UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

ISABELY DO NASCIMENTO COSTA

OPTIMUS: MECANISMO DE OTIMIZAÇÃO DE SEQUÊNCIA DE CASOS DE TESTES EM SISTEMAS AUTOADAPTATIVOS

FORTALEZA

2024

ISABELY DO NASCIMENTO COSTA

OPTIMUS: MECANISMO DE OTIMIZAÇÃO DE SEQUÊNCIA DE CASOS DE TESTES EM SISTEMAS AUTOADAPTATIVOS

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da computação do Programa de Pós-Graduação em Ciência da computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da computação. Área de Concentração: Engenharia de Software.

Orientadora: Profa. Dra. Rossana Maria de Castro Andrade.
Coorientador: Prof. Dr. Ismayle de Sousa Santos.

FORTALEZA

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C872o Costa, Isabely do Nascimento.
Optimus: Mecanismo de otimização de sequência de casos de testes em sistemas autoadaptativos / Isabely do Nascimento Costa. – 2024.
104 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2024.
Orientação: Profa. Dra. Rossana Maria de Castro Andrade.
Coorientação: Prof. Dr. Ismayle de Sousa Santos.

1. sistemas autoadaptativos. 2. Sensibilidade ao Contexto. 3. Teste em Tempo de Execução. 4. Otimização. I. Título.

CDD 005

ISABELY DO NASCIMENTO COSTA

OPTIMUS: MECANISMO DE OTIMIZAÇÃO DE SEQUÊNCIA DE CASOS DE TESTES EM SISTEMAS AUTOADAPTATIVOS

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da computação do Programa de Pós-Graduação em Ciência da computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da computação. Área de Concentração: Engenharia de Software.

Aprovada em: 30 de Agosto de 2024.

BANCA EXAMINADORA

Profa. Dra. Rossana Maria de Castro Andrade (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Ismayle de Sousa Santos (Coorientador)
Universidade Estadual do Ceará (UECE)

Profa. Dra. Valéria Lelli Leitão Dantas
Universidade Federal do Ceará (UFC)

Prof. Dr. Leonardo Sampaio Rocha
Universidade Estadual do Ceará (UECE)

Profa. Dra. Genaína Nunes Rodrigues
Universidade de Brasília (UnB)

Dedico esse trabalho aos meus pais, Rosângela e Francisco. E ao meu noivo Caio. Compartilhar essa jornada com vocês a tornou mais leve.

AGRADECIMENTOS

A Deus, que em todos os momentos difíceis durante a caminhada Ele sempre esteve comigo e por isso consegui chegar até aqui.

Aos meus pais, Francisco e Rosângela. Agradeço por todo cuidado e apoio. A paz de ter um lar refletiu imensamente em minha trajetória.

Ao meu noivo, Caio. Você sempre acreditou mais em mim do que eu mesma.

Aos meus orientadores, pela paciência e os ensinamentos que tive em todo o período de mestrado.

Aos meus amigos do laboratório GREat, quando entrei no mestrado pensava que poderia caminhar sozinha. Vocês me mostraram o quão importante é ter com quem dividir a jornada.

Ao laboratório GREat, que deu minha primeira oportunidade na área de testes e me motivou a estar onde estou.

Ao CNPq pela auxílio financeiro que tive nos últimos meses de mestrado e a todos os projetos que participei no laboratório.

"Tudo o que temos de decidir é o que fazer com o tempo que nos é dado." (Gandalf)

RESUMO

Os sistemas autoadaptativos, do inglês *Self-Adaptive Systems* (SAS), são sistemas capazes de se modificar automaticamente de acordo com o ambiente no qual estão inseridos. Essas adaptações dinâmicas trazem mais flexibilidade ao sistema, mas também podem resultar em falhas durante a sua execução, problemas com desempenho e operações indesejadas. Para os SAS, as abordagens de teste tradicionais são ineficazes devido aos aspectos dinâmicos desses sistemas, tornando a detecção de falhas uma tarefa complexa. Dessa maneira, várias abordagens de teste para estes sistemas foram propostas na literatura como forma de resolver os principais desafios, sendo uma delas o teste em tempo de execução. No entanto, ainda há uma carência em relação a cobertura e o custo de execução de testes em tempo de execução. Em razão disso, este trabalho propõe um mecanismo para diminuir o custo de execução e auxiliar a cobertura de testes em tempo de execução, com o objetivo de contribuir para a identificação de falhas em SAS. Para avaliar este mecanismo foi desenvolvida uma prova de conceito juntamente com simulações de SAS artificialmente gerados, abrangendo diferentes complexidades e níveis de variabilidade (baixa, média e alta). O mecanismo mostrou-se eficiente em termos de tempo de execução e capaz de selecionar casos de teste eficazes para os objetivos em diferentes cenários. As principais contribuições deste trabalho são: o mecanismo de geração de sequências de casos de teste, que visa minimizar o custo de execução e aumentar a cobertura de testes utilizando a métrica de variabilidade de contexto, e uma ferramenta para geração de casos de teste em binário e cálculo de seus custos.

Palavras-chave: sistemas autoadaptativos; sensibilidade ao contexto; teste em tempo de execução; otimização.

ABSTRACT

Self-adaptive systems (SAS) can modify themselves automatically according to their environment. These dynamic adaptations give the system more flexibility, but it can also result in failures during execution, performance problems, and unwanted operations. For SAS, traditional testing approaches are ineffective due to the dynamic aspects of these systems, making fault detection a complex task. Then, various testing approaches for these systems have been proposed in the literature to solve the main challenges, one of which is runtime testing. However, there is still lack of information regarding the coverage and cost of running tests at runtime. Thus, this research proposes a mechanism to reduce the cost of execution and help cover runtime tests to contribute to the identification of faults in SAS. To evaluate this, a proof of concept was developed along with simulations of artificially generated SAS systems covering different covering different complexities and levels of variability (low, medium and high). The mechanism performed efficiently in terms of execution time and was able to select effective test cases in relation to the objectives in different scenarios. The main contributions of this work are a mechanism for generating sequences of test cases, which aims to minimize the cost of execution and increase test coverage using the context variability metric, and a tool for generating test cases in binary and calculating their costs.

Keywords: self-adaptive systems; context awareness; runtime testing; optimization.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

AG	Algoritmos Genéticos
C-KS	*Context Kripke Structure*
CFM	*Context Feature Model*
COfFEE	*CatalOg of measures for Feature modEl quality Evaluation*
CSP	*Constraint Satisfaction Problems*
DAS	*Dynamically Adaptive System*
DFTS	*Dynamic Feature Transition System*
eCFM	*Extended Context Feature Model*
GT	*Grounded Theory*
IBM	*International Business Machines Corporation*
IoT	*Internet of Things*
ISO	*International Organization for Standardization*
ISO/IEC/IEEE	*ISO (International Organization for Standardization), IEC (International Electrotechnical Commission) and IEEE (Institute of Electrical and Electronics Engineers Standards)*
JSON	*JavaScript Object Notation*
RETAkE	*RuntimE Testing of dynamically Adaptive systEms*
SAS	*Self-Adaptive Systems*
SBSE	*Search-Based Software Engineering*
SBST	*Search-Based Software Testing*
SLR	*Systematic Literature Review*
SUT	*System Under Test*
TestDAS	*Testing method for Dynamic Adaptive System*

SUMÁRIO

1 INTRODUÇÃO

Esta dissertação de mestrado apresenta uma mecanismo que utiliza técnicas de otimização para sequenciar casos de teste, visando reduzir o custo dos testes em sistemas adaptativos e aumentar a diversidade de estados avaliados.

A Seção 1.1 deste capítulo apresenta o contexto em que a pesquisa desta dissertação está inserida, oferecendo uma visão abrangente do tema. A Seção 1.2 destaca a importância e a necessidade do mecanismo proposto. A Seção 1.3 discute o objetivo e a metodologia adotados, detalhando as abordagens e técnicas utilizadas para atingir os resultados esperados. Por fim, na Seção 1.4 é apresentada a estrutura do documento de dissertação.

1.1 Contextualização

Os sistemas de informação modernos estão se tornando cada vez mais complexos. Isso ocorre devido ao aumento no uso de dispositivos móveis e à necessidade de que eles funcionem continuamente em qualquer ambiente. Para atender a essas demandas, a indústria de software teve que se adaptar, utilizando sistemas altamente distribuídos. Operando em contextos altamente diversos, esses sistemas precisam integrar dispositivos especializados e heterogêneos, além de lidar com a variabilidade dos recursos de rede (KRUPITZER *et al.*, 2015). Os sistemas de *Internet of Things* (IoT) e a Indústria 4.0 são exemplos disso (MATALONGA *et al.*, 2022).

Todavia, desenvolver, configurar e manter esses sistemas é uma tarefa muito difícil, sujeita a erros e custosa (KRUPITZER *et al.*, 2015), uma vez que o contexto é imprevisível e pode mudar a qualquer momento, alterando a saída do sistema (PRIYA; RAJALAKSHMI, 2022). Uma solução para esse problema é a autoadaptação, de modo que espera-se que o *software* cumpra os seus requisitos em tempo de execução, em resposta as alterações (SALEHIE; TAHVILDARI, 2009). Este tipo de *software* é chamado de *Self-Adaptive Systems* (SAS) e é capaz de se autorreconfigurar em resposta a mudanças de requisitos e condições ambientais (FREDERICKS *et al.*, 2013)[1].

[1] Alguns autores também utilizam os termos sistemas autônomos e sistemas dinâmicamente adaptativos (HEZAVEHI *et al.*, 2021)

1.2 Motivação

As adaptações dinâmicas dos SAS enquanto eles estão em produção podem levar a alterações em tempo de execução que podem levar a novos riscos de bugs, interações inesperadas, degradação de desempenho e modos de operação indesejados (LAHAMI; KRICHEN, 2021). Além disso, detectar falhas neste tipo de sistema de forma eficaz não é uma tarefa trivial (SIQUEIRA *et al.*, 2016).

No contexto de SAS, as abordagens de teste tradicionais são ineficazes devido às características inerentes a esses sistemas e os seguintes desafios tornam a atividade de teste complexa:

(i) muitas das adaptações são realizadas em tempo de execução (SIQUEIRA *et al.*, 2021);

(ii) a quantidade de cenários gerados a partir de alternativas de adaptação ainda pode ser muito grande e inviável do ponto de vista do teste (SIQUEIRA *et al.*, 2021); e

(iii) a geração automática de casos de testes em um ambiente dinâmico(SANTOS,).

A partir disso, o teste em tempo de execução tem potencial para ser uma solução apropriada para a validação de sistemas autoadaptativos (LAHAMI *et al.*, 2013) e podem atuar de várias formas para resolução dos desafios mencionados (SANTOS, 2020). Contudo, há uma carência em abordagens e ferramentas que gerenciem e executem de forma eficiente os testes em tempo de execução podendo gerar preocupações relacionadas a custo de execução e manutenção dos casos de teste (LAHAMI; KRICHEN, 2021; SILVA *et al.*, 2022). Este cenário motiva a proposta deste trabalho, que visa a criação de um mecanismo de geração de sequências de casos de teste com maior variabilidade e com menor custo.

1.3 Objetivo e Metodologia

Com o objetivo de contribuir para a identificação de falhas em SAS, esta pesquisa propõe um mecanismo que utiliza um algoritmo multiobjetivo de otimização para geração de sequências de casos de testes com maior variabilidade de contexto e com menor custo de execução.

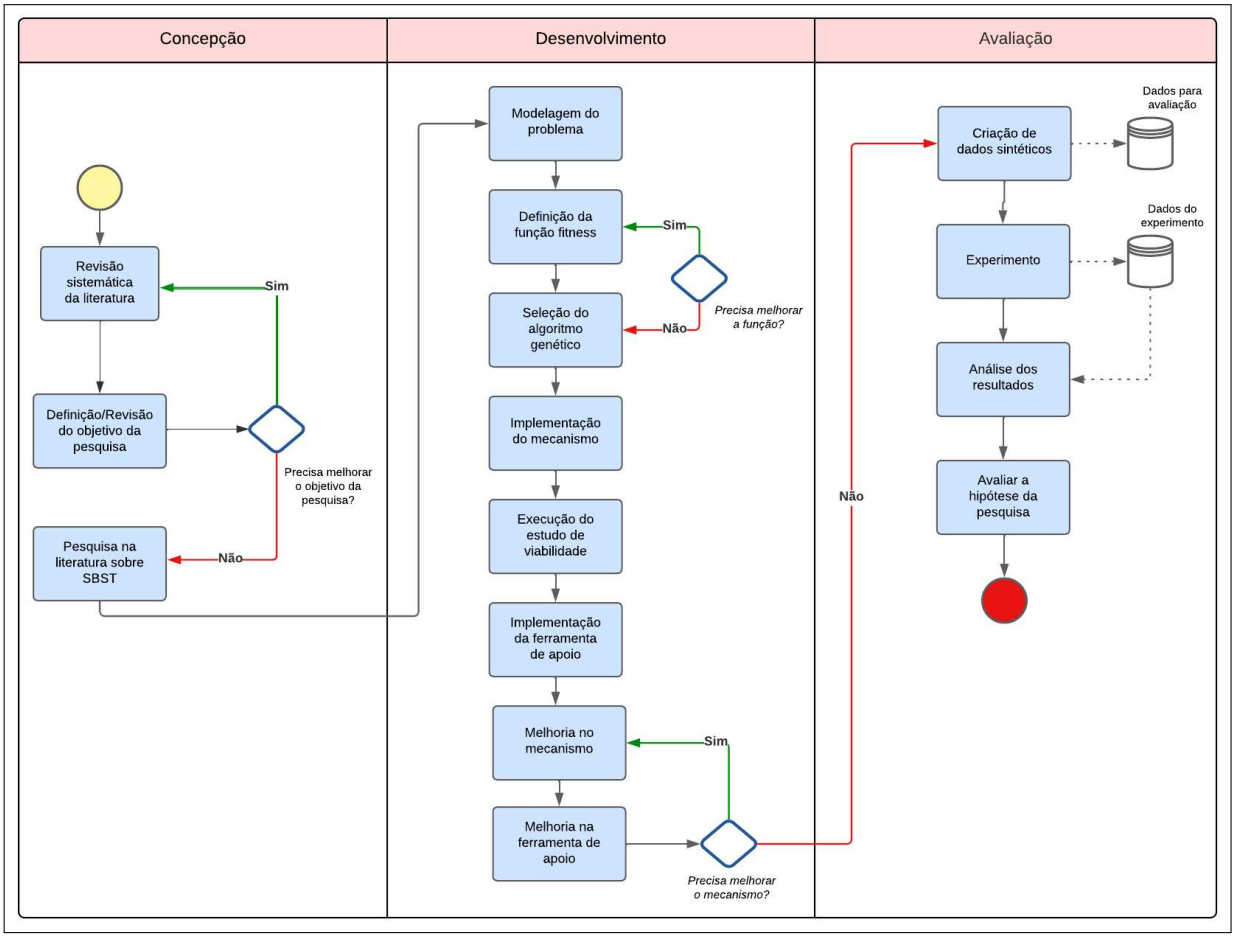
Para atingir este objetivo, esta pesquisa possui as seguintes metas:

– Realizar uma revisão sistemática da literatura para identificar os desafios de testar sistemas adaptativos, as abordagens de teste atuais e as possíveis maneiras de testar SAS;

– Modelar o problema como uma função *fitness*[2];
– Implementar uma ferramenta para auxiliar na geração e cálculo dos custos dos casos de teste;
– Implementar e avaliar o mecanismo proposto para geração de sequências de casos de teste utilizando a função *fitness*;

A Figura 1 apresenta a metodologia utlizada durante este trabalho, a qual foi definida a partir do objetivo e das metas e foi organizada em três fases principais: i) *Concepção*, onde é definido o objetivo do trabalho baseado em uma revisão sistemática da literatura, ii) *Desenvolvimento*, onde o mecanismo é implementado, e, por fim, iii) *Avaliação*, onde o mecanismo é avaliado.

Figura 1 – Metodologia do trabalho



Fonte: elaborada pelo autora.

Na fase de *Concepção* foi realizada uma revisão sistemática da literatura seguindo *guideline* de Kitchenham *et al.* (2016) com o objetivo de identificar os desafios, abordagens, fatores de influência e tendências em testes de sistemas autoadaptativos. A partir dessa revisão,

[2] Uma função *fitness* é uma função objetiva que é utilizada para avaliar soluções (ARRIETA *et al.*, 2019).

foi definido e revisado o objetivo desta pesquisa. Ainda na fase de *Concepção*, uma pesquisa na literatura em busca de trabalhos que abordassem técnicas de *Search-Based Software Testing* (SBST) foi realizado para auxiliar nas atividades da próxima fase.

A fase de *Desenvolvimento* constituiu da modelagem do problema e definição da função *fitness*, seguindo os passos de Harman e Jones (2001) e Harman (2007). Em seguida, foi selecionado o algoritmo para implementação do mecanismo e execução do estudo de viabilidade. Durante o estudo de viabilidade, houve a implementação de uma ferramenta de apoio e o mecanismo e a ferramenta foram avaliados em busca de melhorias.

Por fim, na fase de *Avaliação* o mecanismo foi avaliado por meio de 14 simulações de sistemas SAS sintéticos, abrangendo diferentes complexidades e graus de variabilidade.

1.4 Estrutura da Dissertação

O restante da dissertação está organizada em seis capítulos:

- **Capítulo 2 (Fundamentação Teórica)** descreve os principais conceitos relacionados a dissertação: Sistemas autoadaptativos, otimização em engenharia de *software* e teste de software;
- **Capítulo 3 (Revisão Sistemática da Literatura)** descreve a metodologia e resultados obtidos através de uma revisão sistemática da literatura visando definir e revisar o objetivo desta pesquisa;
- **Capítulo 4 (Trabalhos Relacionados)** compara a proposta desta dissertação com trabalhos encontrados na literatura que abordam otimização e testes em sistemas SAS;
- **Capítulo 5 (Optimus)** apresenta em detalhes o mecanismo proposto nesta dissertação, bem como as etapas para utilização do mesmo;
- **Capítulo 6 (Avaliação)** descreve as avaliações do mecanismo proposto por meio de um estudo de viabilidade e experimentos;
- **Capítulo 7 (Conclusão)** resume as contribuições alcançadas, discute algumas limitações da pesquisa e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos fundamentais para a condução da pesquisa. Na Seção 2.1 é apresentado o conceito de Sistemas autoadaptativos (SAS) e suas propriedades. Ademais, são apresentadas as definições de autoconsciência, consciência de contexto, adaptação e o ciclo MAPE-K. Na Seção 2.2 destaca-se as definições importantes da Otimização em Engenharia de Software SBSE, como aplicar SBSE e apresenta a sub-área *Search-Based Software Testing* (SBST). Por fim, na Seção 2.3 as definições básicas de teste de *software*, atividade de testes e seus objetivos são apresentadas. Além disso, são apresentadas propriedades e desafios do testes em sistemas SAS.

2.1 Sistemas autoadaptativos

Os sistemas baseados em componentes distribuídos podem mudar dinamicamente durante sua execução contínua sem fim. Geralmente, essas mudanças dinâmicas são necessárias para fornecer sistemas mais confiáveis, para apagar deficiências detectadas, ou para apoiar o desenvolvimento rápido dos requisitos dos usuários e a crescente variabilidade dos ambientes de execução (LAHAMI; KRICHEN, 2021). Esses sistemas são chamados de Sistemas autoAdaptativos (em inglês, *Self-adaptive Systems* (SAS)), que podem ser definidos como sistemas que se adaptam em resposta à mudança de condições ambientais (ALVES *et al.*, 2009).

Os *Self-Adaptive Systems* proporcionam as chamadas propriedades fornecem propriedades de autogestão como a autoconfiguração, a autorecuperação na presença de falhas, a autootimização e a autoproteção contra ameaças (KRUPITZER *et al.*, 2015). Para alcançar um comportamento adaptativo, as propriedades básicas do sistema são: autoconsciência e consciência de contexto. A autoconsciência descreve a capacidade de um sistema de estar ciente de si mesmo, ou seja, ser capaz de monitorar seus recursos, estado e comportamento. Consciência de contexto significa que o sistema é ciente de seu ambiente operacional, o chamado contexto (KRUPITZER *et al.*, 2015).

Segundo Abowd *et al.* (1999):

> "O contexto é qualquer informação que pode ser utilizada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto considerado relevante à interação entre um usuário e um aplicativo, incluindo o usuário e os próprios aplicativos."

Por exemplo, a informação de contexto pode ser utilizada pelo sistema para se

adaptar ao nível de bateria de um dispositivo e se o mesmo está ou não ligado a uma fonte de energia (SANTOS,).

A adaptação é então a capacidade de alterar um sistema de acordo com variações de contexto (MULLER *et al.*, 2009) e esta pode acontecer em todos os níveis no sistema (*e.g.* software do sistema, comunicação, recursos técnicos, contexto e aplicação). Por exemplo, aplicações em smartphone que mudam para o modo silencioso quando o usuário está em uma reunião através da utilização de informações do calendário, oferecem adaptação ao nível da aplicação (KRUPITZER *et al.*, 2015). Um exemplo de adaptação de comunicação é mudar a conexão de rede, por exemplo, de 3G para WLAN até um Conexão WLAN está disponível (DOBSON *et al.*, 2006). Os recursos de autocorreção permitem o início automático de sistemas de backup, por exemplo, em um data center, o que altera a nível de recursos técnicos. Um exemplo de adaptação ao contexto é uma sala de reunião inteligente que reduz automaticamente a luz quando uma apresentação começa (KRUPITZER *et al.*, 2015). Em exemplo de adaptação a nível de software do sistema, um *middleware* [1] adaptável oferece a possibilidade para trocar componentes de hardware em tempo de execução (SADJADI; MCKINLEY, 2003). A dinâmica de adaptação, pode ocorrer em tempo de execução. Um sistema autoadaptativo (SAS) é um sistema de software com adaptação em tempo de execução ativada (KRUPITZER *et al.*, 2015; SANTOS,).

Uma vez que uma mudança de contexto é detectada, a lógica de sistemas adaptativos pode usar diferentes tipos de critérios em seu processo de tomada de decisão: modelos, regras/-políticas e objetivos (SANTOS, 2020). Os loops de controle autônomo fornecem um mecanismo genérico de auto-adaptação que muitas vezes é modelado como o ciclo MAPE-K (ou MAPE-K *loop*), este define como os sistemas adaptam seu comportamento para manter seus objetivos controlados, com base em qualquer controle regulatório, rejeição de perturbações ou requisitos de otimização. O MAPE-K loop é dividido em quatro atividades: Monitoramento (M), Análise (A), Planejamento (P) e Execução (E). Essas atividades são baseadas de um Conhecimento (K, da sigla em inglês) (ELEUTéRIO; RUBIRA, 2017; SANTOS, 2020). A Figura 2 apresenta os elementos do MAPE-K loop, segundo a arquitetura proposta pela IBM (COMPUTING *et al.*, 2006). A seguir, são listados os seus elementos com suas respectivas descrições.

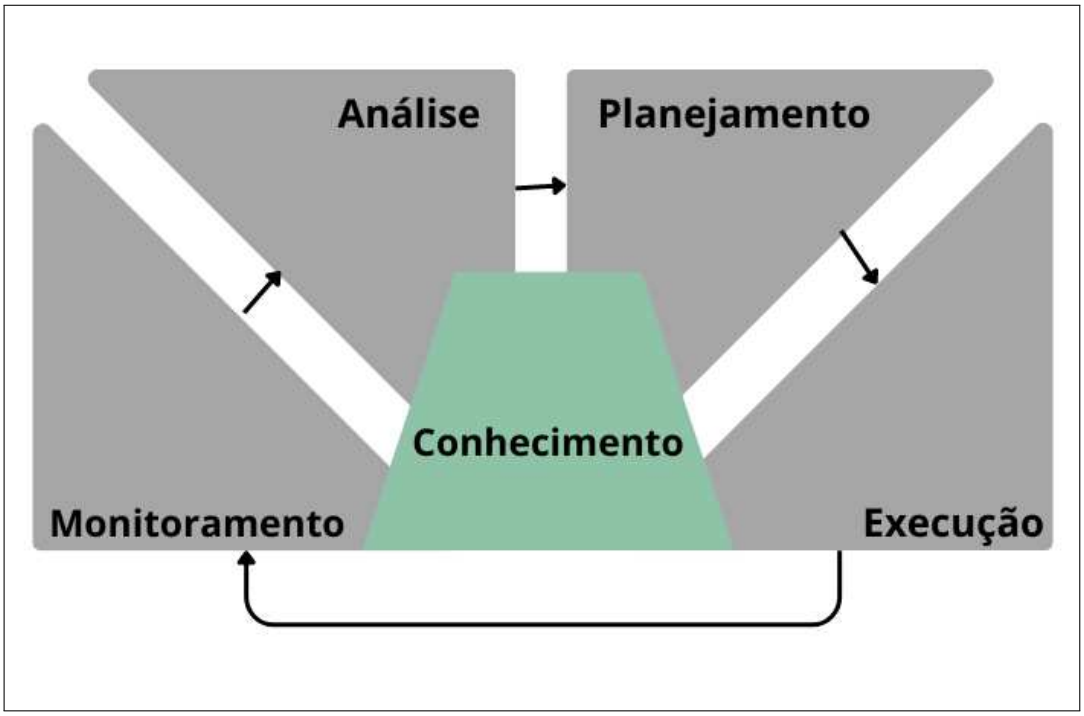
– **Monitoramento:** para deteção e processamento de eventos que possam requerer adaptação,

[1] Uma camada de software que se situa entre a aplicação comercial e a camada de rede de plataformas e protocolos heterogêneos (diversos). Separa as aplicações comerciais de quaisquer dependências da camada de conexão, que consiste em sistemas operativos heterogêneos, plataformas de hardware e protocolos de comunicação (LIGHT; ARUNACHALAN, 2006).

a informação recolhida é enviada para a próxima atividade (SANTOS, 2020; ELEUTéRIO; RUBIRA, 2017);

- **Análise:** com os dados de monitoramento essa atividade correlaciona a informação de contexto para inferir dados do ambiente de tempo de execução e do comportamento do sistema (ELEUTéRIO; RUBIRA, 2017). Em resumo, a função da Análise é prever situações futuras que requerem ações de adaptação (SANTOS, 2020);
- **Planejamento:** a partir da informações de análise, a atividade de planejamento define planos de adaptação (ELEUTéRIO; RUBIRA, 2017);
- **Execução:** esta implementa e executa os planos para adaptação do sistema em execução para obter o comportamento desejado (ELEUTéRIO; RUBIRA, 2017; SANTOS, 2020); e
- **Conhecimento:** é um elemento que funciona como um repositório compartilhado que envia e recebe dados para os demais elementos. Os dados armazenados incluem sintomas de adaptação, políticas, requisições de mudança e planos de mudança (SANTOS, 2020).

Figura 2 – Ciclo MAPE-K



Fonte: adaptado de Computing *et al.* (2006)

2.2 Otimização em Engenharia de Software

Desde a sua emergência como uma técnica de otimização para problemas difíceis de engenharia de software tem sido aplicada com sucesso ao longo do ciclo de vida do desenvolvi-

mento de software (SIMONS, 2013). A conciliação entre técnicas de otimização e Engenharia de Software ficou conhecida como Otimização em Engenharia de Software, em inglês *Search-Based Software Engineering* (SBSE) (MAIA *et al.*, 2013). Pode-se definir otimização como a busca da melhor solução para um dado problema, que consiste em tentar várias soluções e utilizar a informação obtida neste processo de forma a encontrar soluções cada vez melhores (LACERDA; CARVALHO, 1999).

Segundo Harman e Jones (2001), apenas dois componentes são necessários para aplicar o SBSE:

- Uma representação (codificação) do problema (por exemplo, utilizando uma cadeia de bits);
- A definição da função de fitness (por exemplo, similaridade com a consulta de entrada).

As soluções candidatas (que são codificadas a seguir a representação escolhida) são evoluídas (através da aplicação do operações) e são avaliados (pela função de fitness) numa processo iterativo até que uma condição de parada seja cumprida (por exemplo, um número de iterações). Como resultado, soluções ótimas são encontradas para o problema (PÉREZ *et al.*, 2021).

A função *fitness* (ou função quantitativa) é necessária para que o algoritmo possa discriminar entre soluções promissoras e más. Essa noção de qualidade é geralmente definida apenas em termos de métricas software, ainda que os engenheiros de software possam fazer uso de outros mecanismos mais subjetivos para avaliar a qualidade (RAMIREZ *et al.*, 2018). A função *fitness* determina a proximidade entre a solução dada e a solução ótima.

Para a abordagem de objetivo único, a função *fitness* é a função de objetivo do problema, que maximiza ou minimiza para obter soluções ótimas. Por outro lado, a abordagem multiobjetiva tem várias funções objetivo para cada meta, que se maximizam ou minimizam individualmente para obter as soluções ideais. A abordagem multi-objetiva pode comportar-se como uma abordagem de objetivo único se combinarmos as funções objetivo em uma única função *fitness*, atribuindo pesos a cada objetivo de acordo com o seu objetivo (BAJAJ; SANGWAN, 2019).

Os problemas de otimização multiobjetivos podem ser definidos de forma resumida como: encontrar um vetor de variáveis de decisão, que otimiza um vetor de funções objetivas. Sendo as funções objetivas de um problema multiobjetivo, a descrição matemática dos critérios de otimização, que frequentemente estão em conflito entre si (YOO; HARMAN, 2007). Dentro da

multiobjetividade, existe o conceito de Dominância de Pareto (Definição 2.2.1) e de Otimalidade de Pareto (Definição 2.2.2) (VELDHUIZEN *et al.*, 1998).

Definição 2.2.1 (Dominância de Pareto) *Um vetor* $\mathbf{u} = (u_1, ..., u_p)$ *domina um vetor* $\mathbf{v} = (v_1, ..., v_p)$ *se, e somente se,* **u** *é parcialmente menor ou maior (dependendo do objetivo) que* **v***, ou seja,*

$$\forall i \in \{1, ..., p\}, u_i \leq v_i \wedge \exists i \in \{1, ..., p\} : u_i < v_i.$$

Todos os vetores de decisão que não são dominados por nenhum outro vetor de decisão formam o conjunto ótimo de Pareto, enquanto os vetores objetivos correspondentes formam a Fronteira de Pareto (YOO; HARMAN, 2007).

Definição 2.2.2 (Otimalidade de Pareto) *) Uma solução* $x_u \in U$ *é dita ser Pareto ótima se, e somente se, não existe nenhuma outra solução* $x \in U$ *tal que o vetor* $v = f(x) = (v_1, ..., v_p)$ *domine o vetor* $u = f(x_u) = (u_1, ..., u_p)$.

Identificar a Fronteira de Pareto é útil pois pode ser utilizada para uma tomada de decisão bem informada que equilibre as compensações entre os objetivos (YOO; HARMAN, 2007).

Os conceitos abordados encontram aplicação prática por meio de algoritmos de otimização. Sendo o *random search* é o mais simples de se implementar. No entanto, não utiliza uma função de *fitness* (chamado de algoritmo, não guiado), logo não atinge frequentemente soluções globalmente ótimas. Os Algoritmos Genéticos (AG) são considerados pesquisas globais, mostrando muitos pontos no espaço de pesquisa de uma só vez, oferecendo mais robustez aos ótimos locais (HARMAN *et al.*, 2008). Estes utilizam utilizam os conceitos de população e recombinação (GARGARI; KEYVANPOUR, 2022). Frequentemente, um AG utiliza uma representação binária, ou seja, as soluções candidatas são codificadas como sequências de 1s e 0s (HARMAN *et al.*, 2008). Na Figura 4 é descrito em alto nível os passos do algoritmo genético.

Inicialmente, a resposta do problema é formulada como um gene neste algoritmo e um conjunto de respostas é considerado aleatoriamente. Em seguida, dependendo da sua compatibilidade e adequação, três tipos de funções denominadas seleção, crossover e mutação são executadas e são criados novos conjuntos de respostas. Estas respostas substituem as piores respostas do conjunto inicial. Por fim, a resposta é dada satisfazendo a condição de parada (GARGARI; KEYVANPOUR, 2022).

Figura 3 – Algoritmo Genético AG

```
1. Gerar ou semear aleatoriamente a população inicial P;
2. Repetir;
3. Avaliar a aptidão de cada indivíduo em P;
4. Selecionar os pais de P de acordo com o mecanismo de seleção;
5. Recombinar os pais para formar uma nova descendência;
6. Gerar uma nova população P' a partir dos pais e dos descendentes;
7. Mutação de P';
8. P = P';
9. Até atingir a condição de parada.
```

Fonte: adaptado de Gargari e Keyvanpour (2022)

2.3 Teste de software

O teste de software consiste na verificação dinâmica de que um programa fornece comportamentos esperados em um conjunto finito de casos de teste, adequadamente selecionados do domínio de execução geralmente infinito (BOURQUE *et al.*, 2014). Outra definição que pode ser citada, seria a da ISO 29119, que define como um conjunto de atividades realizadas para facilitar a descoberta e/ou avaliação de propriedades de um ou mais itens (HASS, 2014).

A atividade de testes faz parte do processo de Verificação e Validação de software, que tem como objetivo conferir se o produto desenvolvido cumpre sua especificação e funcionalidade para seus usuários (SOMMERVILLE, 2019). Outros objetivos importantes incluem identificação de vulnerabilidades de segurança, avaliação de usabilidade e aceitação de software, para os quais diferentes abordagens seriam tomadas. O propósito de teste varia de acordo com o alvo a ser testado (BOURQUE *et al.*, 2014). Dessa forma, testar é uma forma de corroborar com a qualidade do software desenvolvido (HASS, 2014).

Alguns termos são importantes quando se fala de teste de software, pois ajudam a distinguir a causa de um mau funcionamento e um efeito indesejado no serviço prestado pelo sistema (BOURQUE *et al.*, 2014). Segundo a ISO/IEC/IEEE 24765:2010 (ELECTRICAL; ENGINEERS, 2010), dispõe-se os seguintes termos que auxiliam nesta distinção:

- Engano (*Mistake*) – ação humana que produz um resultado incorreto.
- Defeito (*Fault*) – um passo, processo, ou definição de dados incorreta em um produto de software.
- Erro (*Error*) – diferença entre o valor computado, observado ou medido e o valor teoricamente correto de acordo com a especificação.
- Falha (*Failure*) – inabilidade do sistema ou componente realizar a função requerida, considerando as questões de desempenho exigidas.

A essência do teste de software é determinar um conjunto de casos de teste para o

item a ser testado. Um caso de teste completo deve possuir um identificador de caso de teste, uma breve declaração de propósito (por exemplo, uma regra de negócio), as pré-condições, as entradas reais do caso de teste, as saídas esperadas, pós-condições esperadas e um histórico de execução. O histórico de execução é usado para gerenciamento de teste, nele pode conter a data em que o teste foi executado, a pessoa que o executou, a versão em que foi executado e o resultado (JORGENSEN, 2021). A atividade de teste é dispendiosa, pois o tamanho do conjunto de casos de teste tende a aumentar à medida que o software evolui, e se fosse seguir o seu objetivo ideal seria uma tarefa extremamente exaustiva (BARBOSA *et al.*, 2022). Há outras razões para justificar que testes exaustivos são improváveis como: o domínio de as entradas possíveis de um programa são muito grandes e pode não ser viável simular todas as condições do ambiente do sistema (SANTOS, ; MYERS *et al.*, 2013).

Existem duas maneiras de executar o teste de software: de forma manual e de forma automatizada. No manual, o testador executa os passos especificados pelo caso de teste. Enquanto na forma automatizada, há a utilização de ferramentas de teste que simulam usuários ou processos (MAIA *et al.*, 2013). O uso de ferramentas de teste automatizadas pode minimizar parte do custo do processo de teste (MYERS *et al.*, 2013), mesmo que o processo de implantação dos testes automatizados inicialmente tem um elevado custo, devido à compra de ferramentas apropriadas para a criação e execução dos testes, treinamento da equipe, contratação de pessoas qualificadas, e entre outros (SILVA *et al.*, 2011).

O teste de software geralmente é realizado em diferentes níveis ao longo do desenvolvimento e manutenção processos. Os níveis podem ser definidos com base no objeto de teste, que é chamado o alvo, ou no propósito, que é chamado de objetivo (do nível de teste). O alvo de teste pode ser: um único módulo, grupo de módulo ou o sistema em sua totalidade. Dessa forma, existem três níveis de teste: unidade, integração e de sistema. Sendo o de unidade a verificação de elementos do software de maneira isolada, onde normalmente o testador tem acesso ao código. Já o teste de integração verifica as interações entre os componentes, utilizando estratégias como a de *topdown* (de baixo para cima). E por fim, o teste de sistema que analisa o comportamento do sistema por inteiro (BOURQUE *et al.*, 2014).

Além disso, existem as estratégias de teste, sendo a caixa-preta e a caixa-branca as mais conhecidas (MYERS *et al.*, 2013). A técnica caixa-preta tem como característica os casos de teste dependerem apenas do comportamento de entrada e saída do software. Em contrapartida, a caixa-branca define seus testes de acordo com informações de estrutura de codificação do

software (BOURQUE *et al.*, 2014).

A atividade de teste leva cerca de metade do custo total de desenvolvimento de software, sendo um processo demorado e caro (BAJAJ; SANGWAN, 2019). Dadas as restrições de tempo e custo, uma das principais questões do teste se torna: qual subconjunto de todos os casos de teste possíveis tem a maior probabilidade de detectar a maioria dos erros (MYERS *et al.*, 2013). A partir disso, foram definidos critérios de adequação de teste que podem ser usados para decidir quantos testes serão suficientes ou foram realizados (BOURQUE *et al.*, 2014). Um critério de seleção de teste é um meio de selecionar casos de teste ou determinar que um conjunto de casos de teste é suficiente para um propósito específico. Segundo (COPELAND, 2004), existem cinco critérios básicos para definir até onde deve-se testar um software, são eles: Critérios de cobertura, Taxa de descoberta de defeitos; Custo marginal de encontrar o próximo defeito; Consenso da equipe e Definição do chefe.

A cobertura é uma medida de quanto foi testado em comparação com quanto está disponível para teste. A nível de código pode ser definida com métricas de instrução, cobertura de ramificação e cobertura de caminho. A nível de integração pode ser por meio de quantidade de APIs testadas ou combinações de API e parâmetros. Em nível de sistema, pode ser mensurada por termos de funções testadas, casos de uso ou histórias de usuário testados ou cenários de casos de uso. Uma vez que os casos de testes executados tenham sido suficientes para os critérios de cobertura previamente definidos, pode-se considerar um critério de parada (COPELAND, 2004).

A abordagem de taxa de descoberta de defeitos utiliza do seguinte cálculo: a cada semana (ou curto período de tempo) é contado o número de defeitos descobertos e quando a taxa de descoberta for menor que um limite previamente selecionado os testes podem ser parados (COPELAND, 2004). Algumas situações podem gerar a baixa da taxa, como: criação de testes menos eficazes e testadores de férias, em razão disso (COPELAND, 2004) sugere não depender apenas de um critério para definir a parada dos testes.

O “custo marginal” é associado a uma unidade adicional de produção, que no caso do teste de software seriam os defeitos. O custo de encontrar defeitos vai aumentando uma vez que encontrar os primeiros defeitos é mais simples e menos custosos, enquanto os “próximos” defeitos são mais complexos e consequentemente possuem maior custo. No momento em que o custo do defeito excede a perda que a organização incorreria se entregasse o produto com esse defeito pode-se parar os testes. Vale ressaltar que nem todos os sistemas podem utilizar desse critério de parada, como os que exigem alta confiabilidade. (COPELAND, 2004)

O consenso da equipe pode tomar como base fatores técnicos, financeiros, políticos ou “intuições". A equipe decide que entregar o software após um consenso e assim a atividade de teste é suspensa. E por fim, a definição do “chefe” é associada a entrega do software e consequentemente a parada dos testes quando uma figura de autoridade do produto define que o software deve ser entregue mesmo sem a execução de todos os testes (COPELAND, 2004).

Além disso, existem atividades que auxiliam na determinação de uma ordem de execução mais eficaz (a eficácia dos testes é determinada através da análise um conjunto de execuções de programas) e quais testes devem ser realizados no sistema (FREITAS *et al.*, 2010; BOURQUE *et al.*, 2014). São elas:

- Priorização de testes: Esta atividade trata da determinação da melhor ordem de execução dos casos de teste de um sistema. A definição da qualidade de uma ordem é realizada por meio de uma métrica de cobertura definida matematicamente para calcular o quanto tal ordem executa cedo todo o sistema (FREITAS *et al.*, 2010).
- Seleção de casos de teste: Consiste da escolha de quais testes devem ser realizados em um sistema, seja para a primeira versão ou para versões posteriores (FREITAS *et al.*, 2010).

2.3.1 Teste em SAS

A principal característica do sistema autoadaptativo é que ele pode adaptar-se em tempo de execução de acordo com a informação do contexto. Tanto a utilização da informação de contexto como a reconfiguração do software em tempo de execução, traz vários desafios para a atividade de teste de software (SANTOS,), como:

- Muitas das adaptações são realizadas em tempo de execução (SIQUEIRA *et al.*, 2021), não sendo possível analisá-las durante o tempo de desenvolvimento;
- Mesmo que o ambiente de execução possa restringir o número de possíveis adaptações (SHEVTSOV *et al.*, 2015), a quantidade de cenários gerados a partir de alternativas de adaptação ainda pode ser muito grande e inviável do ponto de vista do teste (SIQUEIRA *et al.*, 2021). Isso ocorre porque muitos dos cenários são imprevistos pelos desenvolvedores (SIQUEIRA *et al.*, 2021);
- A geração automática de casos de testes em um ambiente dinâmico (SANTOS,). O desenvolvimento de testes automatizados em aplicações com estruturas tão complexas torna-se um desafio. (SIQUEIRA *et al.*, 2016).

Os testes durante a fase de concepção (*Design-time*) servem para verificar e validar

que um SAS satisfaz as suas especificações dentro de um determinado conjunto de contextos operacionais previstos (FREDERICKS *et al.*, 2013). Os casos de teste em tempo de concepção são frequentemente estáticos, de forma que o sistema não é executado, e podem ocorrer situações inesperadas por falta de informações em seus requisitos e até mesmo situações de contexto diferentes do esperado. Além disso, podem tornar-se limitados considerando a natureza de auto-reconfiguração dos sistemas autoadaptativos (FREDERICKS *et al.*, 2013).

O teste em tempo de execução tem potencial para ser uma nova solução para a validação de sistemas adaptáveis, devido a dificuldade de identificar em tempo de desenvolvimento todo contexto operacional possível que um SAS pode encontrar em tempo de execução. Segundo (LAHAMI *et al.*, 2015) o teste em tempo de execução (ou *Runtime Testing*) é definido como um método de teste que é realizado em ambiente de execução final de um sistema quando o sistema ou uma parte dele está operacional. Pode ser realizado no momento da implantação ou em tempo de serviço. Para garantir sua alta disponibilidade em tempo de execução, estes sistemas de software são projetados para acomodar novos recursos após os estágios de design e implantação. Eles precisam adaptar-se e evoluir dinamicamente em tempo de execução para atingir novos requisitos e evitar falhas (FREDERICKS *et al.*, 2013). Utilizando as definições de *Field-based Testing Techniques* (BERTOLINO *et al.*, 2021), o teste em tempo de execução que tratamos neste trabalho se adequa a definição de *Online Testing*. O *Online testing* indica atividades de teste de campo realizadas no ambiente de produção no sistema de software real.

Vários métodos foram propostos para apoiar os testes em sistemas autoadaptativos, sendo um deles o *Testing method for Dynamic Adaptive System* (TestDAS) (SANTOS,). O TestDAS utiliza como entrada um modelo de *features* do SAS com as regras de adaptação e um modelo de variação de contexto. O método tem como objetivo a verificação do modelo SAS e gera um conjunto de testes para validar o comportamento adaptativo do sistema.

O TestDAS inicia com a etapa de especificação do SAS usando o *Dynamic Feature Transition System* (DFTS). Na etapa seguinte, há a verificação as propriedades comportamentais do DFTS utilizando uma ferramenta e por fim a geração e execução de testes é feita a partir das propriedades definidas na etapa anterior.

O DFTS modela as mudanças das configurações do SAS de acordo com as mudanças de contexto e as regras de adaptação acionadas. O DFTS é derivado do *Context Kripke Structure* (C-KS) (ROCHA; ANDRADE, 2012) e de um modelo de *features* do SAS com suas regras de adaptação (SANTOS *et al.*, 2016). O DFTS é um grafo cujos nós representam os estados

de contexto e as *features* ativas, enquanto as arestas representam as variações de contexto do sistema (SANTOS, 2020). Este possui dois tipos de proposições atômicas: as proposições de contexto (P_c) e as proposições de *feature* (P_f). As proposições em P_c representam o contexto. As proposições P_f representam todas as características do modelo de *features* (*FM*) do SAS.

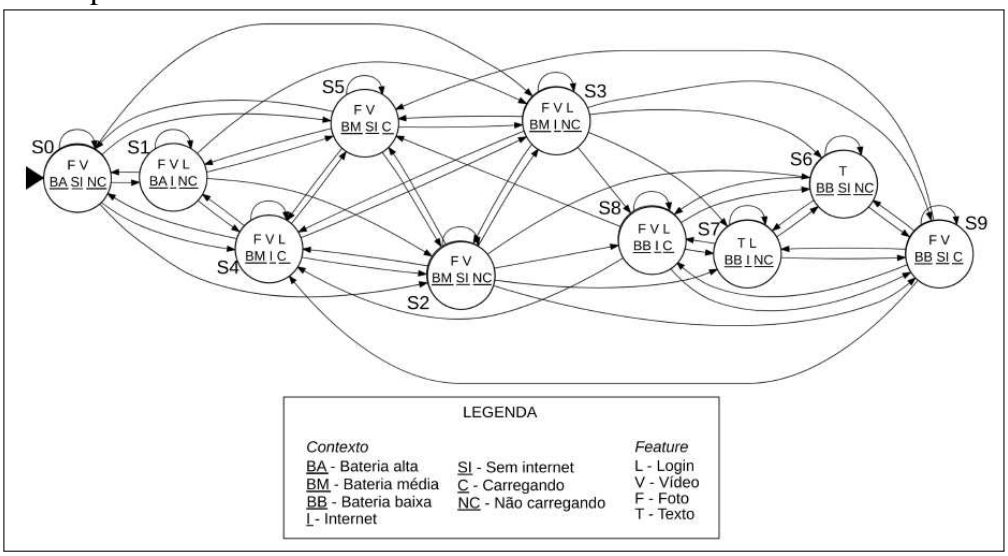
Definição 2.3.1 (DFTS) *Dada uma Context Kripke Structure (C-KS) =* $\langle S, I, C, L \rightarrow \rangle$*, um SAS com modelo de features FM, um conjunto R de regras de adaptação e um conjunto E de configurações inicias do produto, a Dynamic Feature Transition System (DFTS) é dada pela tupla* $\langle S', I', C', L' \rightarrow' \rangle$ *onde Santos et al. (2016):*

- *S' é um conjunto de estados de configuração que contém as features ativas e o estado de contexto atual;*
- *I'* $\subseteq$ *S' é o conjunto de estados de configurações iniciais;*
- *P =* $P_c \uplus P_f$ *é o conjunto de é um conjunto de proposições atômicas que é particionada em contexto e proposições de features. As proposições em* P_c *vem do C-KS. As proposições em* P_f *representam todas as features no modelo de features FM do SAS*
- *L' é uma função de rotulamento tal que L' :S'* $\rightarrow'$ 2^P*; e*
- $\rightarrow' \subseteq (S' \times P_c \times S')$ *é uma relação de transição.*

A Figura 4 ilustra um exemplo de DFTS em uma aplicação que considera as features: Login, Vídeo, Foto e Texto e os contextos: Bateria, Conexão com carregador de bateria e Conexão com a internet. No DFTS, cada nó representa um estado das *features* do sistema e do contexto. Dessa forma, o DFTS reflete os efeitos das regras de adaptação sobre as características do SAS. Por exemplo, no estado S3 possui as features Foto, Vídeo e Login ativadas e OS contextos: Bateria média, Acesso a Internet e Sem conexão com fonte de energia, quando o sistema se reconfigura para o estado S2 havendo a alteração do contexto para Sem conexão com a internet a *feature* de *Login* é desativada.

Uma evolução do trabalho de Santos () é a abordagem *RuntimE Testing of dynamically Adaptive systEms* (RETAkE) de Santos (2020). Esta utiliza os conceitos do modelo de contexto e *features*, tendo como objetivo executar uma sequência de teste no mecanismo de adaptação para verificar as regras de adaptação e checar as propriedades comportamentais. Diferentemente do trabalho de (SANTOS,) que focava em testes em tempo de projeto, o RETAkE tem como objetivo testar a variabilidade sensível ao contexto do SAS durante sua execução, considerando falhas em tempo de execução.

Figura 4 – Exemplo de modelo DFTS



Fonte: Santos (2020)

A abordagem possui 3 etapas que são: Instrumentação do SAS onde o engenheiro de *software* utiliza o modelo DFTS e *Extended Context Feature Model* (eCFM) de (SANTOS *et al.*, 2016) para representar features ativas, contexto atual do sistema e modela a variabilidade do SAS. O eCFM é uma versão extendida do *Context Feature Model* (CFM) proposto por Saller *et al.* (2013) que visa modelar sistemas que adaptam suas *features* em tempo de execução de acordo com o contexto no qual estão inseridos permitindo especificar restrições nas *features*. Essas restrições na features de contexto são feitas através do uso dos Grupos de Contexto que separam em grupos OR e XOR. No grupo XOR, um conjunto de *features* filhas tem relacionamento alternativo com a *feature* pai e somente uma delas pode ser acionada ao mesmo tempo que o pai e no grupo OR, um conjunto de *features* filhas de uma *feature* pai tem um relacionamento de modo que mais de uma delas pode ser ativada ao mesmo tempo em conjunto com o pai.

A segunda etapa do RETAkE é a implantação do SAS na ferramenta proposta pelo autor podendo optar pela execução no ambiente final ou no ambiente controlado. A ferramenta atualiza as regras e modelo de *features* e por fim, inicia a verificação em tempo de execução, com as etapas de checagem de propriedades comportamentais, análise do estado do sistema, geração da sequência de testes e execução dos testes.

A geração dos casos de teste e sequências de testes do RETAkE são baseadas nos conceitos de Santos (). Os casos de testes focam na configuração do sistema pós-reconfiguração e verificam o mecanismo de adaptação que é abstraído em um componente, dessa forma os testes gerados são de nível unitário. A verificação do estado correto das *features* é obtido a partir do modelo eCFM. Para a geração das sequências de teste é definida uma sequência finita de *n*

transições de estado de sistema e testar essas sequências significa avaliar se as features ativas no estado estão de acordo com a ação disparada pelo contexto. Para isso, o engenheiro de *software* deve definir um tamanho *n* para a sequência. Em seguida, o RETAkE seleciona *n* estados no DFTS e utilizando uma adaptação da métrica Diversidade de Contexto de Wang e Chan (2009), Wang *et al.* (2014). Dessa forma para o RETAkE, uma sequência de teste é um número finito de transições consecutivas no DFTS.

Definição 2.3.2 (Diversidade de contexto) *A Diversidade de Contexto (DC) de um fragmento de fluxo de contexto cstream (C) é denotado por DC (cstream(C)) e é definido pela equação:*

$DC(cstream(C)) = \sum_{i=1}^{n-1} HD(ins(C)_i, ins(C)_{i+1})n = |cstream(c)|,$

onde $HD(ins(C)_i, ins(C)_{i+1})i$ *é a distância de Hamming de um par de instâncias de contexto* $ins(C)_i$ *e* $ins(C)_{i+1}$*, e n é o tamanho de um fragmento de stream de contexto C.*

Apesar dos métodos propostos apoiarem os testes, ainda possuem desafios relacionados a necessidade de reduzir o número de testes que são automaticamento gerados (SIQUEIRA *et al.*, 2021; PRIYA; RAJALAKSHMI, 2022) e *overhead* em termos de memória, rede e tempo de execução (SILVA *et al.*, 2022). Portanto, há uma necessidade de explorar técnicas eficazes em termos de custos que reduzam o tempo de teste e os riscos de danificar *hardware*; e técnicas de teste multiobjetivas, *search-based* e verificação de modelos podem ser aplicadas para reduzir os custos dos processos de teste (MATALONGA *et al.*, 2022).

2.3.2 Search-Based Software Testing

Dada a importância da fase de Teste de Software, a subárea denominada Teste de Software Baseado em Busca (em inglês, *Search-Based Software Testing* (SBST)) se destaca em *Search-Based Software Engineering* (SBSE) (MCMINN, 2011; FREITAS *et al.*, 2010). Este destaque é dado principalmente pela quantidade de problemas de teste de software que já se mostraram possíveis de serem modelados e resolvidos através de técnicas de otimização matemática (FREITAS *et al.*, 2010).

A *Search-Based Software Testing* (SBST) é a utilização de técnicas de pesquisa metaheurística otimizada para automatizar total ou parcialmente uma tarefa de teste; por exemplo, a geração automática de dados de teste. A chave para o processo de otimização é a função *fitness* específica ao problema (MCMINN, 2011). Vale destacar que a SBST, complementa as técnicas tradicionais da Engenharia de Software para desenvolvimento de sistemas. Assim, problemas que

não eram completamente resolvidos ou eram resolvidos de maneira insastifastória começaram a ser solucionados com a utilização do SBST. Alguns dos principais problemas estudados são: a geração de dados de teste, seleção de casos de teste e priorização de casos de teste (FREITAS *et al.*, 2010).

3 REVISÃO SISTEMÁTICA DA LITERATURA

Neste capítulo são apresentados os resultados da revisão sistemática da literatura, cujo objetivo foi identificar os desafios, as abordagens e as tendências em testes de sistemas autoadaptativos. Na Seção 3.1 é apresentada a motivação e objetivo da execução da revisão sistemática. Na Seção 3.2 são apresentadas as questões de pesquisa, as questões de extração, bases de dados, string de busca e descrição da metodologia. Na Seção 3.3 são apresentados e discutidos os resultados encontrados ao longo da revisão. Por fim, nesta mesma seção, é apresentado um resumo sobre a revisão, destacando as principais contribuições e implicações a partir dos dados coletados.

3.1 Motivação e objetivo

Uma pesquisa na literatura em busca de revisões sistemáticas da literatura (do inglês, *Systematic Literature Review* (SLR)) que contribuíssem para a justificativa deste trabalho. Contudo, a revisão mais atual encontrada foi a de (SIQUEIRA *et al.*, 2021) que possui apenas trabalhos publicados até 2019 e um trabalho de 2020. Vale ressaltar, que nesta SLR foram identificados poucos trabalhos que utilizavam otimização em testes de sistemas SAS.

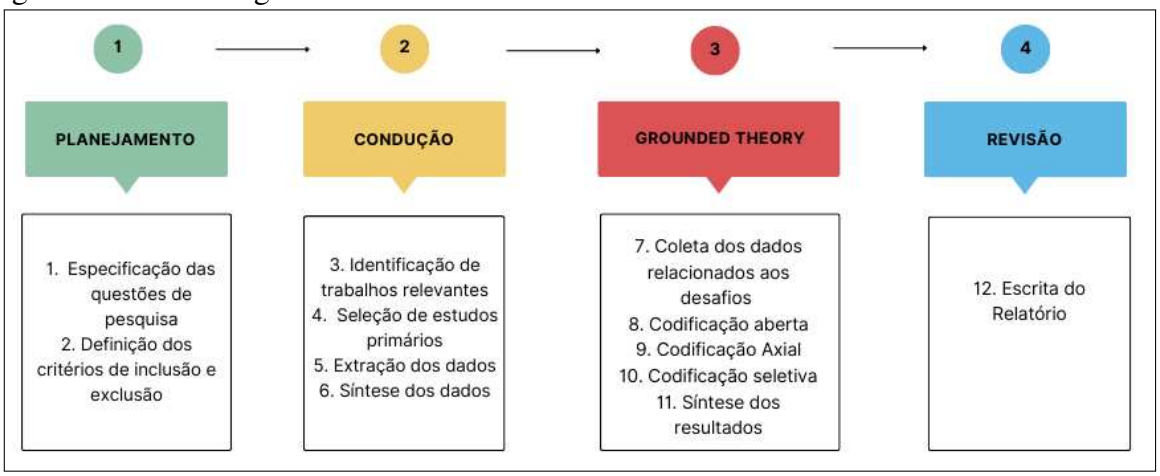
Ao identificar a necessidade de alcançar trabalhos atuais e em busca de realizar uma pesquisa da literatura voltada para o objetivo deste trabalho de mestrado, uma revisão sistemática da literatura foi realizada. Esta SLR teve como objetivo identificar os desafios, abordagens, fatores de influência e tendências em testes de sistemas autoadaptativos.

3.2 Metodologia

A metodologia utilizada para essa SLR foi o *guideline* de Kitchenham *et al.* (2016). Além disso, procedimentos de *Grounded Theory* (GT) (STRAUSS; CORBIN, 1990) foram aplicados para uma parte dos dados obtidos pela etapa de síntese da SLR para uma análise mais aprofundada dos resultados relacionados aos desafios. A Figura 5 ilustra o processo metodológico da revisão sistemática.

Nesta revisão os critérios de inclusão foram: O artigo tratar de teste em SAS e o artigo ser um trabalho primário. O critério de exclusão, por sua vez foi: artigos com uma língua diferente do inglês, artigos publicados antes de 2020. As questões de pesquisa foram definidas a partir do objetivo da revisão sistemática e são as seguintes:

Figura 5 – Metodologia da revisão sistemática da literatura



Fonte: adaptado de Kitchenham *et al.* (2016)

- **RQ1) Quais são as características das abordagens atuais para testar SAS?** Esta questão de pesquisa teve como objetivo apresentar as abordagens de teste para SASs, categorizando o tipo de teste que a abordagem realiza, o nível de teste e o tipo de atividade em que a abordagem está inserida, o tipo de domínio do sistema sob teste ao qual a abordagem é aplicada e quando a abordagem é aplicada.
- **RQ2) Quais são os atuais desafios relacionados ao teste de SAS?** Essa pergunta de pesquisa apresenta os desafios dos testes de SASs, categorizando-os por meio dos procedimentos de *Grounded Theory*.

As seguintes perguntas de extração foram definidas para obter as informações necessárias para responder as perguntas da pesquisa. A relação com as perguntas de pesquisa pode ser visualizada por meio dos IDs, onde as perguntas RQ1.1 a RQ1.3 apoiam a busca pela resposta ä pergunta RQ1, e a RQ2.1 apoia a busca pela resposta da pergunta RQ2.

- **RQ1.1)** Quais são as abordagens para testar sistemas adaptáveis?
- **RQ1.2)** Quais são os tipos de sistema sob teste (SUT) da abordagem?
- **RQ1.3)** As abordagens de teste são aplicadas em tempo de execugäo ou em tempo de projeto?
- **RQ2.1)** Quais são os desafios de testes em SAS?

A estratégia de busca utilizada para pesquisar os artigos foi por meio de pesquisa automática. A string de busca de (SIQUEIRA *et al.*, 2021) foi utilizada. Em razão de ser a revisão sistemática mais atualizada (com artigos até 2019 e um trabalho de 2020) e possuir uma string de busca com *keywords* abrangentes. O guideline de (KITCHENHAM *et al.*, 2016) foi seguido para validar a string de busca e identificar banco de dados relevantes. Em seguida,

alguns artigos encontrados pela string na base de dados do IEEE foram analisados e, com base nos critérios de inclusão e exclusão, confirmou-se que a string estava fornecendo resultados satisfatórios. Por fim, foi efetuada uma pesquisa automatizada.

A motivação para uma nova revisão sistemática em vez de uma atualização da revisão de (SIQUEIRA *et al.*, 2021) baseou-se na lista de verificação 3PDF (MENDES *et al.*, 2020) para definir quando atualizar uma revisão sistemática. Mendes *et al.* (2020) define pode-se dizer que é uma atualização de uma revisão sistemática apenas que for seguida a mesma metodologia da revisão anterior. Além disso, não é possível comparar os resultados de revisões que seguiram protocolos diferentes. A partir dessa definição, foram observadas diferenças em relação a esta revisão e a (SIQUEIRA *et al.*, 2021), como: bases de dados diferentes, este trabalho não utilizou *snowballing*, como os sistemas foram categorizados, critérios de inclusão e exclusão diferentes, a revisão anterior não utilizou *checklist* para avaliar a qualidade dos estudos ou procedimentos de *Grounded Theory* para analisar os resultados.

As bases de dados selecionadas para este trabalho foram a IEEE[1], ACM[2], Scopus[3] e ScienceDirect[4]. Estas bases de dados foram escolhidas devido à sua utilização generalizada pela comunidade acadêmica. Para além disso, foram escolhidas quatro bases de dados para abranger uma maior diversidade de trabalhos.

A seguinte string de busca foi utilizada:

("Testing") AND ("adaptive systems"OR "adaptive system"OR "context aware"OR "context-aware"OR "context awareness"OR "context-awareness"OR "adaptive software"OR "autonomic")

Na etapa de condução, foram encontrados 312 artigos utilizando a string de busca. Foi realizado um filtro para encontrar possíveis trabalhos duplicados, utilizando a ferramenta Parsifal [5], que encontrou 109 duplicados entre as bases de dados. Em seguida, procedeu-se a leitura do título e do resumo dos artigos, utilizando os critérios de inclusão e exclusão, obtendo-se um total de 25 artigos para análise.

Para avaliar a qualidade dos 25 artigos selecionados para o estudo, estes foram analisados de acordo com um checklist de qualidade para avaliar a qualidade dos estudos. Esse checklist foi adaptado para identificar melhor informações relevantes para esta pesquisa, como a

[1] https://ieeexplore.ieee.org/Xplore/home.jsp
[2] https://dl.acm.org/
[3] https://www.scopus.com/
[4] https://www.sciencedirect.com/
[5] https://parsif.al/

descrição da abordagem e a forma como ela é apresentada, com base no checklist e escala de avaliação sugeridas por (KITCHENHAM *et al.*, 2010).

Seguindo a escala de avaliação sugerida por (KITCHENHAM *et al.*, 2010), dez artigos responderam “sim” a todas as perguntas e 15 artigos responderam à maioria das perguntas (mas não a todas) com “sim”. A percentagem mais baixa de perguntas respondidas com “sim” foi de 81,25%, e quanto mais próximo de 100%, melhor a qualidade do trabalho. Esta percentagem foi calculada pelo número de perguntas da lista de verificação dividido pelo número de perguntas respondidas com "sim". Assim, os artigos selecionados têm um grau de qualidade aceitável.

Esta revisão contou com a participação de três pesquisadores. A atividade de seleção foi dividida entre dois, e a revisão e síntese dos resultados foi feita por todos os envolvidos. Além disso, foram realizadas reuniões de alinhamento para garantir que os autores estivessem de acordo na extração das informações, e o índice de concordância foi calculado a partir do teste Kappa (COHEN, 1960) utilizando a ferramenta Jamovi (JAMOVI, 2022) para verificar se os revisores estavam de acordo. Foi obtido um coeficiente de 0.8, que, dentro da escala de interpretação indicada por (KITCHENHAM *et al.*, 2010), se enquadra em "Substancial” e é considerado um bom coeficiente de concordância entre autores.

Em seguida, os dados coletados foram sintetizados em duas formas: RQ1.1, RQ1.2, e RQ1.3, que foram resumidos e analisados para responder à questão de investigação RQ1. Já os dados recolhidos pela questão RQ2 e RQ2.1 foram analisados e sintetizados utilizando procedimentos de *Grounded Theory* (GT). Para os dados recolhidos na fase de extração relacionados com a questão de investigação RQ2, foram aplicados os procedimentos da *Grounded Theory* (GT) para consolidar os resultados obtidos e sistematizar a análise destes dados qualitativos. Os procedimentos efetuados foram:

Codificação aberta: Esta fase envolveu a definição dos códigos e identificadores de códigos, apresentados na Tabela 4, e a sua associação às citações traduzidas. As citações seriam trechos do artigo na fase de extração e os códigos foram elaborados a partir das próprias citações.

Codificação axial: Após a codificação aberta, nesta fase foram definidas as categorias e subcategorias com base nos códigos previamente definidos. Além disso, as categorias e subcategorias foram relacionadas através de proposições de causas e efeitos, condições intervenientes e estratégias de ação. As proposições utilizadas foram: *está associado a*, *é a causa de*, e *faz parte do*.

Codificação selectiva: Finalmente, nesta fase, foi definida a categoria central e

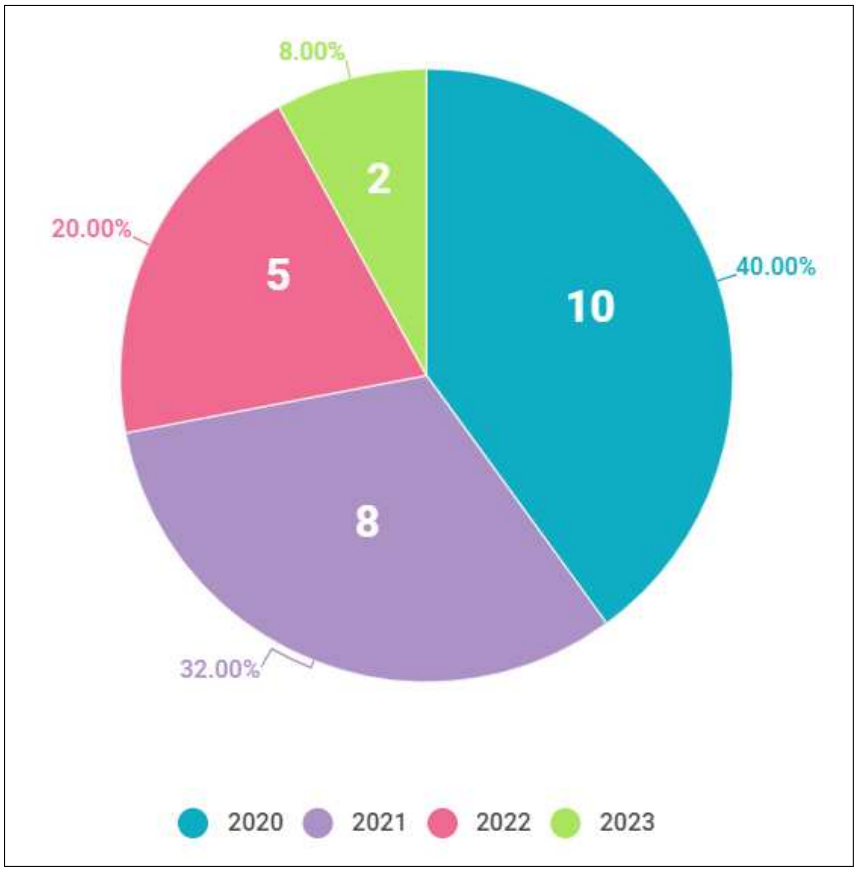
foram revistas as relações entre categorias e subcategorias. Foi criada uma vista de rede para melhorar a visualização dos dados.

3.3 Resultados e discussão

Após a aplicação do processo descrito anteriormente, foram selecionados 25 artigos para responder às questões de pesquisa. Estes artigos estão listados na Tabela 1. Além disso, cada artigo é acompanhado da sua referência e do tipo de sistema tratado na abordagem de teste do artigo (ou seja, Android, Web, Software Embarcado, *Internet of Things* (IoT), Sistemas Ciberfísicos (CPS) e Indefinido).

Foi realizado um filtro dos artigos encontrados por ano de publicação para analisar a quantidade de trabalhos relacionados a abordagens de testes em SASs nos últimos quatro anos. O maior número de publicações foi em 2020 (10 artigos), seguido de 2021 (8 artigos), 2022 (5 artigos) e o menor número 2023 (2 artigos). O baixo número de artigos em 2023 é esperado porque a busca na base de dados foi realizada até o início de 2023, em 16 de março de 2023. Na Figura 6 é possível visualizar o percentual de artigos por ano de publicação.

Figura 6 – Artigos publicados por ano



Fonte: elaborada pelo autora.

Tabela 1 – Artigos selecionados

Ref	Título	SUT
(MICHAELS *et al.*, 2022)	Data Driven Testing for Context Aware Apps	Android
(DADEAU *et al.*, 2022)	Online Testing of Dynamic Reconfigurations w.r.t. Adaptation Policies	Indefinido
(FANITABASI *et al.*, 2020)	A self-integration testbed for decentralized socio-technical systems	IoT
(SANTOS *et al.*, 2021)	Runtime testing of context-aware variability in adaptive systems	Android
(PIPARIA *et al.*, 2021)	Combinatorial Testing of Context Aware Android Applications	Android
(DEVRIES *et al.*, 2021)	Analysis and Monitoring of Cyber-Physical Systems via Environmental Domain Knowledge & Modeling	CPS
(CHEN *et al.*, 2021)	Context-Aware Regression Test Selection	Web
(MANDRIOLI; MAGGIO, 2022)	Testing Self-Adaptive Software With Probabilistic Guarantees on Performace Metrics: Extended and Comparative Results	Indefinido
(SHAFIEI; RAFSANJANI, 2020)	A Test Case Design Method for Context Aware Android Applications	Android
(MIRZA *et al.*, 2021)	ContextDrive: Towards a Functional Scenario-Based Testing Framework for Context-Aware Applications	Indefinido
(YIGITBAS, 2020)	Model-Driven Engineering and Usability Evaluation of Self-Adaptive User Interfaces	Indefinido
(MANDRIOLI; MAGGIO, 2020)	Testing Self-Adaptive Software with Probabilistic Guarantees on Performace Metrics	Indefinido
(ALMEIDA *et al.*, 2020a)	Context-Aware Android Applications Testing	Android
(ALMEIDA *et al.*, 2020b)	ENVIAR: ENVIronment DAta SimulatoR	Android
(CHEN *et al.*, 2020)	Simulated or Physical? An Empirical Study on Input Validation for Context-Aware Systems in Different Environments	CPS
(DORESTE; TRAVASSOS, 2023)	CATS: A Testing Technique to Support the Specification of Test Cases for Context-Aware Software Systems	Indefinido
(USMAN *et al.*, 2020)	TEGDroid: Test case generation approach for android apps considering context and GUI events	Android
(DORESTE; TRAVASSOS, 2020)	Towards supporting the specification of context-aware software system test cases	Indefinido
(YI *et al.*, 2022)	Improving the Exploration Strategy of an Automated Android GUI Testing Tool based on the Q-Learning Algorithm by Selecting Potential Actions	Android
(DADEAU *et al.*, 2020)	Testing adaptation policies for software components	CPS
(DADEAU *et al.*, 2021)	Automated Generation of Initial Configurations for Testing Component Systems	Indefinido
(MAURIO *et al.*, 2021)	Agile services and analysis framework for autonomous and autonomic critical infrastructure	CPS
(SILVA, 2020)	Adaptation oriented test data generation for Adaptive Systems	Indefinido
(CHEN *et al.*, 2022)	Simulation Might Change Your Results: A Comparison of Context-Aware System Input Validation in Simulated and Physical Environments	CPS
(WANG *et al.*, 2023)	Design and implementation of a testing platform for ship control: A case study on the optimal switching controller for ship motion	Embarcado

Fonte: elaborada pelo autora.

Os artigos selecionados também foram categorizados por base de dados, sendo:

– Scopus: Nove publicações (Usman *et al.* (2020), Doreste e Travassos (2020), Dadeau *et al.*

(2020), Dadeau *et al.* (2021), Yi *et al.* (2022), Chen *et al.* (2022), Dadeau *et al.* (2022), Michaels *et al.* (2022), Maurio *et al.* (2021));

- IEEE Xplorer: Sete publicações (Piparia *et al.* (2021), DeVries *et al.* (2021), Chen *et al.* (2021), Mandrioli e Maggio (2022), Shafiei e Rafsanjani (2020), Mirza *et al.* (2021), Silva (2020));
- ACM: Seis publicações (Yigitbas (2020), Mandrioli e Maggio (2020), Almeida *et al.* (2020a), Chen *et al.* (2020), Doreste e Travassos (2023), Almeida *et al.* (2020b))
- ScienceDirect: Três publicações (Fanitabasi *et al.* (2020), Santos *et al.* (2021), Wang *et al.* (2023)).

RQ1) Quais são as características das abordagens actuais para testar o SAS?

Os tipos e níveis de teste de Pierre e Richard (BOURQUE *et al.*, 2014) e as definições de actividades de teste de Garousi *et al.* (2020) foram utilizados para categorizar as abordagens. Assim, as abordagens estão listadas na Tabela 2 por tipo de teste (ex.: Teste de desempenho), por nível do teste (ex.: Unidade) e, finalmente, por tipo de atividade (ex.: Conceção de casos de teste (baseada em critérios)). É importante notar que a atividade de teste indicada na tabela está relacionada com o produto final da abordagem, o que significa que, apesar de a abordagem ajudar noutras actividades, apenas foi considerado o produto final, uma vez que este é o objetivo da abordagem. Além disso, foram utilizadas as siglas “I"e “S"para indicar os níveis de Integração e Sistema na coluna nível de teste, respetivamente. As definições são apresentadas a seguir:

- **Teste de aceitação:** Determina se um sistema satisfaz os seus critérios de aceitação, normalmente verificando os comportamentos desejados do sistema em relação aos requisitos do cliente.
- **Teste de regressão:** É um reteste seletivo de um sistema ou componente para verificar se as modificações não causaram efeitos indesejados e se o sistema ou componente continua a cumprir os requisitos especificados.
- **Teste de desempenho:** Verifica se o software cumpre os requisitos de desempenho especificados e avalia as características de desempenho - por exemplo, capacidade e tempo de resposta.
- **Testes de segurança:** Foca na verificação de que o software está protegido contra ataques externos. Em particular, os testes de segurança verificam a confidencialidade, integridade e disponibilidade dos sistemas e dos seus dados.
- **Testes de interface:** Visa verificar se os componentes estabelecem uma interface

correta para proporcionar a troca correta de dados e informações de controle.

- **Atividade de Design de casos de teste (baseado em critérios):** Design de conjuntos de testes (conjunto de casos de teste) ou requisitos de teste para satisfazer critérios de cobertura.
- **Execução de testes:** Executar casos de teste no sistema sob teste (SUT) e registar os resultados.

Tabela 2 – Categorização da abordagens por tipo, nível e atividade de teste

Ref	Tipo de teste	Nível de teste	Atividade de teste
(FANITABASI *et al.*, 2020)	Performace	I	Execução de teste
(SANTOS *et al.*, 2021)	Aceitação	S	Execução de teste
(PIPARIA *et al.*, 2021)	Aceitação	S	Execução de teste
(DEVRIES *et al.*, 2021)	Aceitação	S	Execução de teste
(CHEN *et al.*, 2021)	Regression	S	Execução de teste
(MANDRIOLI; MAGGIO, 2022)	Performace	S	Execução de teste
(SHAFIEI; RAFSANJANI, 2020)	Aceitação	S	Atividade de Design de casos de teste (baseado em critérios)
(MIRZA *et al.*, 2021)	Aceitação	S	Execução de teste
(YIGITBAS, 2020)	Interface	S	Execução de teste
(MANDRIOLI; MAGGIO, 2020)	Performace	S	Execução de teste
(ALMEIDA *et al.*, 2020a)	Aceitação	S	Execução de teste
(CHEN *et al.*, 2020)	Aceitação	S	Execução de teste
(DORESTE; TRAVASSOS, 2023)	Aceitação	S	Atividade de Design de casos de teste (baseado em critérios)
(USMAN *et al.*, 2020)	Aceitação	S	Execução de teste
(DORESTE; TRAVASSOS, 2020)	Aceitação	S	Atividade de Design de casos de teste (baseado em critérios)
(DADEAU *et al.*, 2020)	Aceitação	S	Execução de teste
(DADEAU *et al.*, 2021)	Aceitação	S	Execução de teste
(YI *et al.*, 2022)	Aceitação	S	Execução de teste
(CHEN *et al.*, 2022)	Aceitação	S	Execução de teste
(DADEAU *et al.*, 2022)	Aceitação	S	Execução de teste
(MICHAELS *et al.*, 2022)	Aceitação	S	Execução de teste
(WANG *et al.*, 2023)	Aceitação	S	Execução de teste
(SILVA, 2020)	Aceitação	S	Execução de teste
(ALMEIDA *et al.*, 2020b)	Aceitação	S	Execução de teste
(MAURIO *et al.*, 2021)	Segurança	S	Execução de teste

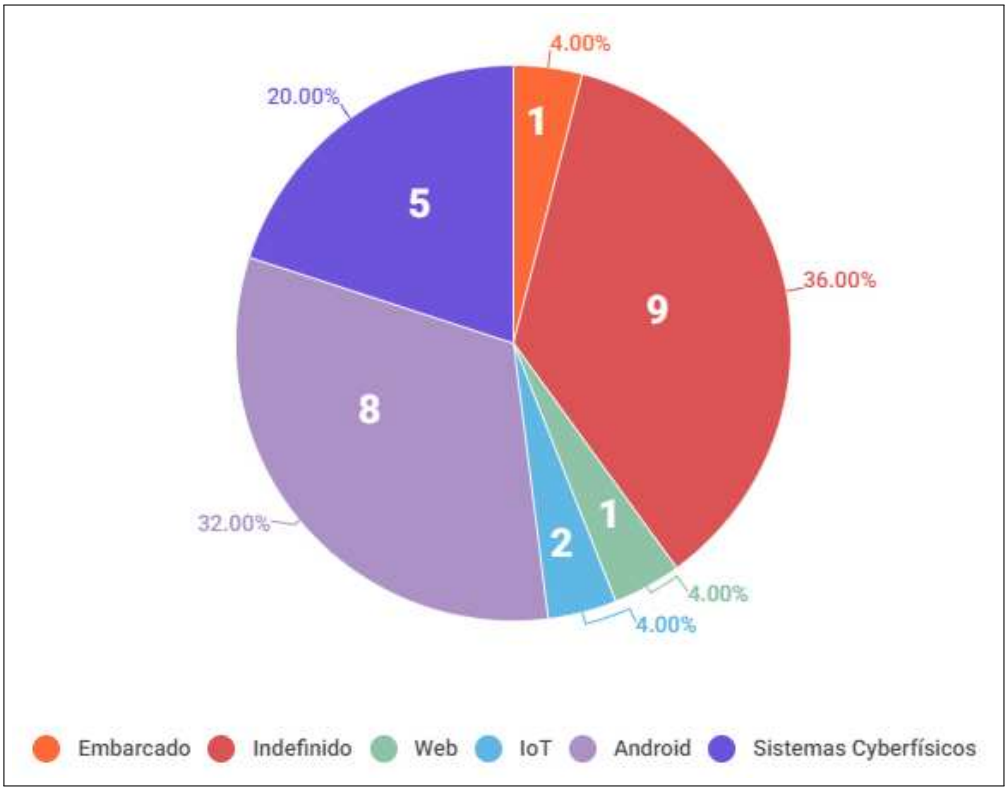
Fonte: elaborada pelo autora.

Com base na categorização da Tabela 2 e seguindo a definição de Pierre e Richard (BOURQUE *et al.*, 2014), foi analisada a porcentagem de abordagens de testes funcionais e não-funcionais. Como resultado, obtivemos 20 abordagens funcionais (80%) (SANTOS *et al.*, 2021) (PIPARIA *et al.*, 2021) (DEVRIES *et al.*, 2021) (CHEN *et al.*, 2021) (SHAFIEI; RAFSANJANI, 2020) (MIRZA *et al.*, 2021) (ALMEIDA *et al.*, 2020a) (CHEN *et al.*, 2020) (DORESTE; TRAVASSOS, 2023) (USMAN *et al.*, 2020) (DORESTE; TRAVASSOS, 2020) (DADEAU *et al.*, 2020) (DADEAU *et al.*, 2021) (YI *et al.*, 2022) (CHEN *et al.*, 2022) (DADEAU

et al., 2022) (MICHAELS *et al.*, 2022) (WANG *et al.*, 2023) (SILVA, 2020) (ALMEIDA *et al.*, 2020b), incluindo Testes de Aceitação e Testes de Regressão, e cinco (20%) (FANITABASI *et al.*, 2020) (MANDRIOLI; MAGGIO, 2022) (YIGITBAS, 2020) (MANDRIOLI; MAGGIO, 2020) (MAURIO *et al.*, 2021) abordagens não-funcionais, divididas em Testes de Segurança, Testes de Interface e Testes de Desempenho.

Conforme mostrado na Tabela 1, não ficou claro na maioria dos artigos (9 artigos) que tipo de SUT a abordagem se destina. No entanto, os principais tipos identificados foram Android (8 artigos), Sistemas ciber-físicos (CPS) (5 artigos) e Web, IoT e Embarcado com um artigo cada. A Figura 7 mostra a percentagem de tipos de sistemas alvo por publicação.

Figura 7 – Percentagem de tipos de SUT por publicação



Fonte: elaborada pelo autora.

A partir da RQ1.3, obtivemos as características elacionadas a quando as abordagens de teste são aplicadas, em tempo de execução ou em tempo de projeto.

Na maioria das publicações analisadas (10 artigos), não foi possível identificar se a abordagem apresentada pelo autor era aplicada em tempo de execução ou em tempo de projeto, pois a distinção entre os dois tipos de execução é o ambiente em que a abordagem será executada. Os artigos não indicam explicitamente o foco. Então, há oito abordagens em tempo de execução, 5 em tempo de projeto e duas que podem ser executadas em tempo de execução e em tempo de

projeto. A tabela 3 relaciona o tempo de execução da abordagem apresentada no artigo com a referência da publicação.

Tabela 3 – Tipo de execução por artigo

Tipo de execução	Artigos
Indefinida	(MIRZA *et al.*, 2021), (CHEN *et al.*, 2020), (DORESTE; TRAVASSOS, 2023), (DORESTE; TRAVASSOS, 2020), (DADEAU *et al.*, 2021), (YI *et al.*, 2022), (CHEN *et al.*, 2022), (DADEAU *et al.*, 2022), (WANG *et al.*, 2023), (SILVA, 2020)
Tempo de execução	(SANTOS *et al.*, 2021), (YIGITBAS, 2020), (ALMEIDA *et al.*, 2020a), (USMAN *et al.*, 2020), (DADEAU *et al.*, 2020), (MICHAELS *et al.*, 2022), (ALMEIDA *et al.*, 2020b), (MAURIO *et al.*, 2021)
Tempo de projeto	(FANITABASI *et al.*, 2020), (PIPARIA *et al.*, 2021), (DEVRIES *et al.*, 2021), (CHEN *et al.*, 2021), (SHAFIEI; RAFSANJANI, 2020)
Ambas	(MANDRIOLI; MAGGIO, 2022), (MANDRIOLI; MAGGIO, 2020)

Fonte: elaborada pelo autora.

Ao analisar os artigos através da RQ1, foi também possível obter dados relativos a abordagens que utilizavam mecanismos de otimização na sua estrutura.

A maioria dos artigos não utiliza otimização nas suas abordagens de teste (20 artigos). Apenas cinco artigos ((FANITABASI *et al.*, 2020), (FANITABASI *et al.*, 2020), (DADEAU *et al.*, 2021), (MAURIO *et al.*, 2021) e (SILVA, 2020)) utilizam mecanismos de otimização para resolver desafios relacionados com testes SAS dentro da definição de *Search-Based Software Engineering*.

RQ2) Quais são os atuais desafios relacionados ao teste de SAS?

Entre os 25 trabalhos analisados, 18 artigos (FANITABASI *et al.*, 2020), (SANTOS *et al.*, 2021), (PIPARIA *et al.*, 2021), (DEVRIES *et al.*, 2021), (MANDRIOLI; MAGGIO, 2022), (MIRZA *et al.*, 2021), (YIGITBAS, 2020), (MANDRIOLI; MAGGIO, 2020), (ALMEIDA *et al.*, 2020a), (CHEN *et al.*, 2020), (USMAN *et al.*, 2020), (DORESTE; TRAVASSOS, 2020), (DADEAU *et al.*, 2020), (YI *et al.*, 2022), (SILVA, 2020), (DADEAU *et al.*, 2022), (ALMEIDA *et al.*, 2020b), (MAURIO *et al.*, 2021) mencionaram desafios de testar SASs e apenas em 7 artigos (CHEN *et al.*, 2021), (SHAFIEI; RAFSANJANI, 2020), (DORESTE; TRAVASSOS, 2023), (DADEAU *et al.*, 2021), (CHEN *et al.*, 2022), (MICHAELS *et al.*, 2022), (WANG *et al.*, 2023) não foi identificada qualquer menção a desafios.

Seguindo os procedimentos *Grounded Theory* (GT) procedeu-se inicialmente a uma codificação aberta onde foram identificadas citações relacionadas com os desafios de testar as

SAS, tendo sido definidos os códigos associados a essas citações. Os códigos da Tabela 4 foram definidos de acordo com a leitura das citações selecionadas. Foram definidos IDs para cada código para melhor organização e manutenção dos códigos.

Tabela 4 – Códigos

ID	Códigos
COD01	A camada de adaptação que reage explicitamente à incerteza
COD02	Dificuldade de detecção de configurações incorretas em tempo de execução
COD03	Como identificar os eventos de contexto de uma aplicação
COD04	Complexidade da atividade de teste
COD05	Custo elevado de manutenção dos testes
COD06	Dados de contexto inconsistentes e imprecisos
COD07	Dependência de monitoramento de contexto dinâmico em tempo de execução para validação e verificação
COD08	Diferentes cenários de execução que podem ser difíceis de reproduzir manualmente
COD09	Ecossistema fragmentado
COD10	Explosão de combinações de cenários
COD11	Falta de abordagens em tempo de execução
COD12	Heterogeneidade do contexto
COD13	Incertezas na mudança que afetam na validade
COD14	Limitação de técnicas de validação e verificação
COD15	Metodologias limitadas que não consideram contexto
COD16	Metodologias limitadas que não consideram adaptação
COD17	Mudança e adaptação contínua
COD18	Necessidade de uma linguagem de modelação de adaptação
COD19	Necessidade de uma linguagem de modelação de contexto
COD20	Plataformas de teste limitadas
COD21	Grande quantidade de eventos GUI e de contexto
COD22	Tempo oneroso para testar muitas combinações
COD23	Custo para testar muitas combinações

Fonte: elaborada pelo autora.

A Tabela 5 refere-se a uma parte da codificação aberta traduzida, uma vez que os artigos selecionados estavam todo em língua inglesa contendo a associação dos artigos, com a citação extraída e o código relacionado à citação.

Na fase de codificação axial, as categorias foram definidas com base nos códigos previamente definidos: A camada de adaptação que reage explicitamente à incerteza (COD01), Complexidade da atividade de teste (COD04) e Limitação das técnicas de validação e verificação (COD14)[7].

Finalmente, na fase de codificação seletiva, a categoria central foi "Desafios de testes em sistemas adaptativos". A Figura 8 mostra o resultado final do *Grounded Theory*, com a categoria central, as subcategorias e suas relações apresentadas por uma visão de rede.

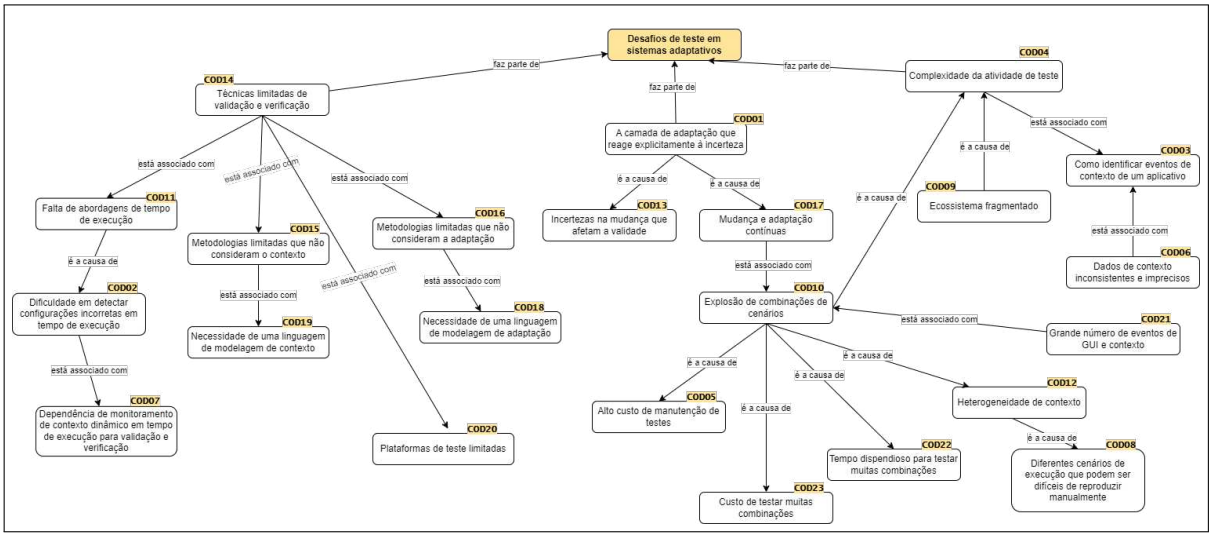
Para identificar os desafios atuais, os dados coletados foram analisados para verificar quantas vezes os códigos da Tabela 4 foram citados nos 25 artigos desta pesquisa. O desafio

Tabela 5 – Parte da codificação aberta traduzida

Ref.	Trecho	ID do Código
(SANTOS *et al.*, 2021)	“Entre os principais desafios, destaca-se a deteção de configurações incorretas em tempo de execução na presença de alterações de contexto. "	COD02
(PIPARIA *et al.*, 2021)	“O grande número de eventos de GUI e eventos de contexto geralmente complicam o processo de teste."	COD22
(PIPARIA *et al.*, 2021)	“Devido às infinitas combinações de eventos e à fragmentação de dispositivos suportados para aplicativos GUI, testá-los é um desafio em termos de tempo e custos."	COD22

Fonte: elaborada pelo autora.

Figura 8 – Visão de rede de categorias e subcategorias



Fonte: elaborada pelo autora.

mais citado entre os artigos foi o *Mudança contínua e adaptação* (citado oito vezes), seguido do *Complexidade da atividade de teste* e do *Explosão de combinações de cenários*, ambos citados sete vezes. Outros desafios envolvem *Incertezas na mudança que afectam a validade* (citado seis vezes) e, finalmente, *Metodologias limitadas que não consideram o contexto* (citado quatro vezes).

As categorias 1 (A camada de adaptação que reage explicitamente à incerteza) e 3 (Técnicas limitadas de validação e verificação) são as que têm mais subcategorias, oito respetivamente. Pode ver-se que os desafios significativos no testede SASs são a camada de adaptação relacionada com:

- Alto custo de manutenção de testes;
- Diferentes cenários de execução que podem ser difíceis de reproduzir manualmente;
- Heterogeneidade do contexto;
- Incertezas na mudança que afetam a validade;
- Mudança e adaptação contínuas;

- Grande número de eventos de GUI e de contexto;
- Tempo dispendioso para testar muitas combinações;
- Custo para testar muitas combinações.

Além disso, a limitação de técnicas para ajudar a validar e verificar esses sistemas, que estão associados a:

- Dificuldade em detectar configurações incorretas em tempo de execução;
- Dependência da monitorização dinâmica do contexto em tempo de execução para validação e verificação;
- Falta de abordagens em tempo de execução;
- Metodologias limitadas que não consideram o contexto ;
- Metodologias limitadas que não têm em conta a adaptação;
- Necessidade de uma linguagem de modelação da adaptação;
- Necessidade de uma linguagem de modelação do contexto;
- Plataformas de teste limitadas.

Além disso, a Categoria 2 (Complexidade da atividade de teste) também é significativa, com quatro subcategorias, pois também se pode verificar que vários fatores tornam o teste destes sistemas complexo (Como identificar os eventos de contexto de uma aplicação, Dados de contexto inconsistentes e imprecisos e Explosão de combinações de cenários).

Com base nos resultados relacionados as características das atuais abordagens de teste em SAS. existem poucas abordagens aos testes não funcionais; apenas cinco artigos tratam deste tipo de testes. Além disso, apenas os testes não-funcionais de Desempenho, Interface e Segurança são abordados. Além disso, a maioria das abordagens centra-se mais na atividade de execução de testes em SASs e em testes ao nível do sistema. Desta forma, abordagens para testes não-funcionais que se concentram em outras atividades e níveis de testes são uma oportunidade de estudo.

A partir dos resultados obtidos pela pergunta de extração RQ1.2, nota-se uma carência de abordagens focadas em sistemas Embarcados, Web e IoT nos últimos anos. Portanto, apresentar novas abordagens para *Self-Adaptive Systems* voltadas para esses sistemas pode ser um tópico interessante para pesquisas futuras.

Pode-se observar que a maioria dos estudos que indicam o tipo de execução da abordagem de teste é voltada para testes em tempo de execução, e apenas dois estudos apresentam abordagens que podem ser tanto em tempo de execução quanto em tempo de projeto. A proposta

de abordagens de teste para SASs flexíveis que possam ser executadas em tempo de execução e em tempo de projeto pode também ser uma oportunidade para investigação futura.

Ademais, foram encontradas poucas abordagens focadas em testes não funcionais; apenas cinco artigos tratam deste tipo de testes. Além disso, apenas os testes não-funcionais de Desempenho, Interface e Segurança são abordados. Além disso, a maioria das abordagens centra-se mais na atividade de execução de testes em SASs e em testes ao nível do sistema. Desta forma, abordagens para testes não-funcionais que se concentram em outras atividades e níveis de testes são uma oportunidade de estudo.

Nota-se também uma carência de abordagens focadas em sistemas Embarcados, Web e IoT nos últimos anos. Portanto, apresentar novas abordagens para SAS voltadas para esses sistemas pode ser um tópico interessante para pesquisas futuras.

Os trabalhos relacionados com as abordagens de otimização são escassos. Apenas cinco artigos em 4 anos tratam de mecanismos de otimização dentro de abordagens de testes em SASs. Vale ressaltar que o uso de mecanismos de otimização em abordagens de testes para SASs pode ser promissor devido aos benefícios já observados com o uso de *Search-Based Software Testing* (SBST) em outros domínios (MCMINN, 2011).

Em resumo, através da revisão sistemática pode-se ter uma visão geral do contexto de testes de SASs nos últimos três anos e vislumbrar desafios e oportunidades. Também foi possível identificar a necessidade de abordagens focadas em sistemas Embarcados, Web e IoT e que apenas duas abordagens são flexíveis quanto ao tipo de execução, sendo possível executá-las em tempo de execução e em tempo de projeto. Além disso, faltam abordagens voltadas para testes não funcionais que suportem vários níveis e atividades de testes. Vale ressaltar ainda que, dos 25 artigos selecionados, apenas cinco aplicavam mecanismos de otimização em suas abordagens de teste.

Por fim, nota-se que existem vários desafios relacionados ao teste de SASs, principalmente ligados à camada de adaptação, ao número limitado de técnicas de teste e à complexidade do teste desses sistemas. Os resultados podem então auxiliar futuras pesquisas sobre testes de sistemas autoadaptativos e incentivar a produção científica que busca mitigar os desafios identificados.

4 TRABALHOS RELACIONADOS

Neste capítulo são apresentados trabalhos relacionados a esta pesquisa, identificados por meio da revisão sistemática da literatura descrita no Capítulo 3. A Seção 4.1 enumera os trabalhos que utilizam otimização para auxiliar na execução de abordagens de teste em sistemas SAS e, por fim, a Seção 4.2 apresenta uma comparação entre os trabalhos relacionados e ao que é proposto nesta dissertação.

4.1 Abordagens de teste para SAS com otimização

Os sistemas autoadaptativos têm a característica de se ajustar em tempo de execução com base nas informações do contexto. Essa característica acarreta desafios relacionados à quantidade de adaptações realizadas durante a execução, à quantidade e complexidade dos cenários de teste para esses sistemas (SIQUEIRA *et al.*, 2021; FANITABASI *et al.*, 2020), e às dificuldades associadas à automatização dos testes nesse contexto (SIQUEIRA *et al.*, 2016).

Diversos métodos foram propostos para apoiar os testes em sistemas autoadaptativos, com destaque para os trabalhos de Dadeau *et al.* (2021), Maurio *et al.* (2021), Mandrioli e Maggio (2022) e Santos (2020), que são apresentados em detalhes nesta seção.

Dadeau *et al.* (2021) apresentam uma abordagem para gerar automaticamente configurações para testar sistemas baseados em componentes. Para tanto, um algoritmo combinatório é utilizado para enumerar todas as soluções possíveis sem simetria do *Constraint Satisfaction Problems* (CSP)[1] definido pelo modelo componente, a fim de produzir configurações iniciais. Este algoritmo integra padrões de eliminação de simetria que reduzem as combinações a serem consideradas.

No trabalho de Maurio *et al.* (2021), os autores descrevem duas abordagens de teste de segurança de sistemas físicos cibernéticos. A primeira abordagem reabilita os sistemas de controle industrial com propriedades autônomas permitindo detectar e recuperar automaticamente de ciberataques e outras falhas através da utilização de microsserviços que reconfiguram os sistemas de forma dinâmica durante os ataques ou falhas. A segunda abordagem utiliza agentes inteligentes numa modelação e quadro de simulação para testar a resiliência de sistemas aéreos autônomos não tripulados. Usando uma abordagem de programação de restrição baseada em algoritmo genético, o escalonador/alocador produz uma configuração inicial contendo os horários

[1] *Constraint Satisfaction Problems* (CSP) são problemas matemáticos definidos como um conjunto de objetos cujo estado dos mesmos deve satisfazer uma série de restrições

de início e locais para todos os microsserviços. Os autores indicam que as duas abordagens em conjunto fornecem a garantia e a resiliência dos sitemas de continuar a operar durante falhas e ataques, e um mecanismo para testar a sua resiliência sob uma série de condições de funcionamento.

Mandrioli e Maggio (2022) apresentam uma abordagem para aspectos não funcionais. Estes buscaram encontrar limites para um desempenho parâmetro de um sistema adaptativo (ou seja, do software e de uma dada estratégia de adaptação implementada sobre o mesmo). Na metodologia de teste proposta pelos autores, a variável de decisão é o desempenho de pior caso da estratégia de adaptação (ou seja, o melhor valor da métrica de desempenho que se garantir com segurança), a função de custo é o próprio desempenho de pior caso e cada um dos os resultados dos testes são uma restrição. A avaliação do limite de desempenho torna-se então, o problema de otimização: maximizar o desempenho que sempre pode ser garantido, sob a restrição de que não pode exceder o que é experimentado nos testes realizados. O objetivo desta abordagem é fornecer garantias empíricas sobre o comportamento do sistema.

O RETAkE de Santos (2020) é uma abordagem para executar o teste em tempo de execução com base na variabilidade do contexto do sistema e na modelação de características. O RETAkE testa o mecanismo de adaptação, permitindo a verificação das suas regras de adaptação com o modelo de variabilidade do sistema. O teste em tempo de execução é apoiado pela verificação das propriedades comportamentais. Esta abordagem gera sequências de testes, contudo de forma aleatória e utilizando o algoritmo de Hamming para calcular a distância de Hamming entre o contexto do estado atual do SAS no *Dynamic Feature Transition System* (DFTS) em relação aos seus vizinhos. Esta abordagem já gera sequências de testes, contudo de forma aleatória e utilizando o algoritmo de Hamming para calcular a distância de Hamming entre o contexto do estado atual do SAS no DFTS em relação aos seus vizinhos.

4.2 Comparação com o trabalho proposto

Diferente dos trabalhos relacionados identificados na literatura, o objetivo deste trabalho é trazer um mecanismo de otimização de sequências de testes que possa ser utilizado de forma ampla em abordagens de testes em sistemas DAS. Assim, as abordagens que busquem os objetivos de diversidade de contexto, cobertura de teste e custo de execução podem utilizar este mecanismo. A Tabela 6 resume as diferenças entre os trabalhos encontrados na literatura e o proposto neste estudo, indicando com "V"os elementos presentes no trabalho e com "X"os

ausentes.

Tabela 6 – Trabalhos relacionados e o Optimus

Trabalho	***Runtime Testing***	**Tipo de teste**	**Otimização**	**Algoritmos genéticos**
Dadeau *et al.* (2021)	X	Funcional	V	X
Maurio *et al.* (2021)	V	Segurança	V	V
Mandrioli e Maggio (2022)	V	Performace	V	X
Santos (2020)	V	Funcional	X	X
Optimus	V	Funcional	V	V

Fonte: elaborada pelo autora.

5 OPTIMUS

Neste capítulo é apresentada a proposta desta dissertação, o mecanismo Optimus [1]. Primeiro, uma visão geral do mecanismo é descrita (Seção 5.1). Ademais, são apresentadas as etapas para aplicação do mecanismo (Seção 5.2) nas quais se apresenta sobre a especificação do SUT (Seção 5.2.1), utilização da ferramenta Bumblebin (Seção 5.2.2) e mais detalhes do Optimus (Seção 5.2.3).

5.1 Visão Geral

Neste trabalho de mestrado é proposto o mecanismo Optimus que busca auxiliar na geração de sequências de casos de teste para *Self-Adaptive Systems* (SAS), utilizando algoritmo genético para maximizar a cobertura de testes e minimizar o custo de execução dos testes. Isso é feito selecionando os melhores casos de teste a partir de parâmetros. A relevância do mecanismo é vista por meio resultados encontrados durante a execução da revisão sistemática da literatura (descrita no Capítulo 3), onde entre os desafios mais citados por autores está a explosão combinatorial de cenários e o custo de testar muitas combinações. Ademais, poucos trabalhos foram encontrados na literatura que utilizassem de otimização para testes em SAS, mesmo com os benefícios já observados em outros domínios (MCMINN, 2011).

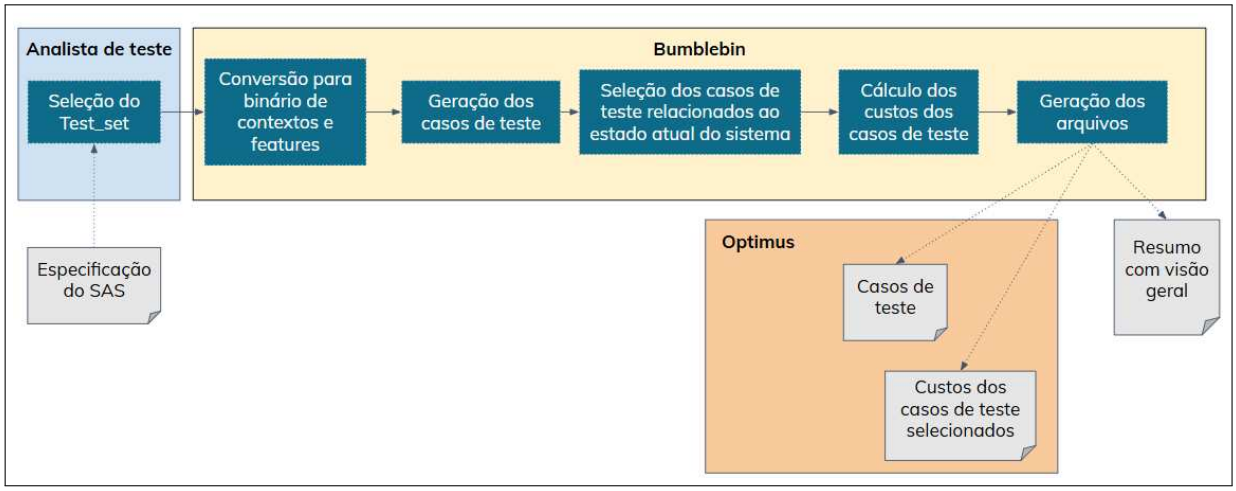
O mecanismo é desacoplado de abordagens e sistemas, o que facilita sua integração e evita impactos no código da aplicação ou do sistema a ser testado. Ademais, o Optimus utilizou e evoluiu alguns os conceitos dos trabalhos de Santos () e Santos (2020). Santos () apresenta um método de teste em SAS (TestDAS) baseado no *Dynamic Feature Transition System* (DFTS) que envolve uma abordagem de verificação de modelos para identificar falhas de *design* na geração de testes para validar os SASs em tempo de projeto. Enquanto isso, o RETAkE de Santos (2020) busca testar a variabilidade sensível ao contexto de SAS em tempo de execução, isto é, considerando falhas ocorridas durante a execução do sistema em seu ambiente final. Por sua vez, o Optimus é um mecanismo que pode auxiliar, por exemplo, na otimização da execução dos testes de Santos () e Santos (2020). Assim como de outras abordagens ou métodos de teste em sistemas SAS que utilizem o modelo DFTS. Outra característica do mecanismo é que este pode ser aplicado tanto para testes em tempo de execução, como também em testes em *Design-time*.

[1] O nome Optimus foi inspirado no robô Optimus Prime do filme Transformers. Uma vez que o mecanismo pretende receber informações e transformá-las, assim como o robô Optimus prime possui essa caractetística de transformação.

Neste trabalho focamos em testes em tempo de execução.

O mecanismo recebe casos de testes e gera uma sequência otimizada por meio das métricas de diversidade de contexto e custo da *feature*. A Figura 9 apresenta uma visão geral de como funciona a aplicação do mecanismo. Inicialmente, o analista de testes especifica o SAS definindo quais features serão testadas, o custo da *feature*, o estado do sistema (podendo ser o atual ou o que se deseja testar) e os grupos de contexto (ver Seção 2.3.1). Em seguida, ele utiliza o conversor binário de features e contextos (Bumblebin) para geração e seleção de casos de teste a partir do contexto atual. Por fim, o Optimus avalia os casos de testes por meio da função *fitness* tendo como objetivos: maximização da diversidade de contexto e minimização do custo da feature e retorna a sequência de testes com os melhores casos de teste.

Figura 9 – Visão geral das etapas para uso do Optimus



Fonte: elaborada pelo autora.

5.2 Etapas para o uso do Optimus

Para a execução do Optimus faz-se necessária a realização de algumas etapas para preparar as informações a serem recebidas pelo mecanismo. As seções 5.2.1, 5.2.2 e 5.2.3 elucidam cada etapa.

5.2.1 Analista de testes - Especificar SAS

Inicialmente, para a geração dos casos de teste o analista de teste deve modelar o *System Under Test* (SUT) no modelo de *features* usando o *Dynamic Feature Transition System* (DFTS). Para isso, foi feita uma adaptação do modelo DFTS proposto por (SANTOS, 2020). O

modelo utilizado por (SANTOS, 2020) define que em um arquivo *JavaScript Object Notation* (JSON) devem ser determinadas as *features*, contextos e caminhos, seguindo a definição do DFTS. Para este trabalho, essa estrutura foi alterada, de forma que devem ser definidas: *features* a serem testadas, custo de cada *feature*, o estado do sistema (que se deseja testar ou estado atual, sendo o estado atual igual a(s) *feature(s)* e contexto(s) ativados) e os conjuntos de contextos relacionados as *features*. Esta alteração foi realizada para que não houvesse a necessidade de definir caminhos para a realização dos testes, uma vez que os casos de teste devem ser escritos para condições de entrada que são inválidas e inesperadas, bem como para aquelas que são válidas e esperadas (MYERS *et al.*, 2013).

Assim, diferentemente da abordagem de geração de Santos () e Santos (2020) o Optimus gera uma sequência de testes baseada em um contexto do sistema, consequentemente expandindo a variabilidade de cenários a serem testados. Para utilizar o Optimus em testes realizados em *design-time*, o analista de teste pode especificar qualquer estado que deseja testar. Para testes em tempo de execução, deve ser utilizado o estado atual do sistema. Orienta-se que o analista concentre-se nas features e contextos que deseja testar, selecionando assim o Test_set. O Test_set para este trabalho é configurado para possibilitar que o analista teste diversos grupos de features e contextos especificados em um único arquivo JSON.

Ademais, foi acrescentada a definição dos custos das *features* baseada na definição dos autores Santos *et al.* (2018), que sugerem que o custo seja calculado considerando:

- O custo inerente de usar a *feature*, que inclui o valor de uso da *feature* e seus ativos que foram criados durante o processo de engenharia;
- O custo médio de consumo de energia da feature no ambiente de implantação;
- O custo de personalizar ativos da feature para o sistema;
- O custo da inserção da lógica de adaptação do tempo de execução (esse custo está relacionado à percepção do contexto).

Para este trabalho, os custos das *features* foram definidos entre 0 e 5 de acordo com nível de complexidade de atividades consideradas. O nível de complexidade envolve a dificuldade de execução da feature, o tempo gasto e o impacto no sistema, onde tem-se o valor:

- 0, se não existir;
- 1, se for muito baixo;
- 2, se for baixo;
- 3, se for moderado;

- 4, se for alto;
- 5, se for muito alto.

É importante destacar que esta é apenas uma sugestão para a definição dos custos de cada *feature*, e o analista de teste tem liberdade para estabelecer esses valores, desde que se mantenha a escala de zero a cinco. No entanto, a definição do custo é essencial para que o mecanismo possa selecionar os casos de teste levando em consideração o custo de cada um.

A Figura 10 ilustra como o arquivo JSON de Santos (2020) (A) é estruturado em comparação com o arquivo desta pesquisa (B). No arquivo de Santos (2020) existe o campo *edges* que é organizado por id indicando o caminho no DFTS a ser seguido, em contrapartida no arquivo do Optimus possui o grupo de contexto e adicionalmente o campo de estado do sistema (atual ou o estado que se deseja testar) que possui as *features* ativadas e contextos; e o campo de custo das *features*. O campo de estado do sistema é indicado em formato binário seguindo a definição de Santos *et al.* (2018), onde as *features* e contextos ativados devem ser representados por 1 e os desativados por zero.

Figura 10 – Exemplos de (A) DFTS no RETAkE e (B) DFTS no Optimus

Fonte: elaborada pelo autora.

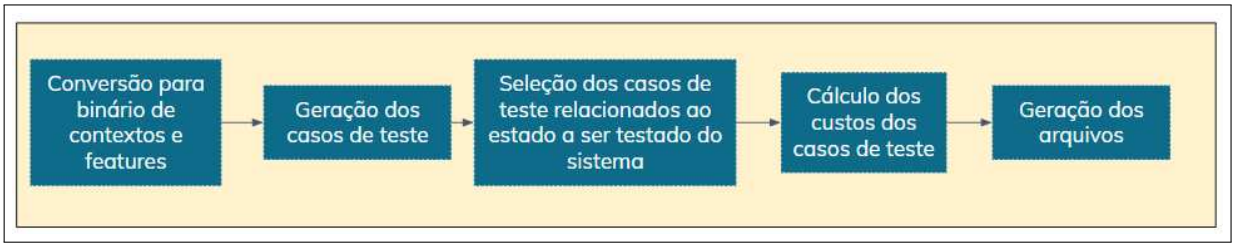
Na Figura 10 (B), o campo **"Test_set"** indica um idenficador do conjunto de *features* e grupos de contexto para teste que é igual a 1. No campo **"features"** são identificadas as *features*

a serem testadas, que são: *Login*, *Video*, *Photo* e *Text*. Em seguida no campo **"features_cost"** são definidos os custos de cada *feature* a ser testada, sendo: Login com custo = 1, Video = 3, Photo = 5 e Text = 4. O campo **"current_context"** indica a partir de qual estado será testado (no caso é o atual). O campo **"feat"** informa quais *features* estão ativas no estado a ser testado, que são todas: 1111 ("Login", "Video", "Photo", "Text"). O campo **"cont"** são os contextos ativados no estado a ser testado, sendo: 0011010, que quer dizer que está: "Low Battery"(001), "No Internet"(10) e "Not charging"(10). A combinação do campo feat e cont formam o estado a ser testado: 11110011010. Por fim, o campo **"context_or"** apresenta os grupos de contexto do tipo OR relacionados as *features* que são: "High Battery", "Medium Battery", "Low Battery", "No Internet", "Internet", "Not charging"e "Charging".

5.2.2 Bumblebin

O Bumblebin [2] é uma ferramenta desenvolvida nesta pesquisa para gerar casos de teste em binário, filtrar os relacionados ao estado do sistema e calcular seus custos. A modelagem com números binários foi escolhida pela eficiência no armazenamento (NOROUZI *et al.*, 2012) e pela capacidade de gerar infinitas combinações. A implementação dessa ferramenta se tornou necessária, em razão de que as primeiras versões do mecanismo Optimus agregavam várias etapas: de filtragem, geração e seleção dos casos de teste. Assim, o desempenho das primeiras versões do Optimus era inferior e com o Bumblebin pode-se separar as atividades e melhorar o desempenho do mecanismo. Além disso, resultou em uma ferramenta desacoplada que pode ser utilizada em outros cenários, como por exemplo, quando um analista de teste precisa de todos os possíveis casos de testes e seus respectivos custos. A Figura 11 apresentação uma visão geral das atividades realizadas pelo Bumblebin.

Figura 11 – Atividades realizadas pelo Bumblebin



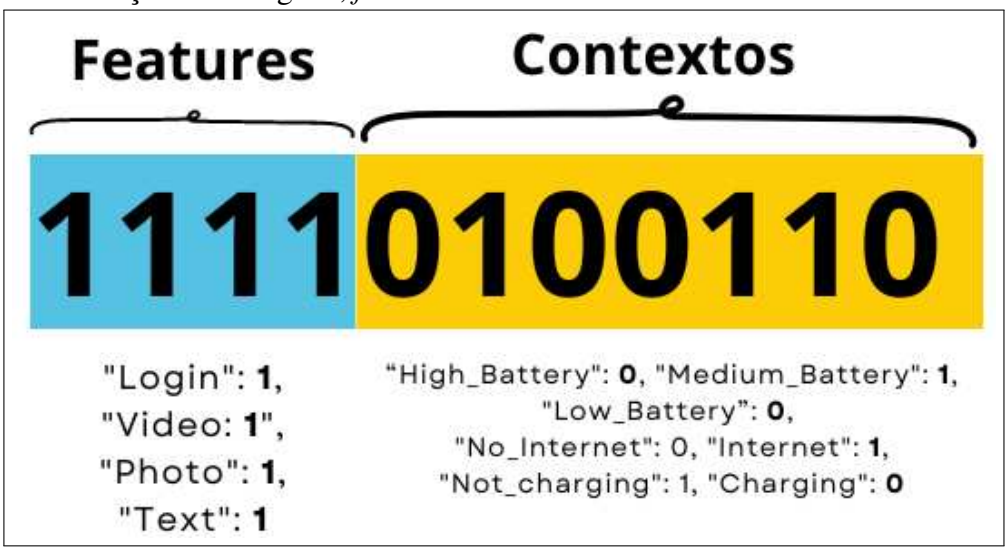
Fonte: elaborada pelo autora.

Após a especificação do SAS o Bumblebin recebe o arquivo JSON e retorna três

[2] O nome Bumblebin foi inspirado no robô Bumblebee do Transformers. Este robô também possui a capacidade de se transformar, assim como a ferramenta propõe a transformação de informações.

arquivos de texto, um contendo todas as informações geradas pela ferramenta (as *features*, os custo das *features*, os contextos e todos os possíveis casos de testes), outro contendo os casos de teste filtrados pelo estado do sistema e por fim; um com os custos dos casos de teste filtrados. Os casos de teste definidos pelo Bumblebin tem como base o conceito aplicado no RETAkE de Santos (2020), onde cada estado do sistema no DFTS gera um caso de teste para avaliar o mecanismo de adaptação. Por exemplo, na Figura 10 B, um caso de teste seria: 11110100110, onde os 4 primeiros dígitos são relacionados as features e os outros 7 dígitos são relacionados ao contexto, a Figura 12 ilustra a relação entre dígitos, *features* e contextos. A presente dissertação não tem o propósito de propor uma nova forma de avaliação ou discussão sobre a execução dos casos de teste. O objetivo é a geração de sequências de configurações (casos de teste) com maior variabilidade e menor custo.

Figura 12 – Relação entre dígitos, *features* e contextos

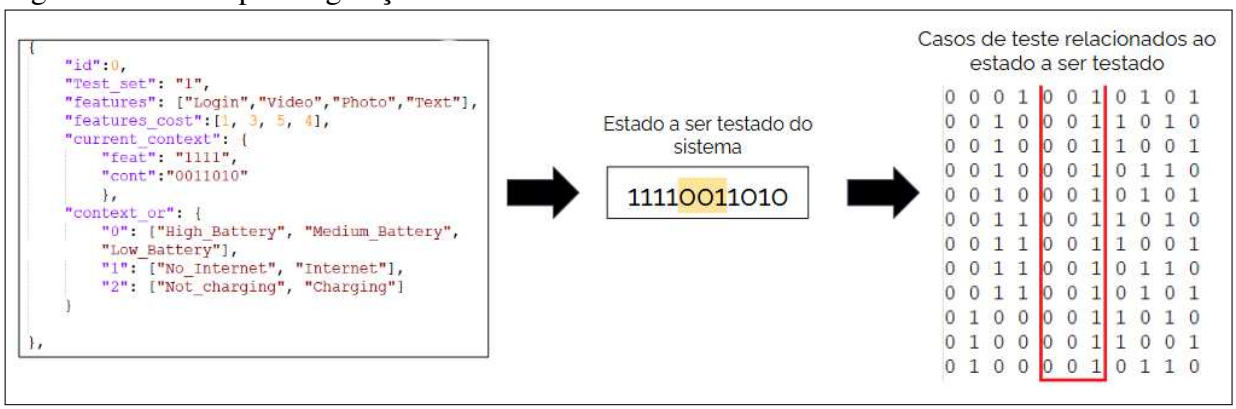


Fonte: elaborada pelo autora.

O analista de teste deve indicar no Bumblebin qual o *Test_set* desejado para conversão binária. Em seguida, o Bumblebin converte as features e os contextos para binário seguindo a mesma definição utilizada para a definição do estado do sistema no arquivo JSON. Após a conversão, são geradas todas as combinações possíveis entre *features* e contextos para a geração dos casos de teste (ver Apêndice A). A partir das combinações, um filtro é realizado selecionando apenas casos de testes relacionados ao estado do sistema (ver Apêndice B). Para selecionar apenas os casos de teste relacionados ao estado do sistema foi definido que apenas casos de teste que tivessem o primeiro grupo de contexto igual ao primeiro grupo de contexto do estado do sistema seriam selecionados. Essa decisão foi feita uma vez que as *features* podem ser ativadas e

desativadas mediante mudança de contexto (SANTOS *et al.*, 2016) e devia-se manter relação com o estado do sistema. A Figura 13 ilustra a um exemplo da geração dos casos de testes relacionados ao primeiro grupo de contexto do estado do sistema.

Figura 13 – Exemplo de geração de casos de teste relacionados



Fonte: elaborada pelo autora.

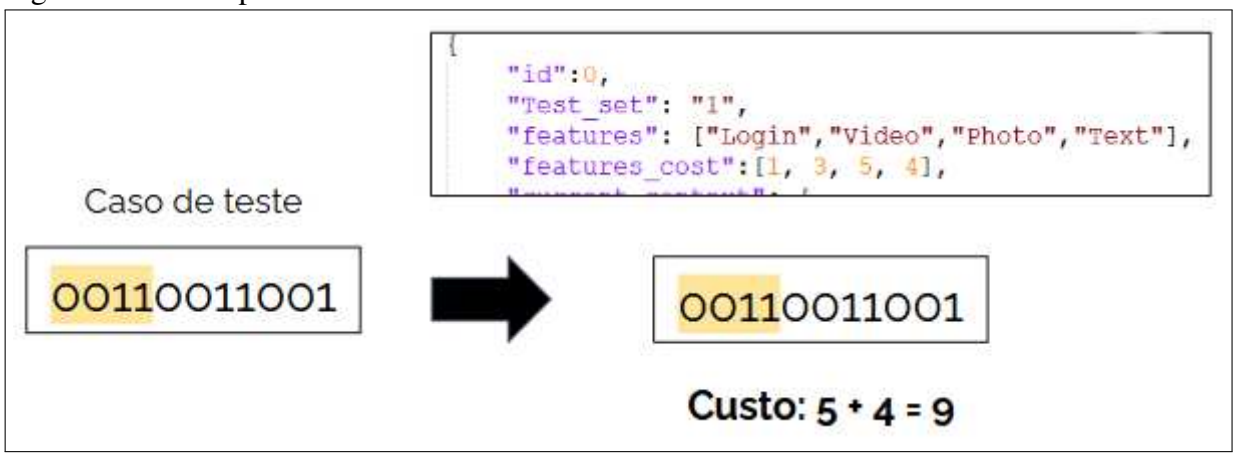
No exemplo da Figura 13, os 4 primeiros dígitos dos casos de teste são relacionados as *features* ["Login", "Video", "Photo" e "Text"] e o restante aos grupos de contexto, sendo o primeiro grupo de contexto igual para todos os casos para manter a relação com o estado do sistema. O primeiro grupo de contexto do estado selecionado a ser testado é 001 ["High_Battery", "Medium_Battery", "Low_Battery"], logo os casos listados todos possuem o primeiro grupo igual a 001 como ilustrado na imagem.

Por fim, o cálculo dos custos é feito por meio da leitura dos casos de teste. O Bumblebin verifica quais *features* estão ativas no caso de teste e realiza o somatório dos custos quando mais de uma *feature* está ativa e se estiver somente uma ativa o custo do caso de teste é o custo da *feature*. Considerando o caso de teste: 00110011001, sendo os 4 primeiros dígitos relacionados as *features* e os custos sendo 1, 3, 5 e 4 (Como na Figura 14 abaixo), o custo desse caso de teste é 9 (0011 = 0+0+5+4), por exemplo.

A Figura 15 apresenta o arquivo de resumo com o Test_set selecionado pelo analista de testes, as *features*, os custos das *features*, os contextos em binário e por fim; as combinações possíveis de *features* e contextos (casos de teste). O arquivo JSON utilizado para esse exemplo é o da Figura 10.

A geração de um arquivo de resumo tem como objetivo fazer com que o analista tenha uma visão geral de quantos casos de teste podem ser gerados a partir das *features* e contextos definidos no arquivo JSON. Em contrapartida, os arquivos com casos de teste filtrados e o de custo serão utilizados no Optimus.

Figura 14 – Exemplo do cálculo do custo do caso de teste



Fonte: elaborada pelo autora.

Figura 15 – Exemplo de arquivo de resumo gerado pelo Bumblebin

```
Test_Set: 1
Features: ['Login', 'Video', 'Photo', 'Text', 'Motion']
Features cost: [1, 3, 2, 4, 5]
Contexts 0:
{'0': ['High_Battery', 'Medium_Battery', 'Low_Battery'], '1': ['No_Internet', 'Internet'], '2': ['Not_charging', 'Charging']}
Binary contexts 0:
  0: [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
  1: [[1, 0], [0, 1]]
  2: [[1, 0], [0, 1]]
######################################################################
COMBINING CONTEXTS AND FEATURES OF THE 1 TEST SET
000011001010
000011001001
000011000110
000011000101
000010101010
000010101001
000010100110
000010100101
000010011010
000010011001
```

Fonte: elaborada pelo autora.

5.2.3 Implementação do Optimus

O Optimus é um mecanimo para geração de sequência de testes otimizadas em relação a custo e diversidade de estados do sistema. O mecanismo busca maximizar a variabilidade de estados do sistema e minimizar o custo execução, dessa forma selecionados os melhores casos de teste em vista à esses objetivos.

Em relação a implementação do Optimus, inicialmente houve a definição do problema. O problema consiste na seleção de casos de testes para a geração de sequência(s) de testes mais eficaz(es). Para selecionar os casos de teste mais adequados, foram considerados os seguintes objetivos:

- Selecionar um subconjunto de casos de teste com maior diversidade de estados do sistema;
- Selecionar um subconjunto de casos de teste com menor custo de execução.

A variabilidade de *features* e contextos nos casos de teste está diretamente relacionada a cobertura de teste do sistema, uma vez que tratando-se de SAS que utilizam o DFTS as *features* e contextos são o que definem como o sistema funciona. Em relação ao custo, esse objetivo torna-se igualmente importante para o Optimus uma vez que para executar testes em tempo de execução pode causar perturbações no estado do sistema (LAHAMI *et al.*, 2016), então quão menos custoso for o teste para o sistema então menos ações indesejadas poderão ocorrer. Portanto, os dois objetivos tem o mesmo peso, isto é, possuem a mesma importância.

A Diversidade de contexto (Definição 2.3.2) de Santos (2020) foi utilizada para determinar a variabilidade do estado do sistema. Utilizando a Distância de Hamming para calcular a similaridade entre dos casos de teste e buscar aquele com maior diversidade. O caso de teste com maior distância de Hamming possui maior variabilidade. Como parâmetro para cálculo da distância foi utilizado o estado do sistema, podendo também utilizar o primeiro caso de teste do arquivo gerado pelo Bumblebin.

Com base nos objetivos, as funções *fitness* são as seguintes:

- máx | variabilidade de estados |, sendo
  | *variabilidade de estados* | = $\sum_{i=1}^{n-1} DH(ct(fc)_i, ct(fc)_{i+1})n$, *onde*
  - DH(ct(fc)$_i$, ct(fc)$_{i+1}$) é a distância de Hamming de um par de casos de teste ct(fc)$_i$ e ct(fc)$_{i+1}$.
  - n sendo o tamanho do caso de teste.
- min | custo |, sendo
  | *custo* | = $\sum_{i=1}^{n} x_i \cdot c_i$, onde
  - x_i é o valor binário indicando se o caso de teste *i* é selecionado (1) ou não (0).
  - c_i é o custo associado ao caso de teste *i*.

O Algoritmo 1 detalha como é realizado o cálculo da Distância de Hamming. A entrada se trata do estado do sistema (ou o primeiro caso de teste da lista) e um caso de teste da lista. Para cada bit diferente entre as duas entradas é somado o valor da Distância de Hamming entre as duas entradas.

O algoritmo utilizado para implementação do Optimus foi o NSGA-II (DEB *et al.*, 2002) em razão da sua propriedade multiobjetiva e preservação do elitismo (mantém os melhores indivíduos da população pai e filho) e a diversidade de soluções. Além disso, o NSGA-II é usado por 30% dos pesquisadores para seleção multiobjetiva de testes em sistemas

Algoritmo 1: Algoritmo do cálculo da Distância de Hamming

```
Entrada: Estado do sistema
Entrada: Um caso de teste da lista
início
    cs ← Estado do sistema;
    ct ← Caso de teste da lista;
    contador ← 0;
    para cada bit ∈ ct faça
        se bit do cs ≠ bit de ct:
        contador ← contador + 1;
    fim
fim
```

(BAJAJ; SANGWAN, 2019). Dessa forma, o NSGA-II foi escolhido por melhor se adequar as características do problema desta pesquisa que seriam: multiobjetivo, busca por diversidade de soluções e ser um algoritmo amplamente empregado pelos pesquisadores para seleção de testes. Vale salientar que para a definição do algoritmo e objetivos, foi analisada a possibilidade de se utilizar somente um objetivo e assim utilizar um algoritmo mono-objetivo. Contudo, pela característica do problema de possuir objetivos contraditórios entre si, isto é uma solução pode ser boa para um objetivo pode ser ruim para outro objetivo, optou-se por seguir as indicações das pesquisas bibliográficas para a seleção de objetivos e algoritmos (CUI *et al.*, 2017).

Para a parametrização do algoritmo, foi definido o tamanho da população igual a 100 (caso a quantidade de casos de teste relacionados seja menor que 100 o tamanho será igual a essa quantidade), sendo possível configurar outros valores. Essa escolha foi feita uma vez que cerca de 40% dos estudos sobre testes de software baseados em algoritmos genéticos estabeleceram o tamanho da população em 100. Entretanto, a definição desses parâmetros não segue uma regra fixa, pois varia conforme a natureza do problema e os objetivos do analista de testes (BAJAJ; SANGWAN, 2019).

O operador de seleção sendo por torneio, que realiza vários torneios entre os indivíduos selecionados aleatoriamente e usa os vencedores para *crossover*. O operador de mutação consiste no bit flit que permite com que cada bit da representação do indivíduo seja trocado (de 0 para 1 ou de 1 para 0) com uma determinada probabilidade (EIBEN *et al.*, 2003), a probabilidade selecionada foi 0.01. O *crossover* sendo de um único ponto de corte e com taxa de 0.5.

Em seguida, o analista de teste define o tamanho da sequência de teste ou alter-

nativamente, pode-se manter o valor padrão de 10, correspondente a 10% da população total definida no algoritmo. No entanto, não há uma regra fixa para essa definição, pois ela depende das necessidades do analista. Por exemplo, quanto maior o tempo disponível para a execução dos testes, maior pode ser a quantidade de testes selecionados. Da mesma forma, com menos tempo disponível, a quantidade de testes a serem executados tende a ser reduzida.

Por fim, após a definição do tamanho da sequência, procede-se à definição do estado do sistema em binário para cálculo da Distância de Hamming (podendo manter por padrão a seleção do primeiro caso de teste do arquivo) e o Optimus segue o fluxo de execução do NSGA-II, sendo:

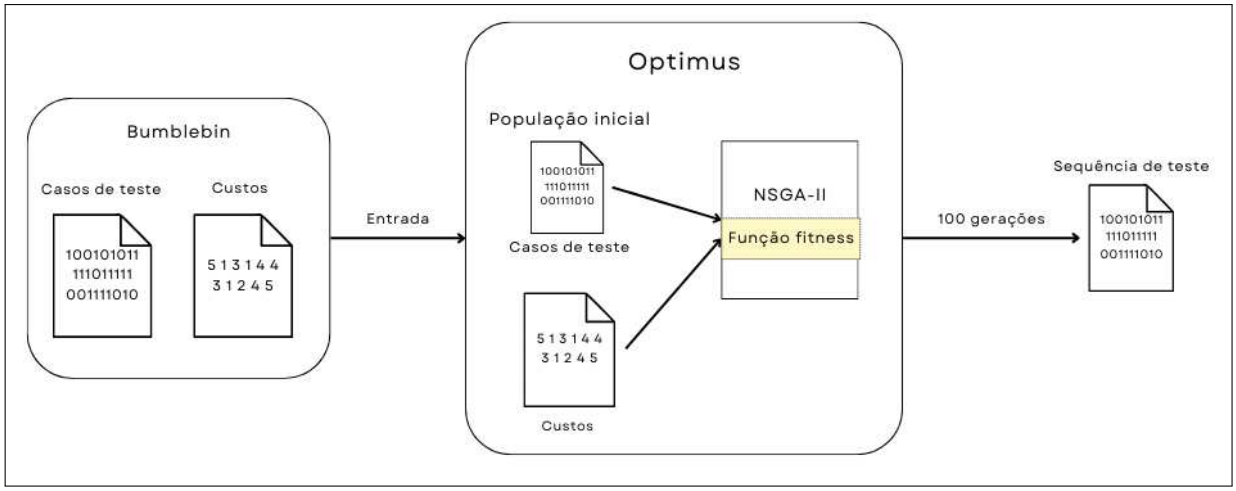
- Inicialização da população: A população inicial são os casos de testes gerados pelo Bumblebin
- Classificação: Processo de classificação com base em critérios de não dominação da população que foi inicializada.
- *Crowding distance*: Nesta etapa ao invés de utilizar o valor da *crowding distance* (que mede o quão longe um indivíduo está do resto da população, isto é, o quão diferente o caso de teste é em relação aos outros) para selecionar os indivíduos, foi utilizada a avaliação do | custo | e | variabilidade de estados | dos indivíduos para selecioná-los. A adaptação foi motivada em razão de que a *crowding distance* possui uma definição bem próxima da | variabilidade de estados |, assim seria uma adição no custo computacional para duas avaliações semelhantes.
- Seleção por torneio: Para selecionar um caso de teste para a próxima geração ou para cruzamento/mutação os casos de são escolhidos aleatoriamente da população e o selecionado é aquele que tem soluções mais dominantes que no caso são aqueles que possuem o menor custo e maior variabilidade de estados
- Repetição do processo: Todas as etapas anteriores são repetidas até que o tamanho da população exceda o tamanho da população atual.

Na etapa de geração de uma nova população, os casos de testes gerados são analisados para verificar se são estados válidos, isto é, se estão dentro da lista de casos de testes definidos anteriormente pelo Bumblebin. Ademais, também retira-se possíveis duplicados na mesma população.

Ao final da execução, o Optimus retorna a sequência de casos de testes ótimos em relação aos objetivos definidos por meio de um arquivo *txt*. Ademais, para garantir que

o mecanismo produzisse os mesmos resultados em execuções diferente, afim de manter a reprodutabilidade foi definida uma *seed*. Ao definir uma *seed* para a geração de números aleatórios, significa que o gerado de números aleatórios inicializa em um estado específico. O gerador então produz uma sequência de números que será sempre a mesma para aquela semente, garantindo que os mesmos "números aleatórios" sejam gerados em execuções subsequentes. Na Figura 16 temos o fluxo de execução do Optimus.

Figura 16 – Fluxo de execução do Optimus



Fonte: elaborada pelo autora.

5.3 Aplicação do Optimus

Para aplicação do Optimus em testes em tempo de execução, na etapa de Especificação do SAS (Detalhada na subseção 5.2.1) deve ser utilizado o estado atual do sistema, sendo o estado atual igual a(s) *feature(s)* e contexto(s) ativados. Embora a execução de testes não esteja incluída no escopo deste trabalho, podem ser utilizados os conceitos de Santos (2020) para testes em tempo de execução em SAS. Santos (2020) define que o mecanismo de adaptação deve ser isolado para execução dos testes, de forma que o mecanismo de adaptação do SUT entre em modo teste através de técnicas de bloqueio e orientação a aspectos. Dessa forma, o analista poderia capturar o estado atual do sistema durante o modo teste do sistema e executar as etapas necessárias (5.2.1, 5.2.2 e 5.2.3) para execução do Optimus e gerar a sequência de teste ótima, assim podendo executar de forma manual ou automatizada os casos de teste. Vale ressaltar, que utilização do Optimus resulta numa diminuição do impacto negativo do bloqueio do mecanismo de adaptação, uma vez que apenas os melhores casos de testes (em relação a custo e diversidade) seriam executados diminuindo o tempo de bloqueio do SUT.

Para aplicação do Optimus em testes em *Design-time*, na etapa de Especificação do SAS deve ser utilizado o estado a ser testado e as outras etapas deve ser seguidas de acordo com as orientações das subseções 5.2.1, 5.2.2 e 5.2.3.

6 AVALIAÇÃO

Este Capítulo apresenta um estudo de viabilidade e três avaliações do Optimus. A Seção 6.1 detalha o estudo de viabilidade, enquanto a Seção 6.2 apresenta as simulações em sistemas de baixa, média e alta complexidade e variabilidade. Por fim, a Seção 6.3 discute os resultados e a validação da hipótese inicial.

6.1 Estudo de viabilidade

Como forma de avaliação do Optimus, um estudo de viabilidade foi realizado com o objetivo de responder a seguinte pergunta: *É viável utilizar o mecanismo para gerar sequências de casos de teste otimizadas?*

Para isso, a aplicação SAS móvel GREat Tour de Marinho *et al.* (2013) foi selecionada para ser especificada. O GREat Tour é um aplicativo de guia turístico de um laboratório de pesquisa e desenvolvimento da Universidade Federal do Ceará, o GREat [1]. Este aplicativo fornece informações sobre o ambiente do laboratório e pesquisadores. A aplicação se adapta seguindo o modelo de *features* de acordo com o contexto atual do visitante, composto pela localização, perfil/preferências e caraterísticas do dispositivo.

As *features* selecionadas para a especificação do SAS foram: Login, Video, Photo e Text;. E os contextos selecionados foram: Bateria alta, Bateria média, Bateria baixa, Sem acesso a internet, Com acesso a internet, Conectado ao carregador e Sem conexão com o carregador (*High_Battery, Medium_Battery, Low_Battery, No_Internet, Internet, Not_charging, Charging*). Dessa forma, são 4 *features* e 7 contextos no total.

A Figura 17 ilustra o resultado da especificação do GREat Tour. Vale notar que, de acordo com a especificação utilizada para a avaliação, o estado de contexto atual do aplicativo é 0011010, significando que estão ativos os contextos *Low_Battery*, *No_Internet* e *Not_charging*.

Após especificação do SAS, utilizando o Bumblebin, foram gerados os casos de teste em binário. Ao total foram obtidos 180 casos de testes, mas somente 60 estavam relacionados ao primeiro grupo de contexto atual referente aos contextos ["High_Battery", "Medium_battery", "Low_Battery"], definidos na etapa de especificação do sistema. Considerando que o estado atual é *Low Battery*, o valor correspondente é (001), indicando que, deste grupo de contexto, apenas o último ("Low_Battery") está ativo. Na Figura 18 é exibido uma parcela dos casos de teste

[1] https://www.great.ufc.br/

Figura 17 – Especificação do GREat Tour

```
{
    "id":0,
    "Test_set": "1",
    "features": ["Login","Video","Photo","Text"],
    "features_cost":[1, 3, 2, 4],
    "current_context": {
        "feat": "1111",
        "cont":"0011010"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"]
    }

},
```

Fonte: elaborada pelo autora.

relacionados ao contexto atual, sendo cada linha do arquivo um caso de teste. Por exemplo, a linha 1 possui o caso de teste 00010011010, que seria somente a *feature* Text ativa e os contextos Low_Battery, No_Internet e Not_charging.

Figura 18 – Parte dos casos de teste gerados pelo Bumblebin para o GREat Tour

1	0	0	0	1	0	0	1	1	0	1	0
2	0	0	0	1	0	0	1	1	0	0	1
3	0	0	0	1	0	0	1	0	1	1	0
4	0	0	0	1	0	0	1	0	1	0	1
5	0	0	1	0	0	0	1	1	0	1	0
6	0	0	1	0	0	0	1	1	0	0	1
7	0	0	1	0	0	0	1	0	1	1	0
8	0	0	1	0	0	0	1	0	1	0	1
9	0	0	1	1	0	0	1	1	0	1	0
10	0	0	1	1	0	0	1	1	0	0	1
11	0	0	1	1	0	0	1	0	1	1	0
12	0	0	1	1	0	0	1	0	1	0	1
13	0	1	0	0	0	0	1	1	0	1	0
14	0	1	0	0	0	0	1	1	0	0	1
15	0	1	0	0	0	0	1	0	1	1	0

Fonte: elaborada pelo autoraa.

Para fins de avaliação foram analisados dois cenários de execução do Optimus: o primeiro cenário utilizando o primeiro caso de teste para cálculo da Distância de Hamming e o segundo cenário utilizando o estado atual do sistema para este cálculo. A motivação para avaliar os dois cenários foi analisar o comportamento do mecanismo em termos de desempenho e qualidade dos casos de teste, modificando o estado inicial para a geração desses casos. Isso se justifica porque, devido à quantidade de casos gerados, a busca por um estado específico pode demandar mais tempo do que simplesmente utilizar o primeiro estado listado.

6.1.1 Cenário 1

Utilizando os arquivos gerados pelo Bumblebin (de custo e com os casos de teste), o Optimus selecionou os 10 melhores casos de teste em relação a custo e diversidade (utilizando como parâmetro de cálculo da distância de Hamming o primeiro caso de teste da lista de casos) em uma média de tempo de 63,38 segundos. A Figura 19 apresenta os casos de teste selecionados pelo mecanismo.

Figura 19 – Casos de teste selecionados pelo Optimus

```
Solução 1: 10000011010
Solução 2: 10000010110
Solução 3: 10000011001
Solução 4: 10000010101
Solução 5: 01000011001
Solução 6: 01000010101
Solução 7: 01000011010
Solução 8: 01000010110
Solução 9: 00010010101
Solução 10: 11000010110
```

Fonte: elaborada pelo autora.

Cada caso de teste selecionado foi avaliado individualmente, a fim de analisar o seu custo individual e sua diversidade de estados. A Tabela 7 apresenta os 10 casos de teste selecionados pelo Optimus.

Tabela 7 – Casos de teste selecionados pelo Optimus

Casos de teste	**Distância de Hamming**	**Custo**
10000011010	2	1
10000010110	4	1
10000011001	4	1
10000010101	6	1
01000011001	4	3
01000010101	6	3
01000011010	2	3
01000010110	4	3
00010010101	4	4
11000010110	5	4

Fonte: elaborada pelo autora.

Ao analisar os dados da Figura 18, nota-se que a Distância de Hamming máxima é 8, uma vez que o primeiro grupo de contexto atual, que são 3 dígitos, estarem sempre no mesmo local no caso de teste. Logo, a Distância de Hamming máxima para estados relacionados a esse

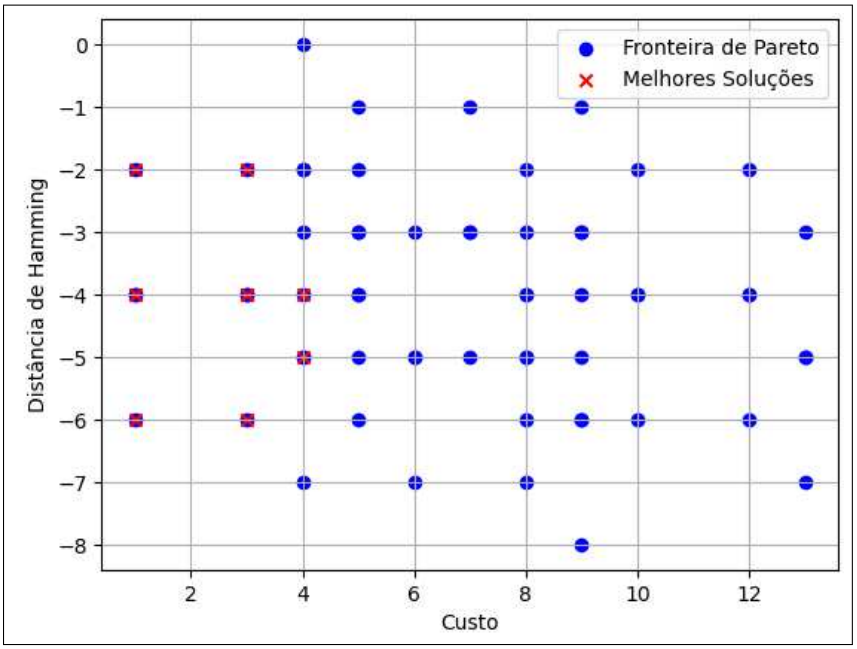
estado atual é de 11 - 3, que resulta em 8. Dessa forma, analisando os dados da Tabela 7 infere-se que o Optimus conseguiu bons resultados balanceando o custo de toda a execução da sequência em relação a variabilidade.

Em relação ao custo (Definição citada na Seção 5.2.1), dentre os casos de teste analisados pelo Optimus existiam casos com os mais diversos custos (1, 3, 4, 5, 9, 7, 8, 9, 10, 12 e 13). O mecanismo conseguiu selecionar os casos de teste com menor custo mantendo uma significativa variabilidade, uma vez que se deve considerar o custo total da sequência.

Ademais, as soluções foram analisadas do ponto de vista de fronteira de Pareto (ver Definições 2.2.2 e 2.2.1). A Figura 20 apresenta a fronteira de Pareto (em azul) e as melhores soluções (em vermelho) em relação a dois objetivos: custo e distância de Hamming. A distância está ilustrada negativa pois visa a maximização. Ao observar o Gráfico, pode-se notar que:

- A distribuição das soluções ao longo da fronteira de Pareto parece ser uniforme, o que é desejável, pois indica uma boa diversidade nas soluções encontradas.
- As melhores soluções estão bastante concentradas em regiões de menor custo. Isso pode indicar que essas soluções são viáveis e dominantes em relação ao custo.
- Observa-se que algumas das "melhores soluções"coincidem com pontos da fronteira de Pareto, o que é positivo, uma vez que essas soluções são não dominadas e, portanto, fazem parte da melhor frente de soluções. Isso significa que as soluções escolhidas são eficientes em termos dos dois objetivos.

Figura 20 – Relação entre fronteira de Pareto e melhores casos de teste



Fonte: elaborada pelo autora.

6.1.2 Cenário 2

As mesmas etapas e dados do Cenário 1 (Seção 6.1.1) foram utilizados para a execução do Cenário 2, tendo como diferença o parâmetro de cálculo da distância de Hamming sendo o estado atual do sistema (11110011010, na Figura 18). A média de execução foi de 88.22 segundos, indicando um aumento no tempo de execução comparado com o Cenário 1.

Assim como no Cenário 1, cada caso de teste selecionado foi avaliado individualmente, afim de analisar o seu custo individual e sua diversidade de estados. Ademais, A Tabela 8 apresenta os 10 casos de teste selecionados pelo Optimus.

Tabela 8 – Casos de teste selecionados selecionados pelo Optimus

Casos de teste	**Distância de Hamming**	**Custo**
10000010101	7	1
10000010110	5	1
10000011001	5	1
10000011010	3	1
01000011001	5	3
01000010101	7	3
01000011010	3	3
01000010110	5	3
00010010101	7	4
11000011001	4	4

Fonte: elaborada pelo autora.

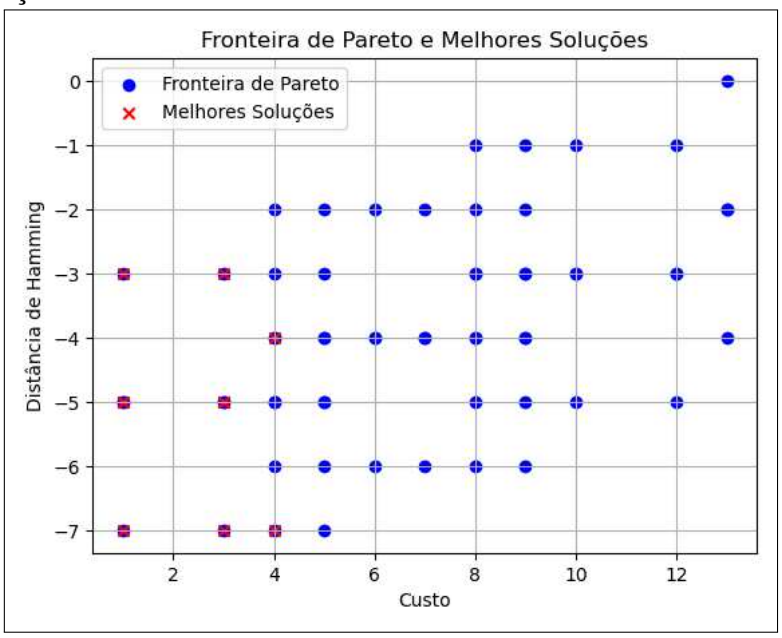
A Distância de Hamming máxima continua sendo a mesma do Cenário 1, de 8. Logo, analisando os dados da Tabela 8 é possível perceber que houve um aumento no valor da distância e o custo se manteve o mesmo. Ademais, mesmo com esse aumento na distância os casos de teste selecionados são em grande parte os mesmos. Somente o caso de teste 11000011001 difere dos selecionados no Cenário 1 para o Cenário 2.

As soluções também foram analisadas do ponto de vista de fronteira de Pareto. A Figura 21 apresenta a fronteira de Pareto (em azul) e as melhores soluções (em vermelho) em relação a dois objetivos: custo e distância de Hamming. Em comparação com o Cenário 1, é possível ver que existem casos de teste com maior variabilidade, mas os resultados relacionados ao custo se mantiveram os mesmos.

6.1.3 Conclusão dos resultados obtidos

A análise das soluções propostas na avaliação de viabilidade do Optimus revela uma boa distribuição ao longo da Fronteira de Pareto, indicando que as estratégias de otimização utili-

Figura 21 – Relação entre fronteira de Pareto e casos de teste



Fonte: elaborada pelo autora.

zadas conseguiram capturar uma bons casos de teste. Ademais, o valor da distância de Hamming obtida é satisfatória, indicando que os casos de teste alcançaram o objetivo de variabilidade de estados e o resultado dos valores de custos também foram satisfatórios.

A análise da sequência de casos de teste gerada: 10000010101, 10000010110, 10000011001, 10000011010, 01000011001, 01000010101, 01000011010, 01000010110, 00010010101 e 11000011001, indica que ela pode ser considerada ótima. Isso se deve ao fato de que a soma das distâncias de Hamming nos cenários 1 e 2 é de 41 e 51, respectivamente. Valores elevados para essas somas indicam maior variabilidade na relação entre o contexto comparado e o selecionado na sequência, o que está alinhado com o objetivo de maximizar a variabilidade. Além disso, o custo associado nos cenários 1 e 2 é de 24, resultando em uma média de custo de 2,4 por caso de teste, o que atende ao objetivo de minimizar o custo.

Ademais ao comparar os Cenários 1 e 2 é possível perceber resultados similares em relação a custo e seleção de testes, mas o desempenho do Cenário 2 foi inferior ao Cenário 1 no que se refere à tempo de execução. Isso possivelmente aconteceu pois no Cenário 2 o Optimus precisa inicialmente encontrar o estado atual e seu custo dentro da lista de casos de casos e apenas depois começar o cálculo da Distância de Hamming.

Com base nos dados coletados nesta avaliação, é possível responder a pergunta "*É viável utilizar o mecanismo para gerar sequências de casos de teste otimizadas?*" de forma afirmativa.

6.2 Simulações

A fim de avaliar o impacto da complexidade estrutural e variabilidade dinâmica dos SAS sobre o Optimus, foram realizadas simulações, com SAS sinteticamente gerados, seguindo os passos de um experimento como definido por Wohlin *et al.* (2012).

Para definir o grau de complexidade e variabilidade dos sistemas foi utilizado o *CatalOg of measures for Feature modEl quality Evaluation* (COfFEE) de Bezerra *et al.* (2014) que define medidas para avaliação de qualidade de modelos de *features*. O catálogo estabelece que quanto menor o valor das medidas de complexidade e variabilidade, menor a complexidade e variabilidade do modelo de *features*. As medidas utilizadas foram: o número de *features* e o número de grupos de contexto, conforme definidos no catálogo COfFEE de Bezerra *et al.* (2014).

Essa geração dos SAS foi feita manualmente, inicialmente supunha-se um *Self-Adaptive Systems* em um domínio de aplicação. Em seguida, foram definidas as *features* com base na aplicação, na complexidade e variabilidade da Simulação. Foram elencadas adaptações em cima destas features, para com isso definir os estados e grupos de contexto. Além disso, foi feita uma revisão para garantir que todos os contextos afetavam todas as *features*, respeitando o formato de modelo de *features* (descrito na Seção 5.2.1).

Nesse sentido, foram definidos 3 cenários distintos, variando a complexidade e variabilidade dos *Self-Adaptive Systems* (SAS). A complexidade e variabilidade do sistema foi diretamente proporcional à quantidade de *features* e contextos que ele abrangia, seguindo a definição do COfFEE. Para cada cenário, o Optimus foi executado 3 vezes para calcular a média de tempo de execução do algoritmo e verificar consistência dos resultados. Assim como no estudo de viabilidade (Seção 6.1), também foram analisados os dois cenários de: cálculo de Hamming com referência ao estado atual e ao primeiro caso de teste da lista (Cenários 1 e 2 respectivamente na Tabela 9). Ademais, foi mantido o valor padrão de 10 casos de teste para a sequência.

Para orientar o experimento a seguinte pergunta foi definida: *Como mecanismo se comporta para os mais complexos e variados SAS?*.

Os resultados obtidos e discutidos neste capítulo apresentam indícios da capacidade do mecanismo de encontrar soluções otimizadas em um tempo significativamente curto, inferior a 7 minutos, mesmo quando aplicado a sistemas complexos e com considerável nível de variabilidade.

Na Seção 6.2.1 são apresentados os resultados da simulação de um sistema de

Tabela 9 – Design da avaliação - Simulações

Simulações	Cenários	Descrição
1	1	Baixa complexidade e variabilidade (5 features e 13 contexts)
2	2	Baixa complexidade e variabilidade (5 features e 13 contexts)
3	1	Média complexidade e variabilidade (9 features e 17 contexts)
4	2	Média complexidade e variabilidade (9 features e 17 contexts)
5	1	Média complexidade e variabilidade (4 features e 17 contexts)
6	2	Média complexidade e variabilidade (4 features e 17 contexts)
7	1	Média complexidade e variabilidade (4 features e 11 contexts)
8	2	Média complexidade e variabilidade (4 features e 11 contexts)
9	1	Média complexidade e variabilidade (5 features e 15 contexts)
10	2	Média complexidade e variabilidade (5 features e 15 contexts)
11	1	Média complexidade e variabilidade (9 features e 19 contexts)
12	2	Média complexidade e variabilidade (8 features e 19 contexts)
13	1	Alta complexidade e variabilidade (10 features e 17 contexts)
14	2	Alta complexidade e variabilidade (10 features e 17 contexts)

complexidade e variabilidade baixa. A Seção 6.2.2 apresenta dois aspectos: um sistema de média complexidade e variabilidade e o mesmo sistema sendo executado fragmentado. Por fim, na Seção 6.2.3 é apresentada uma simulação de estresse, onde o sistema é de alta variabilidade e complexidade.

6.2.1 SAS com Baixa complexidade

6.2.1.1 Simulação 1 - Cenário 1

Para a simulação de um sistema de baixa complexidade e variabilidade foram consideradas 5 *features* e 13 contextos, que são apresentados na Figura 22.

Com o Bumblebin, foram gerados 2.976 casos de teste em binário e 992 casos de teste relacionados ao grupo de contexto atual (100). O tamanho do caso de teste é de 18, sendo 5 bits relacionados as *features* e 13 relacionados aos contextos. O tamanho do caso de teste é relevante pois pode atingir o desempenho do Optimus, especialmente na etapa de *crossover*. Na Figura 23 (A) apresenta uma parte dos casos de teste relacionados ao contexto atual e (B) parte dos custos dos mesmos. A Figura 23 (B) evidencia a variabilidade nos custos dos casos de teste.

O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 0,92 segundos. A Tabela 10 apresenta os casos de teste, as distâncias de Hamming de cada caso de teste e seu custo individual.

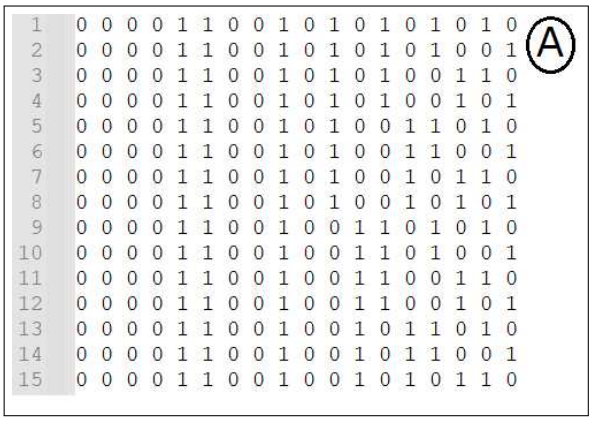
Do ponto vista da Fronteira de Pareto (ver Figura 24), pode-se perceber que o Opti-

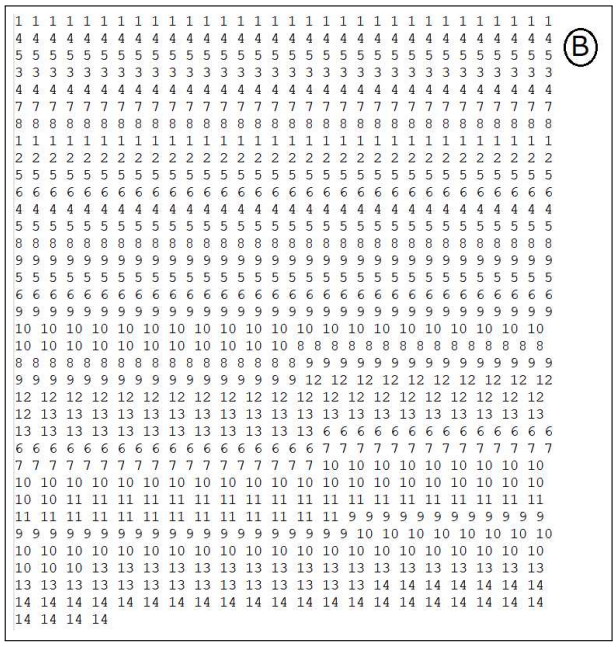
Figura 22 – Especificação do sistema de baixa complexidade e variabilidade


```
{
    "id":0,
    "Test_set": "1",
    "features": ["AR","Pokeballs","Mission","Event",
    "PlayerLevel"],
    "features_cost":[5, 1, 3, 4, 1],
    "current_context": {
        "feat": "01111",
        "cont":"1000110010110"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "3": ["Raining", "Not_Raining"],
        "4": ["Camera_On", "Camera_Off"],
        "5": ["Gyroscope_On", "Gyroscope_Off"]
    }
```


Fonte: elaborada pelo autora.

Figura 23 – (A) parte dos casos de teste relacionados ao contexto atual e (B) custos dos casos de teste





Fonte: elaborada pelo autora.

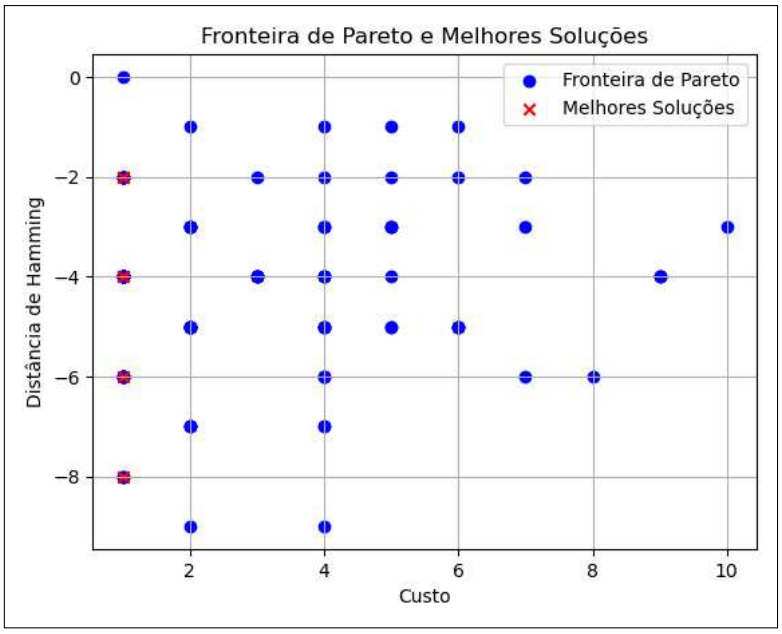
mus foi capaz de encontrar um conjunto diversificado de soluções não dominadas, representando um bom compromisso entre a distância de Hamming e o custo. Ademais, Apesar de ter sido submetido a um conjunto de 992 casos de teste, o mecanismo demonstrou bom desempenho, executando em apenas 0,92 segundos e retornando os bons casos de teste em relação a custo e distância de Hamming.

Tabela 10 – Casos de teste selecionados - Simulação 1

Casos de teste	Distância de Hamming	Custo
0000110010100101 01	6	1
0000110010101010 01	2	1
0000110001101010 01	4	1
0100010001100101 10	8	1
0000110001100110 10	4	1
0000110010011001 10	4	1
0100010001101010 01	6	1
0000110010100110 10	2	1
0000110001010101 10	8	1
0100010010101010 10	2	1

Fonte: elaborada pelo autora.

Figura 24 – Relação entre fronteira de Pareto e casos de teste da Simulação 1 no Cenário 1



Fonte: elaborada pelo autora.

6.2.1.2 Simulação 2 - Cenário 2

As mesmas etapas e dados do Cenário 1 (Seção 6.2.1.1) foram utilizados para a execução do Cenário 2, tendo como diferença o parâmetro de cálculo da distância de Hamming sendo o estado atual do sistema (0111110001100101 10, na Figura 22). A média de execução foi de 1.12 segundos, indicando um aumento no tempo de execução comparado com o Cenário 1.

Assim como no Cenário 1, cada caso de teste selecionado foi avaliado individualmente, afim de analisar o seu custo individual e sua diversidade de estados. Ademais, A Tabela 11 apresenta os 10 casos de teste selecionados pelo Optimus.

A Distância de Hamming máxima é de 15 (18 - 3, sendo 3 dígitos o contexto atual). Logo, analisando os dados da Tabela 11 é possível perceber que houve um aumento no valor da distância e o custo se manteve o mesmo. Ademais, diferente do resultado do estudo de viabilidade

Tabela 11 – Casos de teste selecionados - Simulação 2

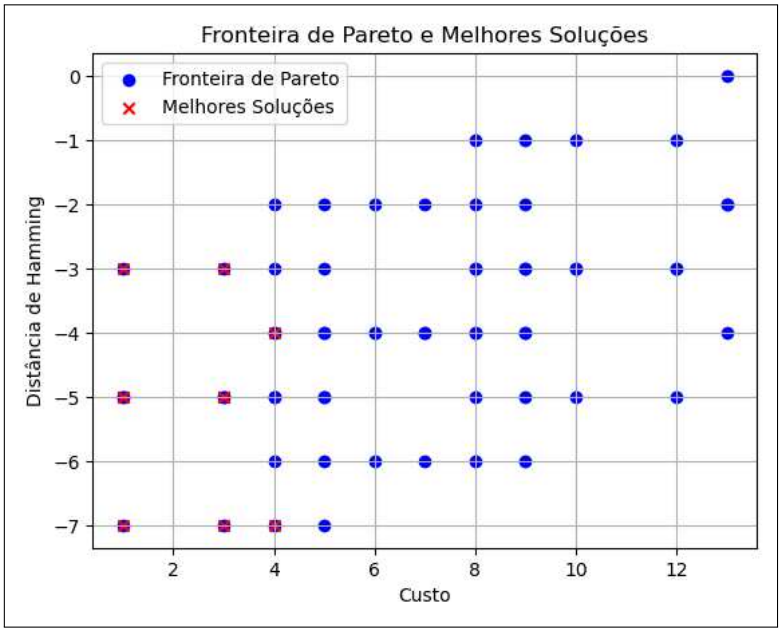
Casos de teste	Distância de Hamming	Custo
010001000110100101	7	1
010001000110101010	7	1
000011000110011001	7	1
000011000110011010	5	1
000011000101010110	5	1
000011000101010101	7	1
010001000110011001	7	1
010001000101010110	5	1
010001001001010110	7	1
000011000110010101	5	1

Fonte: elaborada pelo autora.

apenas 2 casos de testes foram selecionados igualmente nos dois cenários. Esse resultado pode indicar que em sistemas de baixa complexidade e variabilidade, a melhor abordagem seria utilizar o estado atual dependendo do objetivo de teste. Por exemplo, se o objetivo é gerar os casos de teste de forma rápida e alheio ao estado atual do sistema, poderia utilizar o cálculo a partir do primeiro caso de teste da lista.

As soluções também foram analisadas do ponto de vista de fronteira de Pareto. A Figura 25 apresenta a fronteira de Pareto (em azul) e as melhores soluções (em vermelho) em relação a dois objetivos: custo e distância de Hamming. Em comparação com o Cenário 1, é possível ver que existem casos de teste com maior variabilidade, mas os resultados relacionados ao custo se mantiveram os mesmos.

Figura 25 – Relação entre fronteira de Pareto e casos de teste da Simulação 2 no Cenário 2



Fonte: elaborada pelo autora.

6.2.2 SAS com média complexidade e variabilidade

Para o SAS de média complexidade e variabilidade foram executadas 10 simulações para avaliar o impacto que o número de *features* e contextos no desempenho do sistema.

6.2.2.1 Simulação 3 - Cenário 1

Para essa simulação foram consideradas 9 *features* e 17 contextos, que são apresentados na Figura 26.

Figura 26 – Especificação do sistema de média complexidade e variabilidade

```
{
    "id":0,
    "Test_set": "1",
    "features": ["Text","Video","Photo","Push", "Email",
    "Screen", "Buttons", "Vibration", "Sound"],
    "features_cost":[1, 2, 2, 5, 3, 1, 1, 3, 3],
    "current_context": {
        "feat": "111111111",
        "cont":"01001101001101010"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "3": ["Notification_On", "Notification_Off"],
        "4": ["Vibration_On", "Vibration_Off"],
        "5": ["Newsletter_On", "Newsletter_Off"],
        "6": ["Sound_On", "Sound_Off"],
        "7": ["RecommendationBt_On", "RecommendationBt_Off"
        ]
    }

},
```

Fonte: elaborada pelo autora.

Com o Bumblebin, foram gerados 196.224 casos de teste em binário e 65.408 casos de teste relacionados ao primeiro grupo de contexto do estado atual do sistema (010). O tamanho do caso de teste é de 26, sendo 9 bits relacionados as *features* e 17 relacionados aos contextos. O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 87 segundos. Na Tabela 15 são apresentados os casos de teste selecionados pelo Optimus nesta simulação bem como Distância de Hamming e custo de cada.

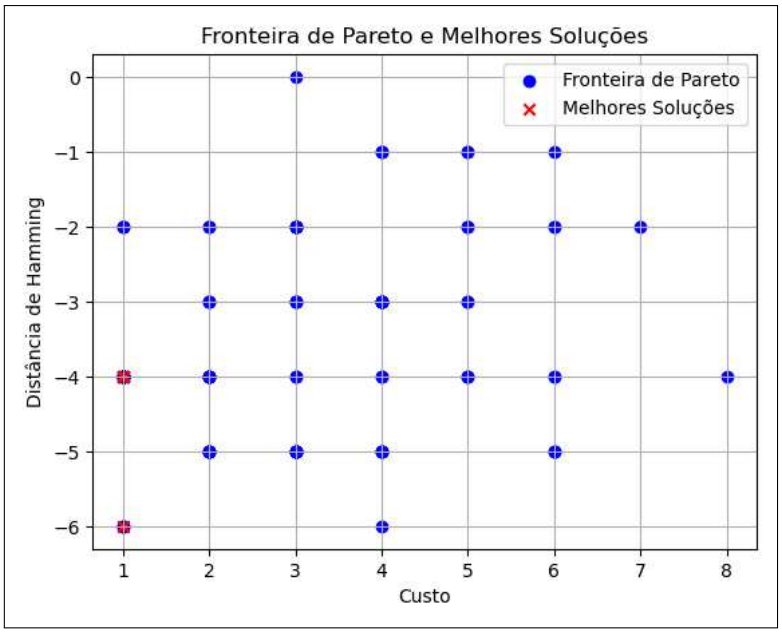
Do ponto vista da Fronteira de Pareto (ver Figura 27), pode-se perceber que o Optimus teve resultados semelhantes ao da simulação 1 no que se refere a soluções não dominadas e relação entre distância de Hamming e custo, mesmo submetido a 65.408 casos de teste.

Tabela 12 – Casos de teste selecionados - Simulação 3

Casos de teste	Distância de Hamming	Custo
10000000001001101010101010	4	1
10000000001010101010100110	4	1
00000010001010101001101010	4	1
00000010001010101001101001	6	1
00000100001010101010101001	4	1
00000010001001101010100110	6	1
00000010001010101010011010	4	1
00000100001010101010100110	4	1
10000000001010101001101010	4	1
00000100001010101001101010	4	1

Fonte: elaborada pelo autora.

Figura 27 – Relação entre fronteira de Pareto e casos de teste da Simulação 3 no Cenário 1



Fonte: elaborada pelo autora.

6.2.2.2 Simulação 4 - Cenário 2

As mesmas etapas e dados do Cenário 1 (Seção 6.2.2.1) foram utilizados para a execução do Cenário 2, tendo como diferença o parâmetro de cálculo da distância de Hamming sendo o estado atual do sistema que é 01001101001101010 (apresentado na Figura 26). O Optimus não conseguiu ser executado, uma vez que pelo número elevado de casos de teste em conjunto com o cálculo de Hamming a partir do estado atual do sistema, afetou o desempenho do mecanismo que solicitou mais poder computacional para executar. Assim, por limitações de ambiente (máquina) não foi possível executá-lo.

6.2.2.3 Simulação 5 - Cenário 1

Nesta simulação foram consideradas 4 *features* e 17 contextos, que são apresentados na Figura 28.

Figura 28 – Especificação do sistema para simulação 5

```
{
    "id":1,
    "Test_set": "2",
    "features": ["Text","Video","Photo","Push"],
    "features_cost":[1, 2, 2, 5],
    "current_context": {
        "feat": "1111",
        "cont":"01001101001101010"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "3": ["Notification_On", "Notification_Off"],
        "4": ["Vibration_On", "Vibration_Off"],
        "5": ["Newsletter_On", "Newsletter_Off"],
        "6": ["Sound_On", "Sound_Off"],
        "7": ["RecommendationBt_On", "RecommendationBt_Off"
        ]
    }

},
```

Fonte: elaborada pelo autora.

Com o Bumblebin, foram gerados 5.760 casos de teste em binário e 1.920 casos de teste relacionados ao primeiro grupo de contexto do estado atual do sistema (010). O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 1.10 segundos. Na Tabela 15 são apresentados os casos de teste selecionados pelo Optimus nesta simulação bem como Distância de Hamming e custo de cada.

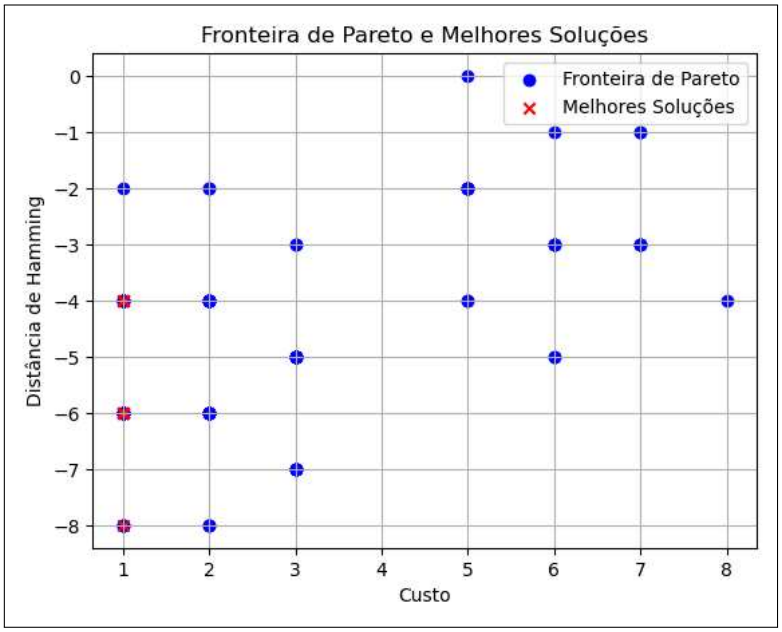
Tabela 13 – Casos de teste selecionados - Simulação 5

Casos de teste	Distância de Hamming	Custo
100001010011010100101	8	1
10000100l101010101001	6	1
10000101010011010101O	4	1
10000101010011010100l	6	1
10000101001l010101001	6	1
10000101001l010101010	4	1
10000100l101010101010	4	1
10000101010101010100l	4	1
10000101001100l101010	6	1
10000101010011010100110	6	1

Fonte: elaborada pelo autora.

Do ponto vista da Fronteira de Pareto (ver Figura 29), pode-se perceber resultados semelhantes a Simulação 3 em relação a soluções e compromisso entre a distância de Hamming e o custo. Ademais, o mecanismo demonstrou bom desempenho, executando em 7.6 segundos sendo submetido a um conjunto de 1920 casos de teste.

Figura 29 – Relação entre fronteira de Pareto e casos de teste da Simulação 5 no Cenário 1



Fonte: elaborada pelo autora.

6.2.2.4 Simulação 6 - Cenário 2

As mesmas etapas e dados do Cenário 1 (Seção 6.2.2.3) foram utilizados para a execução do Cenário 2, tendo como diferença o parâmetro de cálculo da distância de Hamming sendo o estado atual do sistema (11110100l101001101010, na Figura 28). A média de execução foi de 1.33 segundos, indicando um aumento no tempo de execução comparado com o Cenário 1.

Assim como no Cenário 1, cada caso de teste selecionado foi avaliado individualmente, afim de analisar o seu custo individual e sua diversidade de estados. Ademais, a Tabela 14 apresenta os 10 casos de teste selecionados pelo Optimus.

Analisando os dados da Tabela 14 é possível perceber que houve um aumento no valor da distância e o custo se manteve o mesmo. Ademais, diferente do resultado do estudo de viabilidade apenas 1 caso de teste foi selecionado igualmente nos dois cenários.

As soluções também foram analisadas do ponto de vista de fronteira de Pareto. A Figura 25 apresenta a fronteira de Pareto (em azul) e as melhores soluções (em vermelho) em

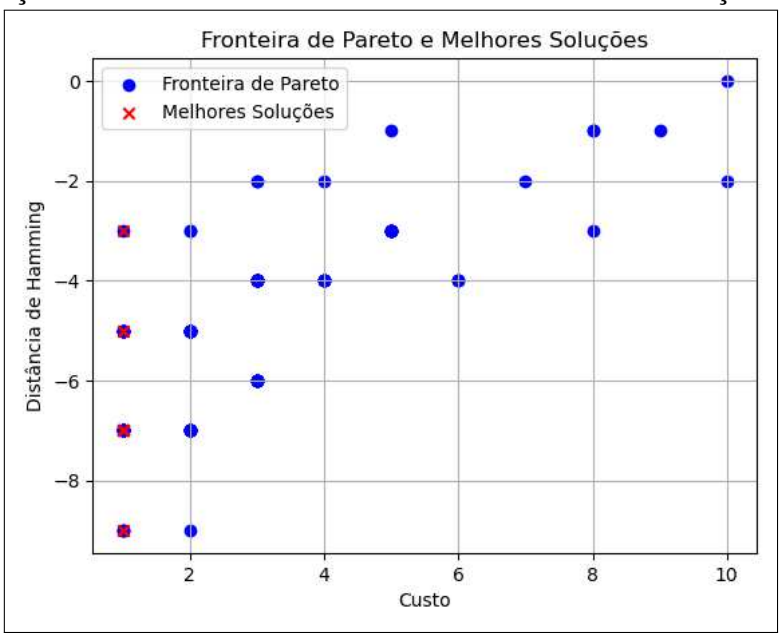
Tabela 14 – Casos de teste selecionados - Simulação 6

Casos de teste	Distância de Hamming	Custo
100001010101010101010	7	1
100001001101001101010	3	1
100001001101001100101	7	1
100001001011001101010	5	1
100001010101010101001	9	1
100001001101001101001	5	1
100001001101010100110	7	1
100001010100101101001	9	1
100001001100101101001	7	1
100001001011001101001	7	1

Fonte: elaborada pelo autora.

relação a dois objetivos: custo e distância de Hamming. Em comparação com o Cenário 1, é possível ver que existem casos de teste com maior variabilidade, mas os resultados relacionados ao custo se mantiveram os mesmos.

Figura 30 – Relação entre fronteira de Pareto e casos de teste da Simulação 6 no Cenário 2



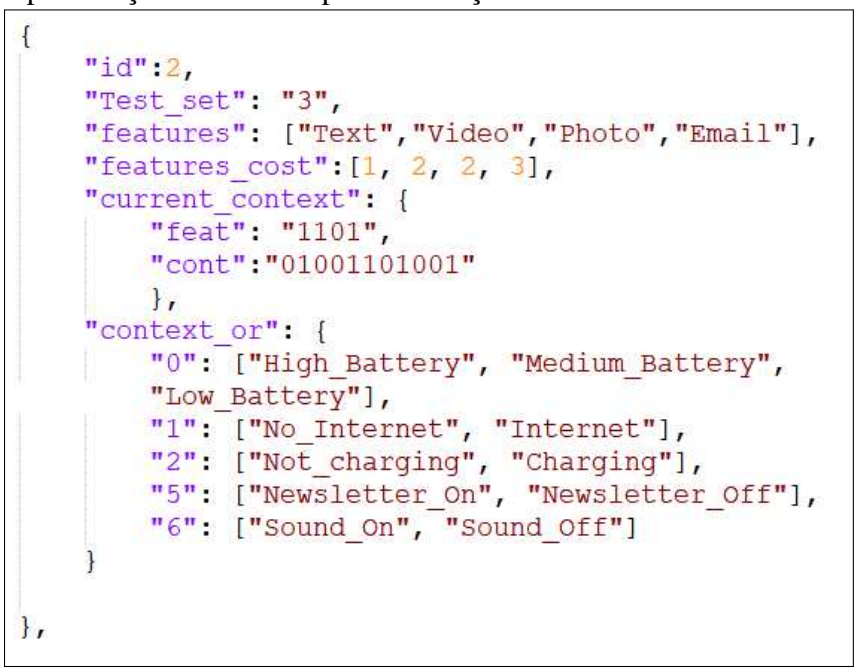
Fonte: elaborada pelo autora.

6.2.2.5 Simulação 7 - Cenário 1

Nesta simulação foram consideradas 4 *features* e 11 contextos, que são apresentados na Figura 31.

Com o Bumblebin, foram gerados 720 casos de teste em binário e 240 casos de teste relacionados ao primeiro grupo de contexto do estado atual do sistema (010). O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 1.22 segundos. Na Tabela

Figura 31 – Especificação do sistema para simulação 7



```
{
    "id":2,
    "Test_set": "3",
    "features": ["Text","Video","Photo","Email"],
    "features_cost":[1, 2, 2, 3],
    "current_context": {
        "feat": "1101",
        "cont":"01001101001"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "5": ["Newsletter_On", "Newsletter_Off"],
        "6": ["Sound_On", "Sound_Off"]
    }

},
```

Fonte: elaborada pelo autora.

15 são apresentados os casos de teste selecionados pelo Optimus nesta simulação bem como Distância de Hamming e custo de cada.

Tabela 15 – Casos de teste selecionados - Simulação 7

Casos de teste	Distância de Hamming	Custo
100001010101010	2	1
100001001100110	6	1
100001001011010	6	1
100001010101001	4	1
100001001101010	4	1
100001010100110	4	1
100001010011010	4	1
100001010011001	6	1
100001010100101	6	1
100001001101001	6	1

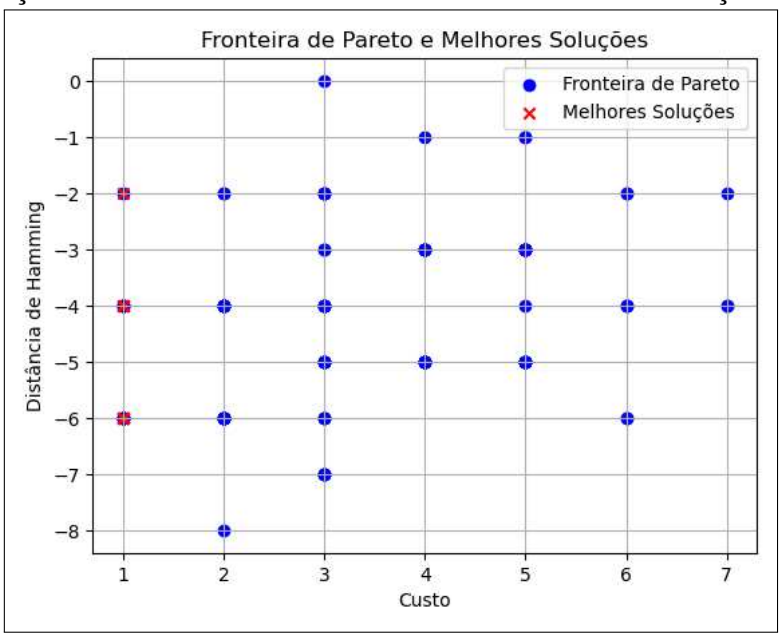
Fonte: elaborada pelo autora.

Do ponto vista da Fronteira de Pareto (ver Figura 32), pode-se perceber resultados semelhantes a Simulação 3 em relação a soluções e compromisso entre a distância de Hamming e o custo.

6.2.2.6 Simulação 8 - Cenário 2

As mesmas etapas e dados do Cenário 1 (Seção 6.2.2.5) foram utilizados para a execução do Cenário 2, tendo como diferença o parâmetro de cálculo da distância de Hamming

Figura 32 – Relação entre fronteira de Pareto e casos de teste da Simulação 7 no Cenário 1



Fonte: elaborada pelo autora.

sendo o estado atual do sistema (110101001101001, na Figura 31). A média de execução foi de 1.42 segundos, indicando um aumento no tempo de execução comparado com o Cenário 1.

Assim como no Cenário 1, cada caso de teste selecionado foi avaliado individualmente, afim de analisar o seu custo individual e sua diversidade de estados. A Tabela 16 apresenta os 10 casos de teste selecionados pelo Optimus.

Tabela 16 – Casos de teste selecionados - Simulação 8

Casos de teste	Distância de Hamming	Custo
100001001011010	6	1
100001001101001	2	1
100001010011001	6	1
100001001101010	4	1
100001001100110	6	1
100001010100101	6	1
100001001100101	4	1
100001010101001	4	1
100001010010101	8	1
100001001010101	6	1

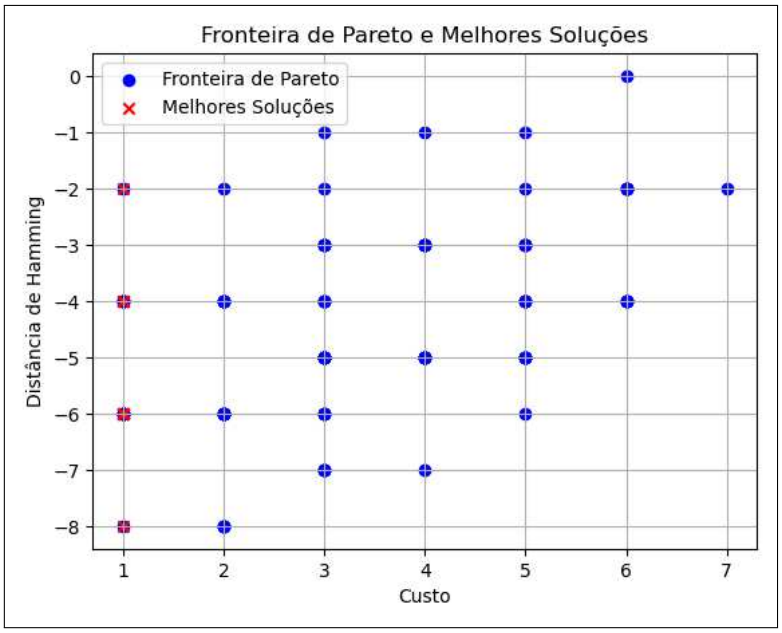
Fonte: elaborada pelo autora.

A Distância de Hamming máxima é de 12. Logo, analisando os dados da Tabela 16 é possível perceber que houve um aumento no valor da distância e o custo se manteve o mesmo. Ademais, diferente do resultado do estudo de viabilidade apenas 7 casos de testes foram selecionados igualmente nos dois cenários.

As soluções também foram analisadas do ponto de vista de fronteira de Pareto. A

Figura 33 apresenta a fronteira de Pareto (em azul) e as melhores soluções (em vermelho) em relação a dois objetivos: custo e distância de Hamming. Em comparação com o Cenário 1, é possível ver que existem casos de teste com maior variabilidade, mas os resultados relacionados ao custo se mantiveram os mesmos.

Figura 33 – Relação entre fronteira de Pareto e casos de teste da Simulação 8 no Cenário 2



Fonte: elaborada pelo autora.

6.2.2.7 Simulação 9 - Cenário 1

Nesta simulação foram consideradas 5 *features* e 15 contextos, que são apresentados na Figura 34. Com o Bumblebin, foram gerados 5.952 casos de teste em binário e 1.984 casos de teste relacionados ao primeiro grupo de contexto do estado atual do sistema (010). O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 1.03 segundos. Na Tabela 17 são apresentados os casos de teste selecionados pelo Optimus nesta simulação bem como Distância de Hamming e custo de cada.

Do ponto vista da Fronteira de Pareto (ver Figura 35), pode-se perceber resultados semelhantes a Simulação 3 em relação a soluções e compromisso entre a distância de Hamming e o custo.

Figura 34 – Especificação do sistema para simulação 9

```
{
    "id":3,
    "Test_set": "4",
    "features": ["Email", "Screen", "Buttons", "Vibration",
     "Sound"],
    "features_cost":[3, 1, 1, 3, 3],
    "current_context": {
        "feat": "11010",
        "cont":"0100110100110 10"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "4": ["Vibration_On", "Vibration_Off"],
        "5": ["Newsletter_On", "Newsletter_Off"],
        "6": ["Sound_On", "Sound_Off"],
        "7": ["RecommendationBt_On", "RecommendationBt_Off"
        ]
    }

},
```

Fonte: elaborada pelo autora.

Tabela 17 – Casos de teste selecionados - Simulação 9

Casos de teste	Distância de Hamming	Custo
00100010100110101001	6	1
00100010101010100110	4	1
00100010100110101010	4	1
01000010101010101010	2	1
01000010011010101010	4	1
01000010101001101001	6	1
01000010100110011010	6	1
00100010101010101010	2	1
01000010011010101001	6	1
01000010101001100110	6	1

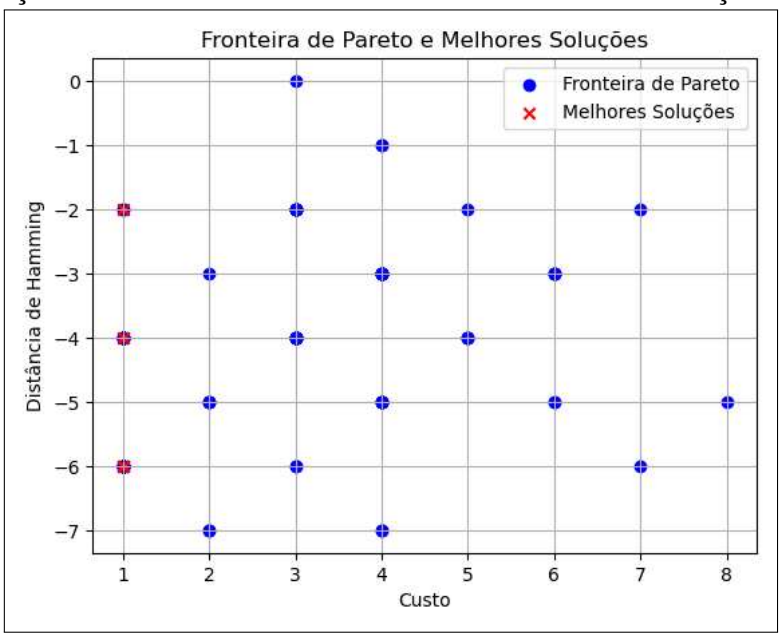
Fonte: elaborada pelo autora.

6.2.2.8 Simulação 10 - Cenário 2

As mesmas etapas e dados do Cenário 1 (Seção 6.2.2.7) foram utilizados para a execução do Cenário 2, tendo como diferença o parâmetro de cálculo da distância de Hamming sendo o estado atual do sistema (11010010011010011010, na Figura 34). A média de execução foi de 1.02 segundos, indicando uma diminuição no tempo de execução comparado com o Cenário 1, isto pode ocorrer por condições ambientais, como por exemplo processamento da máquina.

Assim como no Cenário 1, cada caso de teste selecionado foi avaliado individualmente, afim de analisar o seu custo individual e sua diversidade de estados. A Tabela 18 apresenta os 10 casos de teste selecionados pelo Optimus.

Figura 35 – Relação entre fronteira de Pareto e casos de teste da Simulação 9 no Cenário 1



Fonte: elaborada pelo autora.

Tabela 18 – Casos de teste selecionados - Simulação 10

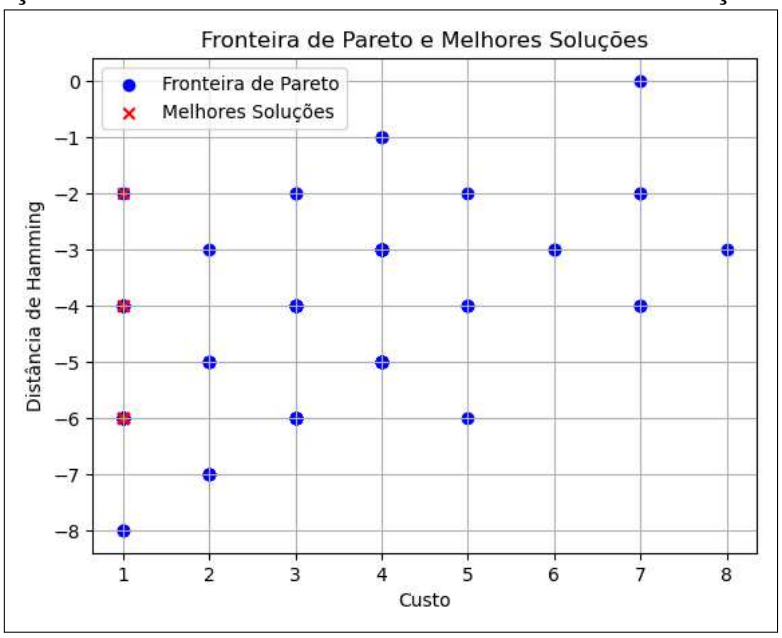
Casos de teste	Distância de Hamming	Custo
01000010010110010110	6	1
01000010010110101010	6	1
01000010101010011001	6	1
01000010011010010110	4	1
01000010011001101010	6	1
01000010011010011010	2	1
01000010011001010110	6	1
01000010010110011010	4	1
01000010010110011001	6	1
01000010011010101010	4	1

Fonte: elaborada pelo autora.

A Distância de Hamming máxima é de 17. Logo, analisando os dados da Tabela 18 é possível perceber que houve um aumento na média de valor da distância (o Cenário 1 teve uma média de 4.6 e no Cenário 2 de 5) e o custo se manteve o mesmo. Ademais, diferente do resultado do estudo de viabilidade apenas 1 caso de teste foram selecionado igualmente nos dois cenários.

As soluções também foram analisadas do ponto de vista de fronteira de Pareto. A Figura 36 apresenta a fronteira de Pareto (em azul) e as melhores soluções (em vermelho) em relação a dois objetivos: custo e distância de Hamming. Em comparação com o Cenário 1, é possível ver que existem casos de teste com maior variabilidade, mas os resultados relacionados ao custo se mantiveram os mesmos.

Figura 36 – Relação entre fronteira de Pareto e casos de teste da Simulação 10 no Cenário 2



Fonte: elaborada pelo autora.

6.2.2.9 Simulação 11 - Cenário 1

Nesta simulação foram consideradas 8 *features* e 19 contextos, que são apresentados na Figura 37.

Figura 37 – Especificação do sistema para simulação 11

```
{
    "id":5,
    "Test_set": "6",
    "features": ["Text","Video","Photo","Push", "Email",
    "Screen", "Buttons","Vibration"],
    "features_cost":[1, 2, 2, 5, 3, 1, 3, 5],
    "current_context": {
        "feat": "11101110",
        "cont":"0100110100110101001"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "3": ["Notification_On", "Notification_Off"],
        "4": ["Vibration_On", "Vibration_Off"],
        "5": ["Newsletter_On", "Newsletter_Off"],
        "6": ["Sound_On", "Sound_Off"],
        "7": ["RecommendationBt_On", "RecommendationBt_Off"
        ],
        "8": ["Test_On", "Test_Off"]
    }

}
```

Fonte: elaborada pelo autora.

Com o Bumblebin, foram gerados 195.840 casos de teste em binário e 65.280 casos

de teste relacionados ao primeiro grupo de contexto do estado atual do sistema (010). O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 85.02 segundos. Na Tabela 19 são apresentados os casos de teste selecionados pelo Optimus nesta simulação bem como Distância de Hamming e custo de cada.

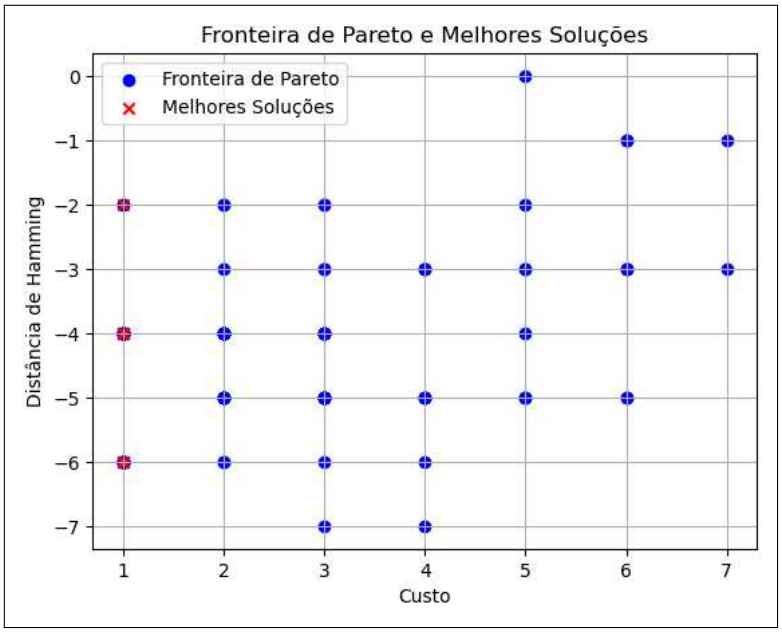
Tabela 19 – Casos de teste selecionados - Simulação 11

Casos de teste	Distância de Hamming	Custo
10000000010101010101101010	4	1
00000100010101010101010101	4	1
10000000010100110101010100110	6	1
00000100010101001101010100110	6	1
00000100010101010101010101010	2	1
00000100010101001101010101010	4	1
10000000010011010011010101010	6	1
10000000010011010101010100110	6	1
00000100010011010101010101010	4	1
00000100010011010101010101010	4	1

Fonte: elaborada pelo autora.

Do ponto vista da Fronteira de Pareto (ver Figura 38), pode-se perceber resultados semelhantes a Simulação 3 em relação a soluções e compromisso entre a distância de Hamming e o custo.

Figura 38 – Relação entre fronteira de Pareto e casos de teste da Simulação 11 no Cenário 1



Fonte: elaborada pelo autora.

6.2.2.10 Simulação 12 - Cenário 2

O Optimus não conseguiu ser executado pelas motivações já citadas (ver Seção 6.2.2.2).

6.2.3 SAS com grande complexidade e variabilidade

6.2.3.1 Simulação 13 - Cenário 1

Para a simulação de um sistema de grande complexidade e variabilidade foram consideradas 10 *features* e 17 contextos, que são apresentados na Figura 39.

Figura 39 – Especificação do sistema de grande complexidade e variabilidade

```
{
    "id":0,
    "Test_set": "1",
    "features": ["Text","Video","Photo","Push", "Email",
    "Screen", "Buttons", "Vibration", "Sound", "Feature1"],
    "features_cost":[1, 2, 2, 5, 3, 1, 1, 3, 3, 5],
    "current_context": {
        "feat": "1111111110",
        "cont":"01001101001101010"
        },
    "context_or": {
        "0": ["High_Battery", "Medium_Battery",
        "Low_Battery"],
        "1": ["No_Internet", "Internet"],
        "2": ["Not_charging", "Charging"],
        "3": ["Notification_On", "Notification_Off"],
        "4": ["Vibration_On", "Vibration_Off"],
        "5": ["Newsletter_On", "Newsletter_Off"],
        "6": ["Sound_On", "Sound_Off"],
        "7": ["RecommendationBt_On", "RecommendationBt_Off"
        ]
    }

},
```

Fonte: elaborada pelo autora.

Com o Bumblebin, foram gerados 392.832 casos de teste em binário e 130.944 casos de teste relacionados ao primeiro grupo de contexto do estado atual do sistema (010). O tamanho do caso de teste é de 27, sendo 5 bits relacionados as *features* e 17 relacionados aos contextos.

O Optimus retornou os 10 melhores casos de teste em uma média de tempo de 330,4 segundos. Na Tabela 20 são apresentados os casos de teste selecionados pelo Optimus nesta simulação bem como Distância de Hamming e custo de cada.

Assim como nas outras simulações, a Fronteira de Pareto foi analisada (ver Figura 40). Apesar de ter sido submetido a um conjunto de 130.944 casos de teste, o mecanismo

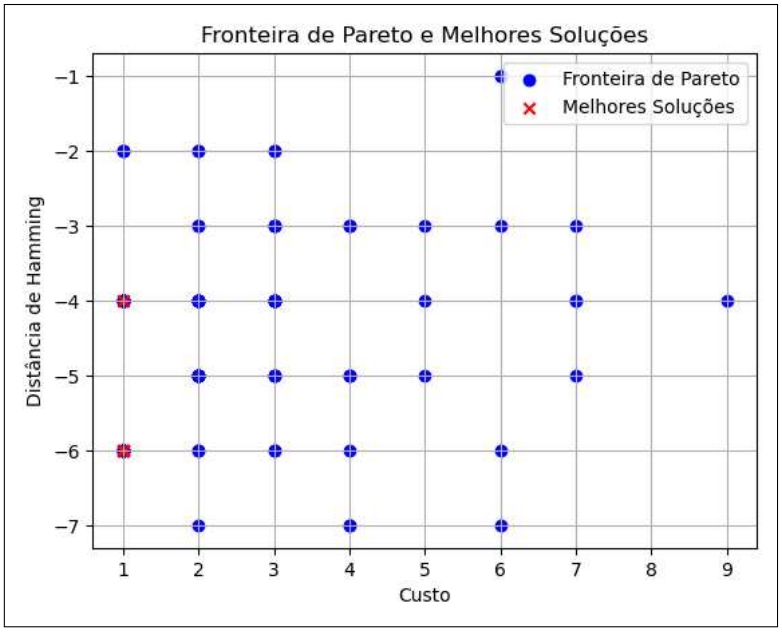
Tabela 20 – Casos de teste selecionados - Simulação 13

Casos de teste	Distância de Hamming	Custo
00000010000101001101010101 0	4	1
10000000000101010101001 1001	6	1
10000000000100110011010101 0	6	1
00000100000101001100110101 0	6	1
00000100000101001101001101 0	6	1
00000100000100110100110101 0	6	1
00000100000101010011010101 0	4	1
00000100000101010100110101 0	4	1
00000100000101010100110100 1	6	1
00000100000101010100110100 1	6	1

Fonte: elaborada pelo autora.

demonstrou bom desempenho, executando em 330,4 segundos e retornando bons casos de teste em relação a distância de Hamming e custo.

Figura 40 – Relação entre fronteira de Pareto e casos de teste da Simulação 13 no Cenário 1



Fonte: elaborada pelo autora.

6.2.3.2 Simulação 14 - Cenário 2

O Optimus não conseguiu ser executado pelas motivações já citadas (ver Seção 6.2.2.2).

6.3 Conclusão da avaliação

Considerando os resultados apresentados, pode-se inferir que o mecanismo Optimus demonstrou bons resultados na geração de casos de teste, apresentando as seguintes características:

- **Eficiência:** O mecanismo foi capaz de gerar os 10 melhores casos de teste em um tempo relativamente hábil, mesmo lidando com um conjunto de 130.944 casos de teste;
- **Qualidade:** Os casos de teste gerados pelo Optimus apresentaram um bom compromisso entre a distância de Hamming e o custo, indicando a capacidade do mecanismo de encontrar soluções diversificadas e não dominadas;

Ademais, pode-se perceber que o parâmetro de cálculo da Distância de Hamming pode afetar a seleção dos casos de teste, mas ao utilizar o estado atual do sistema o mecanismo perde parte do seu desempenho. Também se observou que a divisão do processo de geração de casos de teste em etapas menores demonstra ser uma estratégia eficaz para garantir a escalabilidade. Ao fragmentar a geração, é possível lidar com grandes conjuntos de dados e complexidades crescentes, evitando sobrecarregar os recursos computacionais.

Em resumo, os resultados obtidos (Tabela 21) indicam que o mecanismo Optimus é uma ferramenta promissora para a geração de sequência de casos de teste, oferecendo um bom equilíbrio entre eficiência, qualidade e escalabilidade. Dessa forma, respondendo a pergunta (*Como mecanismo se comporta para os mais complexos SAS?*).

Tabela 21 – Resumo dos resultados da avaliação

Simulações	Qtd. de casos de testes	Tempo	Grau de complexidade e variabilidade
1	992	0.92 segundos	Baixo
2	992	1.12 segundos	Baixo
3	65.408	87 segundos	Médio
4	65.408	-	Médio
5	1.920	1.10 segundos	Médio
6	1.920	1.33 segundos	Médio
7	240	1.22 segundos	Médio
8	240	1.42 segundos	Médio
9	1.984	1.03 segundos	Médio
10	1.984	1.02 segundos	Médio
11	65.280	85.02 segundos	Médio
12	65.280	-	Médio
13	130.944	330.4 segundos	Grande
14	65.280	-	Grande

Fonte: elaborada pelo autora.

6.4 Ameaças à validade

As principais ameaças à validade avaliadas são relacionadas a validade interna e externa. A validade interna considera a relação causal entre fatores, avaliando se há influência de variáveis externas não identificadas que possam comprometer os resultados, enquanto as ameaças à validade validade externas avaliam a generalização dos resultados obtidos (WOHLIN *et al.*, 2012).

A implementação do algoritmo NSGA-II apresenta uma ameaça à validade interna deste trabalho. Para assegurar a implementação correta foram realizados testes com pequenas populações para avaliar soluções de diferentes níveis de dominância, análise da mutação e cruzamento para verificar se a soluções estavam dentro das restrições, variação dos tamanhos dos parâmetros (tamanho da população, número de gerações e probabilidades de cruzamento e mutação) e avaliação das soluções em relação a convergência para a frente de Pareto.

Outra ameaça à validade interna é a implementação do Bumblebin. Para assegurar o correto funcionamento do Bumblebin foram realizados testes com diferentes especificações de SAS (níveis de complexidade e variabilidade), verificando se os casos de teste gerados estavam consistentes com as informações fornecidas pelas especificações.

A principal ameaça à validade externa é a elaboração manual dos modelos de *features* dos SAS utilizados para simulação. Para mitigar foi utilizado como base o modelo de *features* do GREat Tour de Marinho *et al.* (2013) e utilizadas as métricas de avaliação de modelos de *features* de Bezerra *et al.* (2014) para a definição dos sistemas das simulações.

Outra ameaça à validade externa é a quantidade limitada de modelos utilizados na avaliação. Essa restrição implica que os resultados obtidos não podem ser generalizados para *Self-Adaptive Systems* de todos os tamanhos. Para mitigar essa limitação, foram realizadas 14 simulações utilizando diferentes configurações, variando a quantidade de features e contextos.

7 CONCLUSÃO

Este capítulo apresenta as nossa considerações finais com uma visão geral do mecanismo proposto nesta dissertação (Seção 7.1), os resultados obtidos pela pesquisa (Seção 7.2), as limitações (Seção 7.3) e os trabalhos futuros (Seção 7.4).

7.1 Visão geral

O crescente uso de dispositivos móveis e a necessidade de que estes funcionem ininterruptamente em qualquer ambiente resulta, consequentemente, em sistemas mais complexos. Uma vez que a indústria de *software* busca se adaptar a esta demanda, sistemas altamente distribuídos são desenvolvidos para integrar os mais diversos dispositivos e fluxos de dados em diferentes contextos. Contudo, desenvolver, configurar e manter esses sistemas é uma tarefa difícil, sujeita a erros e custosa. A autoadaptação surge como uma solução para auxiliar esta tarefa. Um sistema autoadaptativo deve responder às alterações do ambiente cumprindo os seus requisitos em tempo de execução. Assim, um *Self-Adaptive Systems* (SAS) é capaz de automaticamente modificar-se em resposta às mudanças em seu ambiente operacional.

Apesar de suas vantagens, as adaptações em tempo de execução podem levar a falhas, bugs e à degradação do desempenho, uma vez que é desafiador prever e tratar todos os cenários possíveis que podem surgir durante a execução do sistema. Por isso, testar esses sistema não é uma tarefa trivial.

Essa atividade de teste para SAS torna-se complexa devido a: múltiplas adaptações realizadas em tempo de execução, quantidade de cenários a partir da diversidade de adaptações e dificuldade de gerar automáticamente casos de teste em um ambiente dinâmico.

Sendo assim, o teste em tempo de execução surge como uma alternativa promissora para a validação de sistemas dinamicamente adaptativos. Contudo, a ausência de mecanismos eficazes para gerenciar e manter esses testes em tempo de execução representa um desafio significativo. A principal preocupação reside no alto custo computacional associado a essa abordagem.

A partir dessa problemática, este trabalho prôpos o Optimus cujo objetivo é auxiliar na geração de sequências de casos de teste com maior variabilidade de estados do sistema com menor custo. Por sua característica de desacoplamento, o Optimus é capaz de ser utilizado por abordagens de teste que utilizem um modelo de *features* na especificação do SAS a ser testado.

Ademais, a ferramenta Bumblebin foi desenvolvida para auxiliar na geração dos casos de teste e valoração dos custos. Por meio do Bumblebin, foi possível gerar e converter casos de teste em binário através de um arquivo de especificação. Além disso, esta ferramenta auxiliou na definição dos custos associados a cada caso de teste, otimizando a utilização do Optimus.

Para avaliar o mecanismo, foi realizado um estudo de viabilidade com o objetivo de responder a pergunta: "*É viável utilizar o mecanismo para gerar sequências de casos de teste otimizadas?*"Com os resultado foi possível afirmar que o Optimus alcança sequências de casos de teste ótimas em relação a custo e diversidade de estados.

Por fim, para avaliar o impacto da complexidade dos SAS sobre o Optimus foram realizadas simulações com sistemas sinteticamente gerados. Três simulações foram executadas por ordem de complexidade (baixa, média e alta). Os resultados das simulações indicaram que o Optimus consegue manter a eficiência em relação ao tempo de execução, como também a qualidade dos casos de teste selecionados. Adicionalmente, foi possível observar que dividir o processo de geração dos casos de teste demonstra ser uma estratégia eficaz para diminuição de tempo de execução e custo computacional, asism como o caso de teste parâmetro para o cálculo da distância afeta o desempenho do mecanismo.

7.2 Resultados

Os principais resultados desta pesquisa estão listados a seguir:

- **Optimus.** Um mecanismo para geração de sequências de casos de teste para SAS utilizando NSGA-II para minimizar o custo de execução e maximizar a cobertura de teste. O mecanismo recebe casos de teste em binário e seus respectivos custos, e por meio de funções *fitness* seleciona os casos de teste ótimos. Ao final, o mecanismo retorna uma sequência de casos de teste e seus respectivos custos.
- **Bumblebin.** É uma ferramenta de geração binária de casos de teste. Este artefato permitiu que através de um arquivo JSON com especificações do SAS, fossem geradas todas as combinações possíveis entre *features* e contextos para definição dos casos de teste. Adicionalmente, a ferramenta seleciona todos os casos de teste relacionados ao estado atual do sistema e seus respectivos custos.

Além dos resultados da dissertação propriamente dita, citados anteriormente, durante o período de mestrado foram publicados 2 artigos, listados na Tabela 22, diretamente

relacionados ao tema deste trabalho. O artigo (COSTA *et al.*, 2023) apresenta a ideia inicial da dissertação e o artigo (COSTA. *et al.*, 2024) descreve a revisão sistemática da literatura descrita no Capítulo 3.

Tabela 22 – Artigos relacionados ao tema desta pesquisa

Autores	**Trabalho**	**Evento**	**Qualis (2024)**
COSTA, I. N.; ANDRADE, R. M. C. ; SANTOS, I. S.	Optimus: Mecanismo de otimização de execução de testes em sistemas autoadaptativos	XXII Simpósio Brasileiro de Qualidade de Software: XXI Workshop de Teses e Dissertações em Qualidade de Software (2023)	-
COSTA, I. N.; SANTOS, I. S. ; ANDRADE, R. M. C.	Testing on Dynamically Adaptive Systems: Challenges and Trends	26th International Conference on Enterprise Information Systems (2024)	A3

Fonte: elaborada pelo autora.

Outro artigo, relacionado à área de testes, foi publicado durante o período do mestrado, mas não está diretamente ligado ao tema principal desta pesquisa. Este artigo (COSTA *et al.*, 2022) apresenta uma biblioteca para auxiliar na execução do *Monkey Testing*[1] em várias telas de um dispositivo móvel. O trabalho foi apresentado no *17th Iberian Conference on Information Systems and Technologies (CISTI)* em 2022.

7.3 Limitações

Após a finalização deste trabalho, foram identificadas limitações relacionadas às decisões de design do mecanismo e ao próprio escopo da pesquisa. A seguir são apresentadas as principais limitações:

- Execução da sequência de teste. Apesar das simulações evidenciarem que o mecanismo produz casos de teste ótimos em relação a custo e variabilidade, a execução da sequência de teste em um ambiente real poderia fornecer dados relacionados a desempenho e comunicação com uma ferramenta ou abordagem de teste. Embora a execução de testes não estvesse incluída no escopo deste trabalho, a realização de testes poderia contribuir para a identificação de possíveis melhorias no Optimus.
- Modelo de *features* e contextos. Embora o modelo seja adequado para os SAS, a sua utilização limita a utilização do Optimus. Para diminuir o impacto dessa limitação foi apresentado um modelo menos complexo, uma vez que é necessária a especificação do SAS usando este modelo.

[1] O Monkey é um programa que gera fluxos pseudoaleatórios de eventos do usuário em um dispositivo.

- Atribuição de pesos iguais aos objetivos. Apesar do SAS considerar a mesma importância à diversidade de estados e ao custo, a possibilidade de atribuir pesos distintos a esses objetivos permitiria um ajuste mais fino das soluções, possibilitando que o analista de teste priorize aspectos específicos de acordo com suas necessidades
- Criação dos dados sintéticos. Os dados sintéticos foram gerados manualmente. Apesar de utilizar do modelo de *features* e simular sistemas reais, a utilização de sistemas reais possibilitaria uma visualização concreta de como o mecanismo se comportaria no ambiente real.

7.4 Trabalhos futuros

Nesta Seção são apresentadas possibilidades de evolução desta pesquisa e direções para s novos desafios que surgiram.

- **Pesos diferentes para os objetivos.** O Optimus atualmente trata dos dois objetivos com pesos iguais. Um direcionamneto seria adaptar abordagem multiobjetiva para se comportar como uma abordagem de objetivo único combinando as funções objetivo em uma única função de aptidão, atribuindo pesos a cada objetivo de acordo com sua meta como discutido no trabalho de Bajaj e Sangwan (2019). Ao variar os pesos, seria possível ajustar o comportamento do Optimus de acordo com o grau de importância dado aquele objetivo.
- **Utilização de outras metaheurísticas.** A versão atual do mecanismo utiliza a metaheurística NGSA-II. Ao utilizar outras metaheurísticas pode-se permitir diversificar as soluções encontradas e encontrar pontos de melhoria no mecanismo. Ademais, a metaheurística atual poderia ser combinada com uma nova, a fim de tomar proveito das vantagens de cada uma. O trabalho de Ramirez *et al.* (2018) apresenta algumas metaheurísticas e suas aplicações dentro do SBSE.
- **Avaliação com execução dos testes.** O presente trabalho conduziu a avaliação por meio de simulações com o objetivo de verificar a eficácia e viabilidade do mecanismo. Novas avaliações podem ser realizadas para quantificar a cobertura de falhas proporcionada pelas sequências de teste, permitindo avaliar sua qualidade.
- **Geração dos casos de testes de forma implícita.** A versão atual do mecanismo Optimus necessita do Bumblebin para geração dos casos de teste. Pode-se explorar a geração dos casos de teste implícita, através de regras para geração e validação dos casos de teste no

próprio mecanismo. Desse modo, torna-se possível avaliar o impacto na execução e no desempenho do mecanismo por meio de uma metodologia alternativa de geração de casos de teste.

REFERÊNCIAS

ABOWD, G. D.; DEY, A. K.; BROWN, P. J.; DAVIES, N.; SMITH, M.; STEGGLES, P. Towards a better understanding of context and context-awareness. In: **Handheld and Ubiquitous Computing: First International Symposium, HUC'99**. Germany, Berlin Heidelberg: Springer, 1999. p. 304–307.

ALMEIDA, D. R. de; MACHADO, P. D.; ANDRADE, W. L. Context-aware android applications testing. In: **Proceedings of the XXXIV Brazilian Symposium on Software Engineering**. Natal, Brazil: SBC, 2020. p. 283–292.

ALMEIDA, D. R. de; MACHADO, P. D. L.; ANDRADE, W. L. Enviar: Environment data simulator. In: **Proceedings of the XXXIV Brazilian Symposium on Software Engineerung**. Natal, Brazil: SBC, 2020. p. 532–537.

ALVES, V.; SCHNEIDER, D.; BECKER, M.; BENCOMO, N.; GRACE, P. Comparitive study of variability management in software product lines and runtime adaptable systems. **VaMoS**, Sevilha, v. 9, 2009.

ARRIETA, A.; WANG, S.; SAGARDUI, G.; ETXEBERRIA, L. Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. **Journal of Systems and Software**, United States, v. 149, p. 1–34, 2019.

BAJAJ, A.; SANGWAN, O. P. A systematic literature review of test case prioritization using genetic algorithms. **IEEE Access**, United States, v. 7, p. 126355–126375, 2019.

BARBOSA, G.; SOUZA, É. F. de; SANTOS, L. B. R. dos; SILVA, M. da; BALERA, J. M.; VIJAYKUMAR, N. L. A systematic literature review on prioritizing software test cases using markov chains. **Information and Software Technology**, Netherlands, v. 147, p. 106902, 2022.

BERTOLINO, A.; BRAIONE, P.; ANGELIS, G. D.; GAZZOLA, L.; KIFETEW, F.; MARIANI, L.; ORRÙ, M.; PEZZE, M.; PIETRANTUONO, R.; RUSSO, S. *et al.* A survey of field-based testing techniques. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 54, n. 5, p. 1–39, 2021.

BEZERRA, C. I.; ANDRADE, R. M.; MONTEIRO, J. M. S. Measures for quality evaluation of feature models. In: **Software Reuse for Dynamic Systems in the Cloud and Beyond: 14th International Conference on Software Reuse, ICSR 2015**. Miami, FL, USA: Springer, 2014. p. 282–297.

BOURQUE, P.; FAIRLEY, R. E.; SOCIETY, I. C. **Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0**. 3rd. ed. Washington, DC, USA: IEEE Computer Society Press, 2014. ISBN 0769551661.

CHEN, J.; QIN, Y.; WANG, H.; XU, C. Simulated or physical? an empirical study on input validation for context-aware systems in different environments. In: **Proceedings of the 12th Asia-Pacific Symposium on Internetware**. Singapore: ACM, 2020. p. 146–155.

CHEN, J.-C.; QIN, Y.; WANG, H.-Y.; XU, C. Simulation might change your results: a comparison of context-aware system input validation in simulated and physical environments. **Journal of Computer Science and Technology**, Germany, v. 37, n. 1, p. 83–105, 2022.

CHEN, Y.; CHAUDHARI, N.; CHEN, M.-H. Context-aware regression test selection. In: **28th Asia-Pacific Software Engineering Conference (APSEC)**. Taipei, Taiwan: IEEE, 2021. p. 431–440.

COHEN, J. A coefficient of agreement for nominal scales. **Educational and psychological measurement**, Sage Publications, Thousand Oaks, CA, v. 20, n. 1, p. 37–46, 1960.

COMPUTING, A. *et al.* An architectural blueprint for autonomic computing. **IBM White Paper**, Citeseer, Pensilvânia, v. 31, n. 2006, p. 1–6, 2006.

COPELAND, L. **A practitioner's guide to software test design**. Estados Unidos: Artech House, 2004.

COSTA, I.; ANDRADE, R.; SANTOS, I. Optimus: Mecanismo de otimização de execução de testes em sistemas autoadaptativos. In: **Anais Estendidos do XXII Simpósio Brasileiro de Qualidade de Software**. Porto Alegre, RS, Brasil: SBC, 2023. p. 31–36.

COSTA., I.; S. Santos., I.; ANDRADE., R. Testing on dynamically adaptive systems: Challenges and trends. In: INSTICC. **Proceedings of the 26th International Conference on Enterprise Information Systems - Volume 2: ICEIS**. France: SciTePress, 2024. p. 129–140.

COSTA, L. S. da; SANTOS, I. S.; COSTA, I. N.; ANDRADE, R. M. C. X-monkey: a library to extend the monkey testing. In: **17th Iberian Conference on Information Systems and Technologies (CISTI)**. Portugal: IEE Xplore, 2022. p. 1–6.

CUI, Y.; GENG, Z.; ZHU, Q.; HAN, Y. Multi-objective optimization methods and application in energy saving. **Energy**, Elsevier, Amsterdã, v. 125, p. 681–704, 2017.

DADEAU, F.; GROS, J.-P.; KOUCHNARENKO, O. Testing adaptation policies for software components. **Software Quality Journal**, Springer, Germany, v. 28, p. 1347–1378, 2020.

DADEAU, F.; GROS, J.-P.; KOUCHNARENKO, O. Automated generation of initial configurations for testing component systems. In: **Formal Aspects of Component Software: 17th International Conference, FACS 2021**. Virtual Event: Springer, 2021. p. 134–152.

DADEAU, F.; GROS, J.-P.; KOUCHNARENKO, O. Online testing of dynamic reconfigurations wrt adaptation policies. **Automatic Control and Computer Sciences**, Springer, Germany, v. 56, n. 7, p. 606–622, 2022.

DEB, K.; PRATAP, A.; AGARWAL, S.; MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. **IEEE transactions on evolutionary computation**, IEEE, United States, v. 6, n. 2, p. 182–197, 2002.

DEVRIES, B.; FREDERICKS, E. M.; CHENG, B. H. Analysis and monitoring of cyber-physical systems via environmental domain knowledge & modeling. In: **International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. Madrid: IEEE, 2021. p. 11–17.

DOBSON, S.; DENAZIS, S.; FERNÁNDEZ, A.; GAÏTI, D.; GELENBE, E.; MASSACCI, F.; NIXON, P.; SAFFRE, F.; SCHMIDT, N.; ZAMBONELLI, F. A survey of autonomic communications. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, New York, v. 1, n. 2, p. 223–259, 2006.

DORESTE, A. C. D. S.; TRAVASSOS, G. H. Cats: A testing technique to support the specification of test cases for context-aware software systems. In: **Brazilian Symposium on Software Quality (SBQS)**. Curitiba, Brazil: Association for Computing Machinery, 2023.

DORESTE, A. C. de S.; TRAVASSOS, G. H. Towards supporting the specification of context-aware software system test cases. In: **Proceedings of the XXIII Iberoamerican Conference on Software Engineering (CIbSE)**. Curitiba, Brazil: Curran Associates, 2020. p. 356–363.

EIBEN, A. E.; SMITH, J. E.; EIBEN, A.; SMITH, J. Genetic algorithms. **Introduction to Evolutionary Computing**, Springer, Germany, p. 37–69, 2003.

ELECTRICAL, I. of; ENGINEERS, E. **ISO/IEC/IEEE 24765:2010 Systems and Software Engineering – Vocabulary**. United States: Institute of Electrical and Electronics Engineers, 2010.

ELEUTéRIO, J. D. A. S.; RUBIRA, C. M. F. **Technical Report A Comparative Study of Dynamic Software Product Line Solutions for Building Self-Adaptive Systems**. Campinas, SP, Brasil, 2017. Disponível em: https://ic.unicamp.br/~reltech/2017/17-05.pdf. Acesso em: 5 jun. 2025.

FANITABASI, F.; GAERE, E.; POURNARAS, E. A self-integration testbed for decentralized socio-technical systems. **Future Generation Computer Systems**, Elsevier, Netherlands, v. 113, p. 541–555, 2020.

FREDERICKS, E. M.; RAMIREZ, A. J.; CHENG, B. H. Towards run-time testing of dynamic adaptive systems. In: **8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. United States: IEEE, 2013. p. 169–174.

FREITAS, F. G. de; MAIA, C. L. B.; CAMPOS, G. A. L. de; SOUZA, J. T. de. Otimização em teste de software com aplicação de metaheurísticas. **Revista de Sistemas**, Brazil, v. 5, p. 3–13, 2010.

GARGARI, S. K.; KEYVANPOUR, M. R. Comparative analytical survey on sbst challenges from the perspective of the test techniques. **International Journal of Information & Communication Technology Research**, Iran, v. 14, n. 2, 2022.

GAROUSI, V.; FELDERER, M.; KUHRMANN, M.; HERKILOĞLU, K.; ELDH, S. Exploring the industry's challenges in software testing: An empirical study. **Journal of Software: Evolution and Process**, Wiley Online Library, United Kingdom, v. 32, n. 8, p. e2251, 2020.

HARMAN, M. The current state and future of search based software engineering. In: **Future of Software Engineering (FOSE'07)**. United States: IEEE, 2007. p. 342–357.

HARMAN, M.; JONES, B. F. Search-based software engineering. **Information and software Technology**, Elsevier, Netherlands, v. 43, n. 14, p. 833–839, 2001.

HARMAN, M.; MCMINN, P.; SOUZA, J. T. D.; YOO, S. Search based software engineering: Techniques, taxonomy, tutorial. In: **LASER Summer School on Software Engineering**. Germany: Springer, 2008. p. 1–59.

HASS, A. M. **Guide to advanced software testing**. United States: Artech House, 2014.

HEZAVEHI, S. M.; WEYNS, D.; AVGERIOU, P.; CALINESCU, R.; MIRANDOLA, R.; PEREZ-PALACIN, D. Uncertainty in self-adaptive systems: A research community perspective. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, New York, v. 15, n. 4, p. 1–36, 2021.

JAMOVI. **The jamovi project**. 2022. Disponível em: https://www.jamovi.org. Acesso em: 5 jun. 2025.

JORGENSEN, B. D. P. C. **Software Testing: A Craftsman's Approach**. 4. ed. United States: Auerbach Publications, 2021.

KITCHENHAM, B.; SJØBERG, D. I.; BRERETON, O. P.; BUDGEN, D.; DYBÅ, T.; HÖST, M.; PFAHL, D.; RUNESON, P. Can we evaluate the quality of software engineering experiments? In: **Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement**. United States: ACM, 2010. p. 1–8.

KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, P. **Evidence-based software engineering and systematic reviews**. United States: CRC press, 2016.

KRUPITZER, C.; ROTH, F. M.; VANSYCKEL, S.; SCHIELE, G.; BECKER, C. A survey on engineering approaches for self-adaptive systems. **Pervasive and Mobile Computing**, Elsevier, Netherlands, v. 17, p. 184–206, 2015.

LACERDA, E. G. de; CARVALHO, A. D. Introdução aos algoritmos genéticos. **Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais**, Brasil, v. 1, p. 99–148, 1999.

LAHAMI, M.; KRICHEN, M. A survey on runtime testing of dynamically adaptable and distributed systems. **Software Quality Journal**, Springer, Germany, p. 1–39, 2021.

LAHAMI, M.; KRICHEN, M.; JMAIEL, M. Runtime testing framework for improving quality in dynamic service-based systems. In: **Proceedings of the 2013 International Workshop on Quality Assurance for Service-based Applications**. Lugano, Switzerland: ACM, 2013. p. 17–24.

LAHAMI, M.; KRICHEN, M.; JMAÏEL, M. Runtime testing approach of structural adaptations for dynamic and distributed systems. **International Journal of Computer Applications in Technology**, Inderscience Publishers (IEL), United Kingdom, v. 51, n. 4, p. 259–272, 2015.

LAHAMI, M.; KRICHEN, M.; JMAIEL, M. Safe and efficient runtime testing framework applied in dynamic and distributed systems. **Science of Computer Programming**, Elsevier, Netherlands, v. 122, p. 1–28, 2016.

LIGHT, J.; ARUNACHALAN, B. Mobile middleware service architecture for ems application. In: **2006 1st International Conference on Communication Systems Software & Middleware**. New Delhi, India: IEEE, 2006. p. 1–5.

MAIA, C. L. B.; SOUZA, J. T. de; PEREIRA, P. H. Uma proposta de otimização para seleção de casos de testes para automação. In: **Anais do XLV Simpósio Brasileiro de Pesquisa Operacional**. Natal, Brazil: SBPO, 2013.

MANDRIOLI, C.; MAGGIO, M. Testing self-adaptive software with probabilistic guarantees on performance metrics. In: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. Virtual Event USA: ACM, 2020. p. 1002–1014.

MANDRIOLI, C.; MAGGIO, M. Testing self-adaptive software with probabilistic guarantees on performance metrics: Extended and comparative results. **IEEE Transactions on Software Engineering**, New York, v. 48, n. 9, p. 3554–3572, 2022.

MARINHO, F. G.; ANDRADE, R. M.; WERNER, C.; VIANA, W.; MAIA, M. E.; ROCHA, L. S.; TEIXEIRA, E.; FILHO, J. B. F.; DANTAS, V. L.; LIMA, F. *et al.* Mobiline: A nested software product line for the domain of mobile and context-aware applications. **Science of Computer Programming**, Elsevier, Netherlands, v. 78, n. 12, p. 2381–2398, 2013.

MATALONGA, S.; AMALFITANO, D.; DORESTE, A.; FASOLINO, A. R.; TRAVASSOS, G. H. Alternatives for testing of context-aware software systems in non-academic settings: results from a rapid review. **Information and Software Technology**, Elsevier, Netherlands, v. 149, p. 106937, 2022.

MAURIO, J.; WOOD, P.; ZANLONGO, S.; SILBERMANN, J.; SOOKOOR, T.; LORENZO, A.; SLEIGHT, R.; ROGERS, J.; MULLER, D.; ARMIGER, N. *et al.* Agile services and analysis framework for autonomous and autonomic critical infrastructure. **Innovations in Systems and Software Engineering**, Springer, Germany, p. 1–12, 2021.

MCMINN, P. Search-based software testing: Past, present and future. In: **IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**. Berlin, Germany: IEEE, 2011. p. 153–163.

MENDES, E.; WOHLIN, C.; FELIZARDO, K.; KALINOWSKI, M. When to update systematic literature reviews in software engineering. **Journal of Systems and Software**, Elsevier, Netherlands, v. 167, p. 110607, 2020.

MICHAELS, R.; PIPARIA, S.; ADAMO, D.; BRYCE, R. C. Data driven testing for context aware apps. In: **International Conference on Software Engineering and Knowledge Engineering (SEKE)**. Virtual Event USA: KSI Research Inc, 2022. p. 206–211.

MIRZA, A. M.; KHAN, M. N. A.; WAGAN, R. A.; LAGHARI, M. B.; ASHRAF, M.; AKRAM, M.; BILAL, M. Contextdrive: Towards a functional scenario-based testing framework for context-aware applications. **IEEE Access**, IEEE, United States, v. 9, p. 80478–80490, 2021.

MULLER, H.; KIENLE, H. M.; STEGE, U. Autonomic computing now you see it, now you don't. **Software engineering**, Springer, Italy, p. 32–54, 2009.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. New York: John Wiley & Sons, 2013.

NOROUZI, M.; FLEET, D. J.; SALAKHUTDINOV, R. R. Hamming distance metric learning. **Advances in neural information processing systems**, Curran Associates, United States, v. 25, 2012.

PÉREZ, F.; FONT, J.; ARCEGA, L.; CETINA, C. Empowering the human as the fitness function in search-based model-driven engineering. **IEEE Transactions on Software Engineering**, IEEE, United States, 2021.

PIPARIA, S.; ADAMO, D.; BRYCE, R.; DO, H.; BRYANT, B. Combinatorial testing of context aware android applications. In: **16th Conference on Computer Science and Intelligence Systems (FedCSIS)**. Sofia, Bulgaria: IEEE, 2021. p. 17–26.

PRIYA, S. S.; RAJALAKSHMI, B. Testing context aware application and its research challenges. In: **International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN)**. Villupuram, India: IEEE, 2022. p. 1–7.

RAMIREZ, A.; ROMERO, J. R.; SIMONS, C. L. A systematic review of interaction in search-based software engineering. **IEEE Transactions on Software Engineering**, IEEE, United States, v. 45, n. 8, p. 760–781, 2018.

ROCHA, L. S.; ANDRADE, R. M. Towards a formal model to reason about context-aware exception handling. In: **5th International Workshop on Exception Handling (WEH)**. Zurich, Switzerland: IEEE, 2012. p. 27–33.

SADJADI, S. M.; MCKINLEY, P. K. **Tecnical report A survey of adaptive middleware**. United States, 2003. v. 13. Disponível em: https://www.researchgate.net/profile/S-Masoud-Sadjadi/publication/2946567_A_Survey_of_Adaptive_Middleware/links/0912f50f476b2165c0000000/A-Survey-of-Adaptive-Middleware.pdf. Acesso em: 5 jun. 2025.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. **ACM Trans. Auton. Adapt. Syst.**, Association for Computing Machinery, New York, USA, v. 4, n. 2, 2009.

SALLER, K.; LOCHAU, M.; REIMUND, I. Context-aware dspls: model-based runtime adaptation for resource-constrained systems. In: **Proceedings of the 17th International Software Product Line Conference co-located workshops**. Tokyo, Japan: ACM, 2013. p. 106–113.

SANTOS, E. B. d. **RETAkE: Abordagem para teste em tempo de execução de sistemas dinamicamente adaptativos**, 2020. 16 f. Dissertação (Mestrado em Ciência da Computação), Universidade Federal do Ceará, Fortaleza, 2020.

SANTOS, E. B. dos; ANDRADE, R. M.; SANTOS, I. de S. Runtime testing of context-aware variability in adaptive systems. **Information and Software Technology**, Elsevier, Netherlands, v. 131, 2021.

SANTOS, I. d. S. **TestDAS: Testing method for dynamically adaptive systems**, 2017. 183 f. Tese (Doutorado em Ciência da Computação), Universidade Federal do Ceará, Fortaleza, 2017.

SANTOS, I. de S.; JUNIOR, E. C.; ANDRADE, R. M. de C.; NETO, P. d. A. dos S.; ROCHA, L. S.; WERNER, C. M. L.; SOUZA, J. T. de. Optimized feature selection for initial launch in dynamic software product lines. In: **Internacional Conference on Enterprise Information Systems - ICEIS**. Lisboa: SCITEPRESS, 2018. p. 145–156.

SANTOS, I. S.; ROCHA, L. S.; NETO, P. A. S.; ANDRADE, R. M. Model verification of dynamic software product lines. In: **Proceedings of the XXX Brazilian Symposium on Software Engineering**. Maringá, Brazil: ACM, 2016. p. 113–122.

SHAFIEI, Z.; RAFSANJANI, A. J. A test case design method for context aware android applications. In: **25th International Computer Conference, Computer Society of Iran (CSICC)**. Tehran, Iran: IEEE, 2020. p. 1–8.

SHEVTSOV, S.; IFTIKHAR, M. U.; WEYNS, D. Simca vs activforms: comparing control-and architecture-based adaptation on the tas exemplar. In: **Proceedings of the 1st international workshop on control theory for software engineering**. Bergamo, Italy: ACM, 2015. p. 1–8.

SILVA, D. N. A. d. Adaptation oriented test data generation for adaptive systems. In: **15th Iberian Conference on Information Systems and Technologies (CISTI)**. Seville, Spain: IEE, 2020. p. 1–7.

SILVA, P. C. B. da; ALVES, T. S.; BRUNO, E. A. Automação de testes funcionais: testes funcionais automatizados de software. **Revista de Ciências Exatas e Tecnologia**, Brasil, v. 6, n. 6, p. 113–133, 2011.

SILVA, S.; BERTOLINO, A.; PELLICCIONE, P. Self-adaptive testing in the field: are we there yet? In: **Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems**. Pittsburgh, Pennsylvania: ACM, 2022. p. 58–69.

SIMONS, C. L. Whither (away) software engineers in sbse? In: **1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)**. United States: IEEE, 2013. p. 49–50.

SIQUEIRA, B. R.; FERRARI, F. C.; SERIKAWA, M. A.; MENOTTI, R.; CAMARGO, V. V. de. Characterisation of challenges for testing of adaptive systems. In: **Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing**. Maringa, Brazil: ACM, 2016. p. 1–10.

SIQUEIRA, B. R.; FERRARI, F. C.; SOUZA, K. E.; CAMARGO, V. V.; LEMOS, R. de. Testing of adaptive and context-aware systems: approaches and challenges. **Software Testing, Verification and Reliability**, United States, v. 31, n. 7, p. e1772, 2021.

SOMMERVILLE, I. **Engenharia de Software. Edição 10**. Brasil: Pearson Universidades, 2019.

STRAUSS, A.; CORBIN, J. **Basics of qualitative research**. New York: Sage publications, 1990.

USMAN, A.; IBRAHIM, N.; SALIHU, I. A. Tegdroid: Test case generation approach for android apps considering context and gui events. **International Journal on Advanced Science, Engineering and Information Technology**, Int. J. Adv. Sci. Eng. Inf. Technol, Indonesia, v. 10, n. 1, p. 16, 2020.

VELDHUIZEN, D. A. V.; LAMONT, G. B. *et al.* Evolutionary computation and convergence to a pareto front. In: **Late breaking papers at the genetic programming 1998 conference**. USA: Citeseer, 1998. p. 221–228.

WANG, H.; CHAN, W.; TSE, T. Improving the effectiveness of testing pervasive software via context diversity. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM, USA, v. 9, n. 2, p. 1–28, 2014.

WANG, H.; CHAN, W. K. Weaving context sensitivity into test suite construction. In: **IEEE/ACM International Conference on Automated Software Engineering**. New Zealand: IEEE, 2009. p. 610–614.

WANG, L.; LI, S.; LIU, J.; HU, Y.; WU, Q. Design and implementation of a testing platform for ship control: A case study on the optimal switching controller for ship motion. **Advances in Engineering Software**, ScienceDirect, Netherlands, v. 178, p. 103427, 2023.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. Berlin, Heidelberg: Springer Science & Business Media, 2012.

YI, G. K.; BAHAROM, S. B.; DIN, J. Improving the exploration strategy of an automated android gui testing tool based on the q-learning algorithm by selecting potential actions. **Journal of Computer Science**, Science Publications, Australia, v. 18, n. 2, p. 90 – 102, 2022.

YIGITBAS, E. Model-driven engineering and usability evaluation of self-adaptive user interfaces. **ACM SIGWEB Newsletter**, ACM, New York,USA, n. Autumn, p. 1–4, 2020.

YOO, S.; HARMAN, M. Pareto efficient multi-objective test case selection. In: **Proceedings of the international symposium on Software testing and analysis**. United Kingdom: ACM, 2007. p. 140–150.

APÊNDICE A – PSEUDO-CÓGIDO: GERAÇÃO DE CASOS DE TESTE

Algoritmo 2: Geração de Casos de teste: Combinando Features e contextos

Require: Array A (Features) e lista de dicionários L (Contextos)
Ensure: Lista de strings combinadas R

```
1:  R ← lista vazia
2:  if algum elemento de L não é dicionário then
3:      Erro: "Lista inválida"
4:  end if
5:  for cada v em A do
6:      C ← todas as combinações possíveis dos valores de cada dicionário em L
7:      for cada combinação c em C do
8:          s ← string formada por v seguido dos valores de c
9:          Adicione s em R
10:     end for
11: end for
12: return R
```

APÊNDICE B – PSEUDO-CÓGIDO: SELEÇÃO DE TESTES CONFORME O ESTADO DO SISTEMA

Algoritmo 3: Filtrar Casos de teste a partir do primeiro grupo de contexto

Require: Array de strings A (Todos os casos de teste gerados), contexto inicial C, tamanho do contexto inicial P

Ensure: Lista de Casos de teste filtrados pelo Contexto R

```
1: R ← lista vazia
2: for cada s em A do
3:     if P < tamanho de s then
4:         if substring de s começando em P é igual a C then
5:             Adicione s em R
6:         end if
7:     end if
8: end for
9: return R
```