. Na outra aplicam-se as restrições e só depois fazem-se as combinações. Para este exemplo será considerada inicialmente a primeira abordagem. A segunda abordagem será discutida em uma sub-seção dedicada.

Partindo das características, primeiro calculam-se as combinações possíveis para facilitar as escolhas do critério de teste. Considerando todas as combinações possíveis

das características (**All Combinations**), temos um total de 192 testes possíveis. Esse primeiro cálculo é apenas para ter noção da quantidade máxima de testes caso se queira testar usando "força bruta". Aqui no exemplo 192 testes é considerado um número muito grande de testes, mas caso a quantidade total de testes fosse pequena talvez fosse menos trabalhoso usar todas combinações do que gastar tempo decidindo qual tipo de combinação é melhor.

Calculando-se para **Pair-Wise**, **Each Choice** e **Basic Choice (BC)** (AMMANN; OFFUTT, 2016) encontram-se os totais de 6, 3 e 9 testes, respectivamente. Nesse exemplo, portanto, o **BC** aparenta ser a melhor opção, pois além de ter uma quantidade razoável de testes ele permite partir de um caso válido, que faz sentido, e ir experimentando a influência das diferentes características.

O primeiro passo para a combinação **BC** é escolher uma combinação factível e válida. Essa escolha pode ser a seguinte para o exemplo em questão: A2, B2, C2, D1, E2, F1 e G2. Ou seja, um teste que recebe como entrada uma lista não nula e não vazia, um elemento não nulo que pertence à lista, ocorre uma vez e é o primeiro elemento da lista. A tradução dos *alias* e dos valores dos blocos podem ser consultados na Tabela 2.1.

Dispondo de um caso base, é possível gerar os outros testes. Para cada teste é mudado apenas um bloco de uma característica. O resultado pode ser visto na Tabela 2.2. Na primeira coluna os 'T's indicam a numeração dos testes. Em negrito está destacado o bloco que mudou em cada teste.

Tabela 2.2: Testes resultantes com o critério **BC** para o método **findElement**.

BC	A2	B2	C2	D1	E2	F1	G2
T2	A1	B2	C2	D1	E2	F1	G2
T3	A2	B1	C2	D1	E2	F1	G2
T4	A2	B2	C1	D1	E2	F1	G2
T5	A2	B2	C2	D2	E2	F1	G2
T6	A2	B2	C2	D1	E1	F1	G2
T7	A2	B2	C2	D1	E3	F1	G2
T8	A2	B2	C2	D1	E2	F2	G2
T9	A2	B2	C2	D1	E2	F1	G1

Fonte: O autor.

As combinações resultantes precisam ser analisadas manualmente pois nem todos os testes gerados são factíveis. Em T2, se a lista é nula não faz sentido testar se um elemento pertence a essa lista, então a escolha dos outros blocos não importam. Em T3, seguindo o procedimento do critério **BC**, mantém A em sua escolha básica e B é mudado para lista vazia que, assim como em T2, os outros valores não importam. Em T4, novamente, pode-se considerar que os outros blocos não importam ao mudar para C1 onde

a lista é vazia. Os testes T2, T3 e T4 exploram os parâmetros quanto às características relacionadas aos seus tipos e podem ser considerados testes de sanidade pois avaliam se o comportamento do *software* é correto em situações com entradas inválidas. Já os próximos testes possibilitam explorar aspectos da funcionalidade do método **findElement**.

No teste T5 o elemento não pertence à lista. Esse caso não é um teste de sanidade pois é uma entrada válida, não é um caso de exceção, então os blocos E, F e G importam e precisam ser analisados. No caso da Tabela 2.2 o T5 acaba não sendo um teste factível pois se o elemento não pertence à lista (D2) ele não tem um número de ocorrência nem pertence à uma posição específica. Para que T5 seja factível, deve-se mudar E1 para E2 e F1 para F2, ou seja, agora o elemento em T5 não pertence à lista, tem zero ocorrências e não é o primeiro nem o último da lista. T6 também não é factível e precisa ser modificado, pois se o elemento pertence à lista o número de ocorrências não vai ser zero. Mudando E1 (zero ocorrências) para E2 (uma ocorrência) em T6 resulta em um teste igual à escolha básica, que não faz sentido testar novamente. Mudando E1 (zero ocorrências) para E3 (mais de uma ocorrências) pode ser interessante, mas T6 se tornaria igual ao T7. Então uma alternativa para o T6 seria mudar E1 (zero ocorrências) para E3 (mais de uma ocorrências) e G2 (elemento é o último) para G1 (elemento é o primeiro).

A Tabela 2.3 mostra a seleção resultante após a análise das combinações. Os 'X' indicam blocos *don't care* cuja escolha não faz diferença para aquele teste. Foram mantidos os blocos C2 nos testes T2 e T3, pois dessa forma pode-se garantir que **list** e **element** são testados de forma independente entre si nos testes de sanidade. Dessa forma é possível distinguir de forma clara as seguintes situações. Em T2 é corretamente levantada a exceção por causa de a lista ser nula. Em T3 é corretamente levanta a exceção para quando a lista é vazia (dependendo de como for definido esse caso). Em T4 é levantada corretamente a exceção por causa de o elemento ser nulo.

Tabela 2.3: Testes resultantes com o critério **BC** para o método **findElement** após análise.

BC	A2	B2	C2	D1	E2	F1	G2
T2	A1	X	C2	X	X	X	X
T3	A2	B1	C2	X	X	X	X
T4	A2	B2	C1	X	X	X	X
T5	A2	B2	C2	D2	E1	F2	G2
T6	A2	B2	C2	D1	E3	F1	G1
T7	A2	B2	C2	D1	E3	F1	G2
T8	A2	B2	C2	D1	E2	F2	G2
T9	A2	B2	C2	D1	E2	F1	G1

Fonte: O autor.

5. Refinar as Combinações de Blocos em Entradas de Teste. Após definirem-

se as combinações dos blocos segundo algum dos critérios é preciso definir valores de entrada e de saída. O método auxilia na escolhas das combinações e deixa ao testador o papel de decidir quais são de fato os valores a serem combinados. A Tabela 2.4 exemplifica escolhas de valores de entrada e saída. Note que em "T3" o resultado esperado está com um ponto de interrogação, pois este comportamento não está definido. O testador nesse caso deve então elucidar essa dúvida pois isso pode indicar alguma deficiência de requisito ou de arquitetura.

Tabela 2.4: Definição de valores de entrada e resultados esperados para o método **findElement**.

Teste	list	element	Resultado Esperado
T1	[1, 2, 3, 4, 5]	1	True
T2	null	2	NullPointerException
T3	[]	2	False (?)
T4	[1, 2]	Null	NullPointerException
T5	[1, 2]	5	False
T6	[1, 2, 1]	1	True
T7	[1, 1, 2]	1	True
T8	['a', 'z', 'a']	'z'	True
T9	[1]	1	True

Fonte: O autor.

Os testes gerados no exemplo com a técnica não são necessariamente todos os testes relevantes possíveis. A forma como o testador define o domínio de entrada influencia na qualidade dos testes gerados. Portanto cabe ao testador analisar e decidir se os testes gerados são suficientes. O método auxilia na escolha dos testes relevantes sem impedir que sejam inseridos mais testes além dos gerados caso o responsável pelos teste julgue necessário. No exemplo, seria possível inserir um décimo teste em que o elemento está na última posição da lista, supondo que seja julgado que essa é uma situação relevante a ser testada.

2.1.1 Discussão Sobre a Abordagem em que Considera-se as Restrições das Características Antes de Fazer as Combinações

Os três primeiros passos do método podem ser seguidos da mesma forma que na abordagem da seção anterior. A diferença dá-se no quarto passo (**Aplicar um Critério de Teste para Escolher Combinações de Valores**) antes de aplicar um critério de teste para escolher combinação de valores.

Então, no início do quarto passo, partindo dos mesmos parâmetros, características

e blocos descritos anteriormente, nesta abordagem analisam-se, primeiro, as restrições entre os blocos. Por exemplo, quando a lista é nula os valores dos demais parâmetros e características não importam, então não precisam ser combinados. Isso significa que só é preciso um teste em que **A** é verdadeiro, então, em todas as outras combinações **A** vai ser igual a falso - observe que apenas aqui o total de testes já foi reduzido praticamente pela metade. Assim, para o exemplo do método **findElement**, as seguintes restrições se aplicam:

- Se a lista é nula então ela não é vazia, o elemento não é nulo e não pertence à lista.
- Se a lista é vazia então o elemento não é nulo e não pertence à lista.
- Se o elemento é nulo então ele não pertence à lista.
- Se o elemento não pertence à lista então o número de ocorrências é zero.
- Se o número de ocorrências é zero então o elemento não é o primeiro nem o último.

Os testes escolhidos devem respeitar todas essas regras restritivas de forma simultânea, por isso é importante que elas sejam coerente e logicamente corretas. Por exemplo, se a segunda restrição determinasse que se a lista é vazia então ela também é nula, nesse caso estaria em contradição com a primeira restrição. Repare que se for selecionado lista vazia para um teste, resta escolher os blocos das características E, F e G. No caso esses blocos não importam (se a lista é nula gera exceção). Então é interessante identificar blocos que geram testes de sanidade e gerar apenas um teste com eles (lista nula e elemento nulo por exemplo). Isso em combinação com as restrições na forma como foram descritas garante testes de sanidade em que valores inválidos para **list** e **element** sejam testados de forma independente em testes de sanidade além de eliminar testes não factíveis.

Uma vez que todas restrições foram definidas para eliminar casos de teste desnecessários, então pode-se seguir no passo quatro, fazendo-se as combinações possíveis dentro do escopo do que é permitido. A partir daqui pode-se escolher algum critério de combinação *T-Wise* que for considerado mais adequado pelo testador. Depois podem ser escolhidos os valores de entrada e saída no passo cinco (**Refinar as Combinações de Blocos em Entradas de Teste.**).

A principal diferença nas duas abordagens está no momento em que o testador gasta esforço para eliminar testes não factíveis. Tendo conhecimento dessas duas abordagens, o testador pode ser capaz de discernir e decidir se a melhor opção para o sistema em teste é gerar as combinações e depois refinar os testes, ou limitar as possibilidades de combinações com base nas restrições gerando apenas teste factíveis.

2.2 Segundo Exemplo de Aplicação da Técnica: Exemplo da Soverteria

Este segundo exemplo foi extraído do tutorial da ferramenta **Tcases** e que será descrito a diante. Por meio dessa ferramenta será demonstrada a geração de testes com restrições um pouco mais complexas do que as do exemplo anterior.

O exemplo é a representação abstrata de uma sorveteria que serve cones de sorvete aos clientes. Os clientes podem escolher entre seis opções de sabores e entre seis opções de cobertura. É possível escolher mais de um sabor de sorvete (no máximo três), mas para cada sabor só é servida uma porção (uma bola). Podem ser escolhidas entre zero e três coberturas, mas cada tipo de cobertura é servida somente uma vez. A sorveteria classifica os possíveis tamanhos de cones de sorvete em cinco categorias (*Empty*, *Plain*, *Plenty*, *Grande* e *Too-Much*), de acordo com as escolhas dos clientes. O responsável pela preparação do pedido deve tomar o cuidado de não entregar um cone vazio (*Empty*), caso o cliente não escolha nenhum sabor. Esse responsável também deve evitar desperdícios e se negar a servir cones de sorvete com mais de três sabores ou com mais de três coberturas (*Too-Much*). A Tabela 2.5 mostra as condições para escolha dos tipos de cones válidos.¹

Tabela 2.5: Instruções para o atendente da soverteria.

Cones	Quantidade de Sabores	Quantidade de Coberturas
<i>Plain</i>	1	[0, 1]
<i>Plenty</i>	[1, 2]	[0, 2]
<i>Grande</i>	[1, 3]	[1, 3]

Fonte: O autor.

O atendente deve primeiro ter certeza de que a escolha da quantidade de sabores e coberturas é válida (ou seja, não é *Empty* nem *Too-Much*). Em seguida verifica se há condições para o cone ser *Plain*. Se não for o caso, verifica se há condições para o cone ser *Plenty*. Se novamente não for o caso, por último, verifica se há condições para o cone ser *Grande*.

Esse exemplo foi descrito da forma como foi fornecido e nela não há qualquer menção à técnica de particionamento dos domínios de entrada. No entanto, é possível discorrer sobre como a técnica poderia ter sido aplicada para modelar o sistema na forma como ele foi modelado.

Para o primeiro passo (**Identificar Funções Testáveis**) poderia-se considerar que a sorveteria do exemplo é um sistema testável com entradas e saídas que se comportam de acordo com a entrada.

¹Note que a Tabela 2.5 não define o caso em que é escolhido três sabores e zero coberturas. Essa condição será discutida na seção 4.2.

No segundo passo (**Encontrar Todos os Parâmetros**) os parâmetros de entrada desse exemplo seriam formados pelas escolhas dos sabores e das coberturas. Foi considerado no exemplo que cada sabor, assim como cada cobertura, pode ser escolhido ou não. Ou seja, seriam doze parâmetros booleanos. O exemplo não explora características desses parâmetros, então no passo de (**Definir o Domínio de Entrada**) poderia-se assumir que há dois blocos (um para verdadeiro e outro para falso) para cada um dos doze parâmetros.

A ferramenta na qual o exemplo foi fornecido modela o sistema de acordo com suas restrições. Como o exemplo foi fornecido para funcionar nessa ferramenta, poderia-se considerar que foi usada a abordagem em que se aplicam as restrições antes de realizar as combinações (passo quatro, **Aplicar um Critério de Teste para Escolher Combinações de Valores**). Poderia-se considerar que após aplicar as restrições se utiliza o critério de teste **T-Wise**. E para o passo cinco (**Refinar as Combinações de Blocos em Entradas de Teste**) a escolha dos valores de teste é ditada pelo bloco, que é verdadeiro ou falso, então os valores não precisariam ser escolhidos uma vez que o bloco já indica qual seu único valor possível.

Tanto para esse exemplo da sorveteria quanto para o do método **findElement** os testes podem ser gerados de forma manual. A técnica de Particionamento do Espaço de Entrada oferece um caminho sistemático para isso. Ao direcionar o testador a compreender claramente o espaço de entrada abre maior possibilidade de que os testes sejam relevantes e com maior potencial de evidenciar defeitos de *software*. Algumas partes desse processo podem ser automatizadas, no caso, a geração das combinações, que é um processo repetitivo, possivelmente demorado, e relativamente fácil de que erros sejam cometidos pelo testador. Ao automatizar esse processo, com ferramentas que gerem as combinações de forma rápida, possibilita-se que o testador tenha liberdade de experimentar e decidir qual o melhor critério para gerar combinações de características, avaliar os riscos envolvidos, e produzir testes que sejam mais adequados para *software* sob teste.

2.3 Trabalhos Relacionados

Kruse et al. (2013), Yao et al. (2017) e Eitner e Wotawa (2019) apresentam características de ferramentas relacionadas a Testes Combinatórios que podem servir como critério de comparação neste trabalho de conclusão de curso. Khalsa (2017) analisa 75 algoritmos e ferramentas para a geração de uma suíte de testes baseada na técnica de Teste Combinatório. O estudo categorizou esses algoritmos e ferramentas de acordo com ca-

racterísticas específicas da técnica de Teste Combinatório como os algoritmos para gerar as combinações, força das combinações e suporte às restrições de combinações.

Os trabalhos estudados como referência apresentam as vantagens do uso de Teste Combinatório em relação a outros métodos, mostram resultados de aplicações do método em casos reais e contêm características relevantes para se tomar como critério de comparação entre ferramentas. Khalsa (2017) fez um estudo mais abrangente, em relação à quantidade de ferramentas analisadas, e acaba facilitando a busca por ferramentas (no entanto o estudo de Khalsa (2017) já faz quatro anos desde a sua publicação, possivelmente já surgiram novas ferramentas que podem ser analisadas também).

3 PROPOSTA

O objetivo principal deste trabalho é analisar comparativamente ferramentas para o desenvolvimento de testes, baseados na técnica de **Particionamento do Espaço de Entrada**. Para isso, seguiu-se a seguinte metodologia: foi feita uma busca ampla por esse tipo de ferramenta; em seguida foi feita a definição de critérios preliminares; com esses critérios foi feita a seleção das ferramentas para análise detalhada; definiu-se critérios para a análise comparativa; por último é feita a análise comparativa a partir de dois estudos de caso.

3.1 Levantamento e Análise Preliminar

Foram buscadas ferramentas gratuitas e disponíveis na *internet* visando descobrir as opções existentes e a partir disso fechar o foco nas que sejam mais relevantes para o estudo. Algumas foram descobertas nos materiais de referência bibliográfica enquanto outras foram descobertas em buscas pelo tema na internet¹. Muitas das ferramentas citadas em publicações não foram encontradas. Possivelmente algumas delas foram criadas com propósitos acadêmicos apenas e já não se encontram mais disponíveis na internet, ou são de difícil acesso (não foram encontradas em resultados de sites de buscas, como o *Google*). Outras apenas foram descontinuadas e nem suas páginas para download estão disponíveis mais. A Tabela 3.1 resume as 22 ferramentas que foram encontradas e puderam ser consideradas para este trabalho.²

Os critérios levados em consideração para a seleção preliminar são descritos abaixo.

1. **Critério Preliminar: Documentação.** Foi considerada importante a existência de documentação para o entendimento das características e de como utilizar cada ferramenta. A grande maioria delas apresentam alguma forma de documentação, indo desde um simples arquivo de texto na pasta compactada do projeto até um site com explicações detalhadas sobre a ferramenta com tutoriais didáticos de uso com exemplos. Espera-se que exista o mínimo de informação documentada para entender e operar suas funcionalidades. Arquivos executáveis sem qualquer informação associada são considerados ferramentas sem documentação neste trabalho, a não

¹No site de busca utilizado foram usadas as palavras chave "ferramenta para teste combinatório", "*combinatorial testing tool*" e "*input space modeling tool*".

²Sites acessados entre março e junho de 2021.

Tabela 3.1: Ferramentas gratuitas e sites onde elas podem ser encontradas.

Ferramenta	Site
ACTS	https://tinyurl.com/26psf32p
AllPairs	https://sourceforge.net/projects/allpairs/
AllPairs by Satisfice	https://www.satisfice.com/download/allpairs
AllPairsPy	https://github.com/thombashi/allpairspy/
CAgen	https://matris.sba-research.org/tools/cagen/#/workspaces
CAMetrics	https://matris.sba-research.org/tools/cametrics/#/new
CoverTable	https://github.com/walkframe/covertable
CTWedge	https://foselab.unibg.it/ctwedge/
JCUnit	https://github.com/dakusui/jcunit
Jenny	http://burtleburtle.net/bob/math/jenny.html#tools
NTestCaseBuilder	https://www.nuget.org/packages/NTestCaseBuilder/
NUnit	https://docs.nunit.org/2.6.4/pairwise.html
Pairwise Online Tool	https://pairwise.teremokgames.com/4s8/
Pairwise Pict Online	https://pairwise.yuuniworks.com/
PICT	https://github.com/Microsoft/pict/blob/main/doc/pict.md
PictMaster	https://osdn.net/projects/pictmaster/
SpecExplorer	https://tinyurl.com/cj74765z
SQA Mate Tools: Pairwise	https://sqamate.com/tools/pairwise
Tcases	https://github.com/cornutum/tcases
Test Vector Generator	https://sourceforge.net/projects/tvg/
tslgenerator	https://github.com/alexorso/tslgenerator
VPTAG	https://sourceforge.net/projects/vptag/

Fonte: O autor.

ser que a ferramenta contenha um sistema de *help* integrado.

2. **Critério Preliminar: Obsolescência.** Quanto à obsolescência da ferramenta, isso foi levado em conta pois ferramentas mais antigas podem apresentar defasagem no emprego de algoritmos, técnicas e métodos de testes combinatórios. Esses métodos vêm sendo estudados e evoluídos até o presente momento. Por isso foi dada preferência por ferramentas mais recentes. No escopo deste trabalho foi estabelecido que as ferramentas consideradas recentes e não obsoletas são aquelas criadas nos últimos cinco anos, ou receberam atualizações em seus projetos nos últimos cinco anos.
3. **Critério Preliminar: Oferta de Combinações *T-Wise*** Considera-se importante que as ferramentas de teste combinatórios possibilitem ao desenvolvedor dos testes escolher a força das combinações. O desenvolvedor é quem deve avaliar e decidir se os testes gerador são adequados para o seu sistema. Ele é quem deve decidir se não são necessários tantos testes ou se é preciso gerar mais testes. Esse último caso pode ser necessário em sistemas críticos a fim de aumentar ao máximo a cobertura

do espaço de entrada e a cobertura estrutural do código para diminuir ao mínimo possível a probabilidade de não detecção de defeitos de software, mesmo que custe mais caro em termos de quantidade de testes gerados. Cabe então ao desenvolvedor decidir a força da combinação. Portanto, as ferramentas estudadas devem oferecer a opção de gerar combinações *T-Wise*, e não apenas *Pair-Wise*.

3.2 Ferramentas Seleccionadas para Estudo Após a Análise Preliminar

A Tabela 3.2 apresenta o resultado da análise preliminar das ferramentas encontradas, com base nos critérios de seleção definidos. Pode-se observar que a maioria fornece algum tipo de documentação. Mais da metade não é obsoleta enquanto que menos da metade oferece combinações *T-Wise*. Apenas quatro das ferramentas citadas anteriormente atendem a todos os critérios preliminares de forma simultânea, são elas: **ACTS**, **CAgen**, **CTWedge** e **Tcases**.

Tabela 3.2: Conformidade com os critérios preliminares. As ferramentas que estão de acordo com os critérios receberam S (sim) enquanto as que não estão de acordo estão marcadas com N (não).

Ferramenta	Documentação	Obsoleta	<i>T-Wiser</i>
ACTS	S	N	S
AllPairs	S	S	N
AllPairs by Satisfice	S	S	N
AllPairsPy	S	S	N
CAgen	S	N	S
CAMetrics	S	N	N
CoverTable	S	N	N
CTWedge	S	N	S
JCUnit	S	N	N
Jenny	S	S	S
NTestCaseBuilder	S	S	S
NUnit	S	N	N
Pairwise Online Tool	N	N	N
Pairwise Pict Online	S	N	N
PICT	S	N	N
PictMaster	S	S	N
SpecExplorer	S	S	S
SQA Mate Tools: Pairwise	S	N	N
Tcases	S	N	S
Test Vector Generator	S	S	S
tslgenerator	S	N	N
VPTAG	S	S	N

Fonte: O autor.

4 AVALIAÇÃO EXPERIMENTAL

Neste capítulo será apresentada a análise comparativa feita com as ferramentas **ACTS**, **CAgen** e **Tcases**. Note que **CTWedge** não foi citada. Apesar de ela satisfazer os critérios preliminares, não foi capaz de gerar resultados por algum problema técnico da ferramenta. Então foi decidido não dar continuidade em sua análise após alguns testes básicos nos quais ela, aparentemente, não foi capaz de gerar as combinações. Enquanto as outras três ferramentas geraram as combinações em milésimos de segundo, **CTWedge** não retornou as combinações mesmo depois de uma hora de processamento.

A ferramenta **ACTS** (*Automated Combinatorial Testing for Software*) foi desenvolvida e ainda é mantida pelo CSRC¹ (*Computer Security Resource Center*) pertencente ao **NIST** (*National Institute of Standards and Technology*), nos Estados Unidos. Ela possui usuários em organizações em todo o mundo, como Adobe, IBM, Jaguar Land Rover, Lockheed Martin, Red Hat, Rockwell Collins, Siemens, US Air Force, US Marine Corps, entre outras. **CAgen** foi desenvolvida e é mantida pelo *SBA Research Group*², na Áustria. Sobre a ferramenta *Tcases*, ela é desenvolvida e mantida pelo grupo CORNUTUM³ (a página *web* do grupo não fornece informações sobre o grupo em si).

Para analisá-las, primeiramente foram definidos critérios de comparação que são quatro: usabilidade; suporte às restrições; extensão de conjunto de teste; e algoritmos suportados. A análise em si foi feita através da geração de testes para os dois exemplos citados na fundamentação teórica deste documento usando cada uma das três ferramentas.

4.1 Definição de Critérios Comparativos

Esta seção define critérios de comparação entre as ferramentas selecionadas para análise.

1. **Critério Comparativo: Usabilidade.** Este critério aponta a forma que o testador interage com a ferramenta. As seguintes perguntas podem ser correlacionadas a esse critério: Qual o tipo de *user interface* da ferramenta? Como o testador informa os parâmetros para a ferramenta? Como o testador informa as restrições para a ferramenta? Como a ferramenta apresenta os resultados? Quais as estatísticas que

¹Site do **CSRC**: <https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software>, último acesso em julho de 2021

²Site do **SBA Research Group**: <https://www.sba-research.org/>, último acesso em julho de 2021

³Site do **cornutum**: <http://www.cornutum.org/>, último acesso em julho de 2021

a ferramenta apresenta?

2. **Critério Comparativo: Suporte às Restrições.** Este critério indica se um ferramenta é capaz de fazer uso de restrições de combinações configuradas pelo testador. Essas restrições podem ser descritas por meio dos seguintes tipos de operação:
 - Booleanas: Operações relacionadas à álgebra booleana.
 - Relacionais: Operações para avaliar entre duas variáveis se elas são iguais entre si, diferentes, maior ou menor.
 - Algébricas: Operadores algébricos podem permitir operações algébricas básicas como soma, subtração, multiplicação e divisão de variáveis na descrição das restrições.
3. **Critério Comparativo: Extensão de Conjunto de Teste.** Extensão de conjunto de teste refere-se à capacidade da ferramenta de receber não apenas parâmetros para combinação, mas também combinações já existentes. Essa capacidade permite uma suíte de teste já existente ser aprimorada, estendendo-se seu conjunto de teste para aumentar a cobertura do espaço de entrada.
4. **Critério Comparativo: Algoritmos Suportados.** Esse critério preocupa-se em entender se a ferramenta é capaz de realizar combinações não apenas *T-Wise*, mas também por meio de **Basic Choice** ou algum outro critério para combinação.

4.2 Geração de Testes com as Ferramentas para o Método findElement

ACTS e **CAgen** são semelhantes na forma em que os parâmetros e as restrições são inseridos no sistema. Em ambas é possível inserir cada parâmetro ao se apertar o respectivo botão para isso e informar o tipo de dados que pode ser booleano, enumeração entre outros. As restrições podem ser inseridas uma a uma ou carregadas de um arquivo de texto que, inclusive, possuem uma sintaxe semelhante. Na Figura 4.1 podem-se observar as restrições criadas para o primeiro exemplo no **ACTS**. Com a ferramenta **CAgen** a descrição das restrições foi exatamente igual.

Na **Tcases** a inserção de parâmetros e restrições é diferente. Nesta é feita uma descrição do modelo como um todo, em formato **xml**. Esse modelo permite descrever de forma estruturada os parâmetros e a forma como eles interagem. Diferentemente das outras duas ferramentas, as restrições no **Tcases** fazem parte do modelo do sistema. Na Figura 4.2 é possível notar que é feita a descrição das características diretamente relaci-

onadas aos parâmetros de entrada a partir da linha 4 até a linha 19. A partir da linha 21 o sistema é modelado funcionalmente. Observe que às restrições são inerentes a descrição do modelo do sistema uma vez que os casos de testes serão gerados de acordo com o modelo funcional. A partir desse modelo a ferramenta produz apenas os casos de teste que são relevantes para o sistema descrito.

Figura 4.1: Restrições criadas no ACTS.

1	(A_listIsNull = true) => B_listIsEmpty = false
2	(A_listIsNull = true) => C_elementIsNull = false
3	(A_listIsNull = true) => D_elementBelowsToaList = false
4	(B_listIsEmpty = true) => C_elementIsNull = false
5	(B_listIsEmpty = true) => D_elementBelowsToaList = false
6	(C_elementIsNull = true) => D_elementBelowsToaList = false
7	(D_elementBelowsToaList = false) => E_numOfOccur = "0"
8	(E_numOfOccur = "0") => F_elementIsFirst = false
9	(E_numOfOccur = "0") => G_elementIsLast = false

Fonte: O autor.

Figura 4.2: Modelagem do exemplo **findElement** descrito no Tcases.

```

1 <System name="exemplo-de-aula">
2   <Function name="findElement">
3     <Input>
4       <VarSet name="list">
5         <Var name="null">
6           <Value name="F" property="listIsValid"/>
7           <Value name="T" failure="true"/>
8         </Var>
9         <Var name="empty">
10          <Value name="T" failure="true"/>
11          <Value name="F" property="listIsNotEmpty"/>
12        </Var>
13      </VarSet>
14      <VarSet name="element">
15        <Var name="null">
16          <Value name="T" failure="true"/>
17          <Value name="F" property="elementIsValid"/>
18        </Var>
19      </VarSet>
20
21      <VarSet name="operation" when="listIsValid, listIsNotEmpty, elementIsValid">
22        <Var name="elementBelongsToaList">
23          <Value name="T" property="belong"/>
24          <Value name="F" failure="true"/>
25        </Var>
26        <VarSet name="occurr" when="belong" >
27          <Var name="numberOfOccurr">
28            <Value name="0" failure="true" />
29            <Value name="1" property="atLeastOneOccurr" />
30            <Value name="+_than_1" property="atLeastOneOccurr" />
31          </Var>
32          <VarSet name="position" when="atLeastOneOccurr">
33            <Var name="first">
34              <Value name="T" />
35              <Value name="F" />
36            </Var>
37            <Var name="last">
38              <Value name="T" />
39              <Value name="F" />
40            </Var>
41          </VarSet>
42        </VarSet>
43      </VarSet>
44    </Input>
45  </Function>
46 </System>

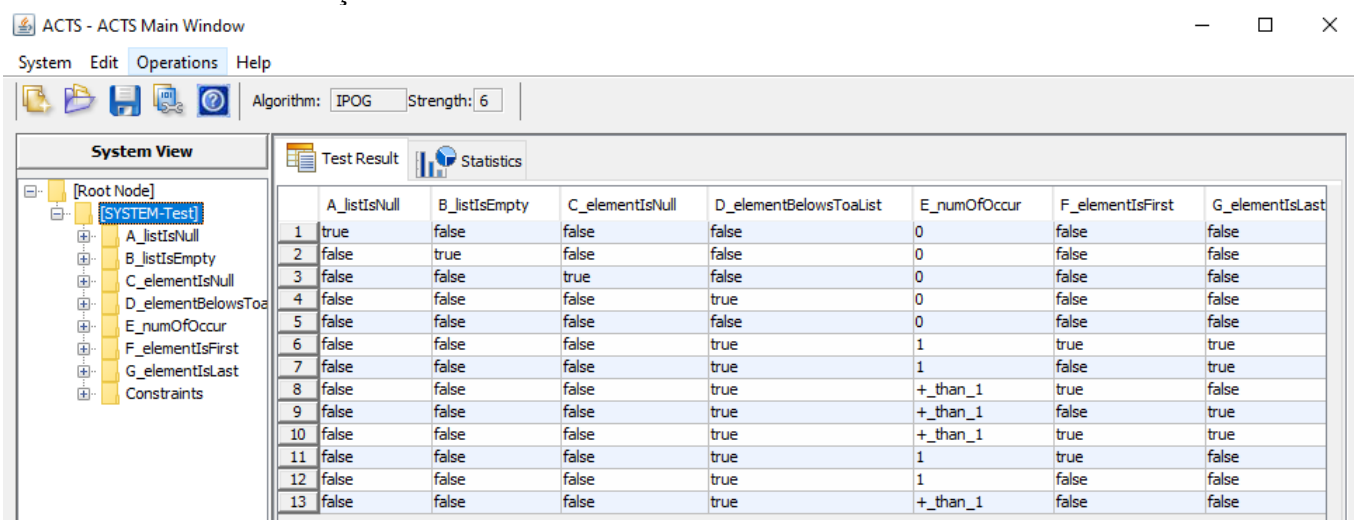
```

Fonte: O autor.

Com o uso das restrições, pôde-se usar as ferramentas **ACTS** e **CAGen** para gerar casos de teste conforme mostrado na Figura 4.3 para o **ACTS**. Casos de sanidade (**T1**, **T2**, **T3** e **T7** nos quais estão presentes os blocos **A1**, **B1**, **C1** e **D2**, respectivamente) são

testados apenas uma vez cada. Demais casos relevantes foram gerados realizando todas as combinações possíveis restantes. Apesar das diferenças em relação às outras duas, com a ferramenta **Tcases** foi gerada exatamente a mesma quantidade de testes com as mesmas combinações de parâmetros.

Figura 4.3: Testes para o método **findElement** gerados com a ACTS usando restrições e todas as combinações.



	A_listIsNull	B_listIsEmpty	C_elementIsNull	D_elementBelongsToList	E_numOfOccur	F_elementIsFirst	G_elementIsLast
1	true	false	false	false	0	false	false
2	false	true	false	false	0	false	false
3	false	false	true	false	0	false	false
4	false	false	false	true	0	false	false
5	false	false	false	false	0	false	false
6	false	false	false	true	1	true	true
7	false	false	false	true	1	false	true
8	false	false	false	true	+_than_1	true	false
9	false	false	false	true	+_than_1	false	true
10	false	false	false	true	+_than_1	true	true
11	false	false	false	true	1	true	false
12	false	false	false	true	1	false	false
13	false	false	false	true	+_than_1	false	false

Fonte: O autor.

Somente a ferramenta **ACTS** permite a geração dos testes com o algoritmo *Basic Choice*, como pode ser observado na Figura 4.4. Conforme previsto na descrição do primeiro exemplo no Capítulo 2, o esperado é que sejam gerados nove testes para este algoritmo e a ferramenta gerou exatamente nove testes. No entanto, observa-se que os testes diferem dos testes selecionados na Tabela 2.4. Isso ocorre devido ao fato de que a ferramenta não é capaz de gerar testes com o algoritmo *Basic Choice* na presença de restrições. Enquanto a Tabela 2.4 foi construída levando em consideração cenários que fazem sentido, alguns testes gerados pela ACTS acabam não fazendo sentido. É o caso do teste **T5** em que, caso o elemento não pertença a lista (**D2**), o número de ocorrências só pode ser igual a zero (**E1**). O ponto negativo dessa deficiência da ferramenta é que possíveis casos de teste mais interessantes podem acabar não sendo gerados pois a ferramenta não é capaz de saber quais casos são relevantes ou não.

4.3 Geração de Testes com as Ferramentas para o Exemplo da Sorveteria

Este exemplo está disponível em (CORNUTUM, 2021) na seção *Cardinality conditions*. Ele também faz parte da documentação disponível junto com o *download* da fer-

Figura 4.4: Testes para o método **findElement** gerados com a ACTS usando *Basic Choice*.

	A_listIsNull	B_listIsEmpty	C_elementIsNull	D_elementBelowsToaList	E_numOfOccur	F_elementIsFirst	G_elementIsLast
1	false	false	false	true	1	true	false
2	true	false	false	true	1	true	false
3	false	true	false	true	1	true	false
4	false	false	true	true	1	true	false
5	false	false	false	false	1	true	false
6	false	false	false	true	0	true	false
7	false	false	false	true	+_than_1	true	false
8	false	false	false	true	1	false	false
9	false	false	false	true	1	true	true

Fonte: O autor.

ramenta *Tcases* na forma de modelo de um sistema que pode ser lido pela ferramenta para gerar os casos de teste. A modelagem do exemplo utiliza-se de "Condições de Cardinalidade" que, no contexto do *Tcases*, permite que seja levada em conta a quantidade de vezes que alguma característica acontece. Por exemplo, a quantidade de sabores e coberturas escolhidas. O modelo descreve diferentes comportamentos para diferentes ocorrências de determinadas características.

Para que as outras ferramentas consigam gerar os testes baseados em uma modelagem semelhante é necessário que a ferramenta permita a descrição de regras de restrições não apenas booleanas e relacionais, mas também algébricas. A ferramenta **CAGen** mostrou-se incapaz de prover esse tipo de regra de restrição. Com ela não foi possível gerar testes para o exemplo. **Tcases** oferece esse tipo de regra, conforme demonstrado na Figura 4.5, e foi capaz de gerar testes. Para isso, cada sabor e cobertura foi definido como uma enumeração com valores entre zero e um. Considera-se que, para cada cobertura e sabor, o valor "1" indica que o cliente o escolheu.

Observa-se na Figura 4.5 que, para o **ACTS**, a descrição das restrições é composta por diversas linhas com relações lógicas entre as características do modelo, enquanto que na *Tcases* há a visão do sistema como um todo, de forma estruturada e organizada, semelhante às linguagens de programação com controle de fluxo lógico. Supondo-se que seja feita a expansão desse exemplo com a adição de mais variáveis e mais comportamentos específicos para testar, a *Tcases* aparenta ser mais vantajosa nesse caso pois permite mais organização na descrição do modelo (incluindo restrições que fazem parte do modelo) ao invés de necessitar dezenas, talvez centenas, de regras cada vez maiores e que são apenas declaradas sequencialmente.

Os casos de teste gerados pela **ACTS** podem ser vistos na Figura 4.6. Não foi

Figura 4.5: Restrições criadas para a ACTS para o método `findElement`.

```

1 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe < 1) => (Cone = "Empty")
2 (Cone = "Empty") => Sprinckles = 0
3 (Cone = "Empty") => Pecans = 0
4 (Cone = "Empty") => Oreos = 0
5 (Cone = "Empty") => Cherries = 0
6 (Cone = "Empty") => MMs = 0
7 (Cone = "Empty") => Peppermint = 0
8 (Cone = "Empty") => Vanilla = 0
9 (Cone = "Empty") => Chocolate = 0
10 (Cone = "Empty") => Strawberry = 0
11 (Cone = "Empty") => Pistachio = 0
12 (Cone = "Empty") => Lemon = 0
13 (Cone = "Empty") => Coffe = 0
14
15 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 1) && (Sprinckles + Pecans + Oreos +
16 Cherries + MMs + Peppermint <= 1) => (Cone = "Plain")
17
18 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 1) && (Sprinckles + Pecans + Oreos +
19 Cherries + MMs + Peppermint = 2) => (Cone = "Plenty")
20 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 2) && (Sprinckles + Pecans + Oreos +
21 Cherries + MMs + Peppermint <= 2) => (Cone = "Plenty")
22
23 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 1) && (Sprinckles + Pecans + Oreos +
24 Cherries + MMs + Peppermint = 3) => (Cone = "Grande")
25 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 2) && (Sprinckles + Pecans + Oreos +
26 Cherries + MMs + Peppermint = 3) => (Cone = "Grande")
27 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 3) && ((Sprinckles + Pecans + Oreos +
28 Cherries + MMs + Peppermint >=1) && (Sprinckles + Pecans + Oreos + Cherries + MMs + Peppermint <= 3)) =>
29 (Cone = "Grande")
30
31 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe > 3 ) => (Cone = "Too-Much")
32 (Cone = "Too-Much") => Vanilla = 0
33 (Cone = "Too-Much") => Chocolate = 0
34 (Cone = "Too-Much") => Strawberry = 1
35 (Cone = "Too-Much") => Pistachio = 1
36 (Cone = "Too-Much") => Lemon = 1
37 (Cone = "Too-Much") => Coffe = 1
38
39 (Sprinckles + Pecans + Oreos + Cherries + MMs + Peppermint >= 4) => (Cone = "Too-Much")
40 (Cone = "Too-Much") => Sprinckles = 0
41 (Cone = "Too-Much") => Pecans = 1
42 (Cone = "Too-Much") => Oreos = 1
43 (Cone = "Too-Much") => Cherries = 1
44 (Cone = "Too-Much") => MMs = 1
45 (Cone = "Too-Much") => Peppermint = 1
46
47 (Vanilla + Chocolate + Strawberry + Pistachio + Lemon + Coffe = 3) => !(Sprinckles + Pecans + Oreos +
48 Cherries + MMs + Peppermint = 0)

```

Fonte: O autor.

possível chegar no mesmo conjunto de testes gerado pela **Tcases**. Uma possível causa seria a de que o conjunto de regras criadas na **ACTS** não foram equivalentes às regras implícitas no modelo da sorveteria na **Tcases**.

Tentou-se modificar pontualmente a modelagem para que a **ACTS** apresentasse os mesmos resultados que **Tcases**. Por isso, foram adicionadas duas novas características para combinar: a quantidade de sabores e a quantidade de coberturas. Mesmo assim não foi atingido exatamente o mesmo resultado. Entretanto, foi possível observar o seguinte: a ferramenta **Tcases**, em suas combinações, gerou testes levando em conta a quantidade de sabores e coberturas mas não explorou quantidades diferentes de sabores de forma simultânea com quantidades diferentes de cobertura. Por exemplo, testou apenas casos

Figura 4.6: Testes gerados para o exemplo da sorveteria com ACTS.

	Cone	Vanilla	Chocolate	Strawberry	Pistachio	Lemon	Coffe	Sprinkles	Pecans	Oreos	Cherries	MMs	Peppermint
1	Plain	0	1	0	0	0	0	1	0	0	0	0	0
2	Plain	1	0	0	0	0	0	0	1	0	0	0	0
3	Plenty	0	0	1	1	0	0	1	1	0	0	0	0
4	Plenty	1	1	0	0	0	0	0	0	1	1	0	0
5	Grande	0	0	1	0	1	1	0	0	1	1	1	0
6	Grande	1	1	1	0	0	0	1	1	1	0	0	0
7	Grande	1	1	0	1	0	0	0	0	0	1	1	1
8	Plain	0	0	1	0	0	0	0	0	1	0	0	0
9	Plain	0	0	0	1	0	0	0	0	1	0	0	0
10	Grande	0	1	0	1	1	0	1	1	0	1	0	0
11	Plenty	1	0	0	0	1	0	0	0	0	0	1	1
12	Plain	0	0	0	0	1	0	0	0	0	1	0	0
13	Grande	0	1	0	1	0	1	1	1	0	0	1	0
14	Plenty	1	0	0	0	0	1	0	0	1	0	0	1
15	Plain	0	0	0	0	0	1	0	0	0	0	1	0
16	Grande	0	0	1	1	1	0	1	1	0	0	0	1
17	Plain	1	0	0	0	0	0	0	0	0	0	0	1
18	Empty	0	0	0	0	0	0	0	0	0	0	0	0
19	Too-Much	0	0	1	1	1	1	0	1	1	1	1	1

Fonte: O autor.

com um sabor e uma cobertura, dois sabores e duas coberturas, etc. **ACTS**, com uma pequena modificação na modelagem do domínio, testou casos por exemplo, em que a quantidade de sabores é três e a de coberturas é zero.

A partir da Tabela 2.5 podemos entender que o caso de dois sabores e zero coberturas é interessante pois o comportamento para esse caso em específico não está definido. Essa falta de definição pode ser proposital, mesmo que a justificava não faça sentido no contexto do exemplo, ou pode ser acidental.

Supondo-se que a não definição do comportamento para dois sabores e zero coberturas seja proposital, o modelo do sistema provido à ferramenta **Tcases** também não modela o comportamento para esse caso, então não gera teste para isso. O testador, fazendo uso da **Tcases**, pode acabar nem percebendo que esse é um caso não definido pelo sistema, mas não seria problema uma vez que esse caso não importa. Por outro lado, supondo que essa falta de definição seja uma deficiência na implementação do sistema, o testador usando **Tcases** não se daria conta desse caso. O conjunto de testes gerado, mesmo que todos eles passem durante a execução, não seria capaz de encontrar um defeito no software relacionado a esse comportamento.

Se no exemplo da sorveteria tivesse sido aplicada a técnica de Particionamento do Espaço de Entrada, da mesma forma que foi aplicada no exemplo do método **findElement**, explorando as características dos parâmetros, possivelmente, poderiam ter sido incluídas características e blocos para quantidades (de sabores e de coberturas). Com a combinação entre essas características seria possível notar essa falta de definição do sis-

tema. Nesse caso o testador teria que interagir com os desenvolvedores do sistema que indicariam a ele o comportamento esperado. Se esse comportamento é importante, a suíte de testes deve testar.

4.4 Observações Sobre as Ferramentas a Respeito dos Critérios Comparativos

Sobre o critério de **Usabilidade**, **ACTS** e **CAgen** permitem entrar com parâmetros e restrições tanto via interface gráfica quanto via linhas de comando enquanto que **Tcases** não possui interface gráfica. Nas que possuem interface, os parâmetros e restrições podem ser introduzidos e configurados um a um de forma interativa. Para as três ferramentas é possível fornecer essas informações via arquivos de texto. Sistemas completos com parâmetros, blocos e restrições podem ser lidos de arquivos de texto os quais podem ser escritos manualmente ou produzidos via interface gráfica. Somente **ACTS** e **CAgen** permitem o recurso chamado *negative testing* ou *failure* que seria a possibilidade de distinguir parâmetros que não precisam ser combinados exaustivamente, como parâmetros inválidos em que há a necessidade de usá-los apenas em um teste.

ACTS e **CAgen** apresentam a visualização de uma tabela com os casos de teste gerados. **ACTS** ainda permite analisar o conjunto de testes e apresentar estatísticas referente à cobertura do espaço de entrada. Essas duas ferramentas permitem exportar os casos de teste em 3 formatos; **nist**, **csv** e **excel**. **Tcases** apresenta os resultados no console ou em uma página *web*. Um diferencial no critério de usabilidade da **Tcases** é a possibilidade de gerar *scripts* de teste em *Java*, mas pode ser possível configurar a ferramenta para gerar código em qualquer linguagem.

No critério **Suporte às Restrições**, conforme visto na aplicação dos exemplos, as três ferramentas possibilitam a criação de restrições baseadas em operações booleanas e relacionais. Somente **CAgen** não suporta operações algébricas.

Sobre os **Algoritmos Suportados**, de forma geral, as três ferramentas possibilitam - com e sem restrições - fazer combinações *1-wise*, *2-wise (pairwise)*, até *6-wise*. Somente **ACTS** possibilita gerar testes com o critério **Basic Choice**, mas sem o uso de restrições.

Quanto ao critério **Extensão de Conjunto de Teste**, as três ferramentas permitem esta funcionalidade (**CAgen** somente via linhas de comando). Elas permitem que um arquivo possa ser importado contendo descrição de casos de teste. Esse arquivo deve descrever os blocos para cada teste e pode ser escrito manualmente ou gerado pelas ferramentas. Ao se importar uma suíte de teste existente, deve-se solicitar às ferramentas

que gerem testes a partir dos que foram importados. Testes que não respeitam as restrições são cortados (removidos da suíte de testes). As ferramentas adicionam novos testes caso seja necessário aumentar a cobertura do espaço de entrada. A Tabela 4.1 resume essas observações dos critérios.

Tabela 4.1: Resumo das características das ferramentas em relação aos critérios comparativos.

Ferramenta	Usabilidade	Suporte às Restrições	Extensão de Conjunto de Testes	Algoritmos Suportados
ACTS	Interface gráfica (<i>desktop</i>) e linhas de comando.	Suporta	Suporta	<i>T-Wise, Basic Choice</i>
CAgen	Interface gráfica (<i>web</i>) e linhas de comando.	Suporta, porém sem operadores algébricos.	Suporta	<i>T-Wise</i>
Tcases	Linhas de comando.	Suporta	Suporta	<i>T-Wise</i>

Fonte: O autor.

Foi possível observar que a modelagem usada na ferramenta **Tcases** buscou representar perfeitamente o sistema mas pode não ter permitido identificar (mascarou) um possível defeito na definição do sistema do segundo exemplo que pode vir a causar alguma falha. Fazer a modelagem do domínio do espaço de entrada de forma sistemática como no exemplo do método **findElement** em conjunto com a ferramenta **ACTS** (talvez até com a **CAgen** e a própria **Tcases**, dependendo de como for feita o modelagem) tem potencial para gerar casos de teste mais relevantes para o sistema. Considerando que casos de teste interessantes sejam aqueles com maior potencial de encontrar defeitos, **ACTS** destacou-se por ter sido capaz de apresentar melhores resultados em conjunto com a técnica de particionamento do espaço de entrada.

As ferramentas **ACTS** e **CAgen** são relativamente mais simples de usar, se comparado a **Tcases**, pois o testador não precisa descrever o sistema em uma linguagem específica, basta inserir as características via interface gráfica. Outra vantagem dessa simplicidade é que se o testador seguir de forma sistemática os passos da técnica de Particionamento do Espaço de Entrada e encontrar características interessantes para o teste, ele poderá explorar de forma mais ágil as possibilidades de combinações entre as características usando **ACTS** e **CAgen**. Entre essas duas, **ACTS** destaca-se por suportar restrições de forma mais completa e oferecer a possibilidade de uma forma a mais de combinação (*Basic Choice*).

ACTS, *CAgen* e *Tcases* apresentam diferenças de usabilidade e na forma de modelar o sistema (descrever o espaço de entrada e as restrições), mas as três possibilitam de forma gratuita o auxílio na geração de testes baseada no particionamento do espaço de entrada. Em cenários simples, com poucas variáveis em sistemas de baixa complexidade as ferramentas chegaram a resultados muito semelhantes. Isso passa segurança a quem pretende adotá-las nos ciclo de vida do *software* em seu ambiente de desenvolvimento. No entanto, em casos mais complexos as ferramentas podem não apresentarem exatamente os mesmos resultados, mas isso não significa que elas estejam erradas. Isso reforça apenas a importância no entendimento teórico e no domínio da técnica de particionamento além de conhecer o sistema sob teste.

Independente da ferramenta, o testador sempre vai ter que avaliar não apenas se os testes fazem sentido e são suficientes, mas também precisa garantir que o particionamento foi feito de forma correta pois o resultado final é extremamente dependente disso. Em outras palavras, independente das características das ferramentas em relação aos critérios técnicos comparativos, o mais importante é que as ferramentas gerem testes com qualidade e pra que isso aconteça o testador deve usar a técnica corretamente. O que garante que os testes sejam gerados com qualidade é a etapa manual da técnica, que resulta no particionamento e na escolha do critérios combinatórios, e não a combinação automática dos blocos das partições.

5 CONCLUSÃO

A técnica de Particionamento do Espaço de Entrada pode oferecer às empresas e instituições desenvolvedoras de *software* uma metodologia para o processo de desenvolver testes. Com ela, profissionais são direcionados a investir mais tempo na análise das características do *software*, o que pode levar a combinações de características mais relevantes se comparado à uma metodologia (ou ausência de metodologia) que dependa apenas da experiência do testador. Parte do processo de aplicar essa técnica pode ser automatizado com ferramentas que geram as combinações de características, o que possivelmente diminui erros humanos e aumenta a agilidade. Existem muitas ferramentas que se propõem a realizar essa tarefa e por isso, pessoas e companhias interessadas nos benefícios dessa técnica, podem não saber qual seria a mais adequada para começar a incorporar essa técnica na cultura de suas empresas. Então, com a intenção de auxiliar na propagação da técnica, este trabalho realizou uma análise comparativa entre as opções (gratuitas) de ferramentas disponíveis atualmente.

Este trabalho identificou 22 ferramentas gratuitas, algumas citadas em publicações relacionadas a testes combinatórios e outras encontradas em sites de buscas na *internet*. O total de 19 delas foram consideradas obsoletas, ou sem documentação mínima ou limitada ao critério de teste *Pair-Wise*. Apenas 4 delas foram consideradas relevantes para este estudo. Entretanto, **CTWedge** não funcionou corretamente então, de fato foram analisadas apenas as ferramentas **ACTS**, **CAgen** e **Tcases**.

A aplicação correta da técnica é o fator mais importante e determinante para que qualquer uma das ferramentas seja capaz de gerar testes com qualidade. As três ferramentas mostraram-se capazes de gerar bons testes quando a técnica foi bem utilizada. E entre elas, a ferramenta **ACTS** destacou-se por ser mais completa em relação às funcionalidades oferecidas. Tanto a **ACTS** quanto a **CAgen** destacam-se em relação a **Tcases** por se encaixar de forma mais clara e eficiente na automatização da geração dos casos de teste seguindo a técnica de Particionamento do Espaço de Entrada.

Para fechar o escopo deste trabalho limitou-se a busca por ferramentas somente às gratuitas pela facilidade de utilizá-las sem custos financeiros. Algumas ferramentas pagas possivelmente poderiam ser estudadas em suas versões de avaliação, mas entendeu-se que isso poderia atrapalhar o andamento desse trabalho uma vez que versões de avaliações podem ter recursos limitados. Considerando o tempo disponível para a realização deste trabalho, limitou-se também a analisar nas ferramentas apenas os critérios **Basic Choice**

e **Pair-Wise** (não foram levadas em consideração a influência do aumento de T em combinações *T-Wise*). Outra limitação deste trabalho é que os estudos de casos foram apenas em aplicações fictícias.

As três ferramentas possibilitaram realizar combinações *6-Wise* (*T-Wise* com T igual a 6) que pode ser interessante para sistemas críticos. Em uma aplicação real pode-se começar usando *Pair-Wise* para fazer as combinações, que tem potencial para garantir uma boa cobertura do espaço de entrada. Se os projetistas dos testes perceberem que é necessário aumentar a quantidade de testes para melhorar a cobertura estrutural do código pela execução da suíte de teste, é possível utilizar as ferramentas para aumentar a força das combinações, de forma incremental inclusive. Uma possibilidade de trabalho futuro seria a avaliação do impacto na análise estrutural de código para sistemas cujas suítes de teste foram desenvolvidas com diferentes forças de combinação.

Notou-se neste trabalho que as ferramentas possibilitaram critérios para escolha das combinações dos testes de forma combinacional (*T-Wise*) principalmente. Uma delas possibilitou *Basic Choice*, mas de forma limitada. Existe então, como possibilidade de trabalho futuro, desenvolver uma ferramenta que seja capaz de realizar esse tipo de combinação, pois **Basic Choice** é uma opção de critério de combinação que pode resultar em um quantidade razoável casos de teste se comparado com outros critérios.

Outra possibilidade de extensão deste trabalho seria realizar esta análise comparativa com ferramentas comerciais pagas, pois, uma instituição que está decidida em incorporar o uso de ferramentas para serem usadas no desenvolvimento de testes fundamentados no particionamento do espaço de entrada pode ter capital para investir em uma ferramenta paga que apresente vantagens em relação às gratuitas. E seguindo no mesmo sentido de estender a quantidade de ferramentas estudadas, pode ser possível modificar os critérios de seleção usados aqui de forma a expandir esse estudo ao considerar algumas das ferramentas gratuitas que foram desconsideradas preliminarmente. Por exemplo, as ferramentas que somente possibilitam *Pair-Wise* ou as que foram consideradas obsoletas mas ainda podem apresentar algum potencial de uso.

O critério de usabilidade pode ser explorado de forma mais concreta em trabalhos futuros. O uso de interface gráfica pode ser um diferencial, mas até que ponto? Para projetos grandes seria seja mais vantajoso não utilizar interface gráfica a fim de facilitar a configuração e manutenção do modelo do espaço de entrada do sistema fornecido às ferramentas? Então, pode ser feita a avaliação das ferramentas em questão a escalabilidade e tentar entender como eles se comportam no ciclo de vida de software e sistemas

de proporções diferentes.

REFERÊNCIAS

- AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. Reino Unido: Cambridge University Press, 2016.
- BHARGAVI, S. B. et al. Conventional testing and combinatorial testing: A comparative analysis. In: **2016 International Conference on Inventive Computation Technologies (ICICT)**. [S.l.: s.n.], 2016. v. 1, p. 1–5.
- CORNUTUM. **Tcases Tutorial**. 2021. <<http://www.cornutum.org/tcases/docs/Tcases-Guide.htm#cardinalityConditions>>. Acessado em 12/06/2021.
- EITNER, C.; WOTAWA, F. Crucial tool features for successful combinatorial input parameter testing in an industrial application. In: **Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2019**. [S.l.]: IEEE Computer Society, 2019. p. 188–189. ISBN 9781728108889.
- HAGAR, J. et al. Introducing combinatorial testing in a large organization. **IEE Computer Society**, v. 48, p. 64–72, 04 2015.
- HU, L. et al. How does combinatorial testing perform in the real world: an empirical study. **Empirical Software Engineering**, Springer, v. 25, n. 4, p. 2661–2693, jul 2020. ISSN 15737616.
- KHALSA, S. K. **An Analysis and Extension of Category Partition Testing in the presence of Constraints**. 2017. <https://curve.carleton.ca/system/files/etd/cfc97d04-b80f-4046-81a8-8a46861adbdb/etd_pdf/ad6db1ff0d85c4e7407362a0a40184e3/khalsa-ananalysisandextensionofcategorypartition.pdf>. Tese (Doutorado em Engenharia Elétrica e Computação) - Ottawa-Carleton Institute of Electrical and Computer Engineering, Canadá. Acessado em 12/06/2021.
- KRUSE, P. M. et al. Combinatorial testing tool learnability in an industrial environment. In: **International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2013. p. 304–312. ISSN 19493770.
- KUHN, D. R.; WALLACE, D. R.; GALLO, A. M. Software fault interactions and implications for software testing. **IEEE Transactions on Software Engineering**, v. 30, n. 6, p. 418–421, 2004. ISSN 00985589.
- LI, X. et al. Applying Combinatorial Testing in Industrial Settings. In: **Proceedings - 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016**. [S.l.]: IEEE Computer Society, 2016. p. 53–60. ISBN 9781509041275.
- OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating functional tests. **Communications of the ACM**, v. 31, n. 6, p. 676–686, jun 1988. ISSN 15577317.
- OZCAN, M. An industrial study on applications of combinatorial testing in modern web development. In: **Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2019**. [S.l.]: IEEE Computer Society, 2019. p. 210–213. ISBN 9781728108889.

PEZZÈ, M.; YOUNG, M. **Software Testing and Analysis: Process, Principles, and Techniques**. Estados Unidos: John Wiley & Sons, 2008.

YAO, Y. et al. Design and implementation of combinatorial testing tools. In: **Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017**. [S.l.]: IEEE Computer Society, 2017. p. 320–325. ISBN 9781538620724.