



UNIVERSIDADE
ESTADUAL DE LONDRINA

DENISE FIGUEIREDO DE REZENDE

UMA FERRAMENTA DE TESTE DE CONFORMIDADE
PARA MODELOS DE PILHA

LONDRINA
2024

DENISE FIGUEIREDO DE REZENDE

**UMA FERRAMENTA DE TESTE DE CONFORMIDADE
PARA MODELOS DE PILHA**

Trabalho de Conclusão de Curso apresentado
ao curso de Bacharelado em Ciência da Com-
putação da Universidade Estadual de Lon-
drina para obtenção do título de Bacharel em
Ciência da Computação.

Orientador: Prof. Dr. Adilson Luiz Boni-
fácia

**LONDRINA
2024**

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Rezende, Denise.

Uma Ferramenta de Teste de Conformidade para Modelos de Pilha / Denise Rezende. - Londrina, 2024.
78 f.

Orientador: Adilson Bonifácio.

Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2024.
Inclui bibliografia.

1. Teste de conformidade - TCC. 2. Sistemas Reativos - TCC. 3. Formalismos - TCC. 4. ioco-like - TCC. I. Bonifácio, Adilson. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

DENISE FIGUEIREDO DE REZENDE

**UMA FERRAMENTA DE TESTE DE CONFORMIDADE
PARA MODELOS DE PILHA**

Trabalho de Conclusão de Curso apresentado
ao curso de Bacharelado em Ciência da Com-
putação da Universidade Estadual de Lon-
drina para obtenção do título de Bacharel em
Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Adilson Luiz Bonifácio
Universidade Estadual de Londrina

Prof. Dr. Rodolfo Miranda de Barro
Universidade Estadual de Londrina - UEL

Prof. Dra. Vanessa Matias Leite
Universidade Estadual de Londrina - UEL

Londrina, 23 de setembro de 2024.

Este trabalho é dedicado àqueles que, com coragem, seguem suas aspirações e inspiram outros a encontrarem seu próprio caminho.

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus, que me sustentou com perseverança ao longo desta jornada. A cada passo, Ele me mostrou que a resposta ao amor é amar, e que no trabalho e no estudo, essa entrega encontra seu auge na entrega de si à serviço dos outros.

Ao meu pai, Pedro, minha eterna gratidão por seu apoio inabalável, por acreditar em mim em cada etapa e me inspirar com sua confiança no meu potencial. À minha mãe, Ketty, cujo exemplo de dedicação, serviço e amor é uma luz que me guia constantemente no caminho que Deus traçou para mim. Aos meus irmãos, Daniel, Diana, Susanna, Joel, Beatriz e Djenane, vocês são os presentes mais valiosos que a vida me deu, sempre me acompanhando, mesmo de longe. Agradeço também à família do Daniel, sua esposa e seus seis filhos, meus queridos sobrinhos, todos me acompanhando de longe. Agradeço de coração ao Joel, Paula, Gabriel, Luísa e Isabela, minha família em Londrina. Desde minha mudança até esta conclusão, a presença de vocês tem enriquecido minha vida com alegria e me ajudado a reconhecer a beleza na busca pelo crescimento em todas as áreas.

Às minhas amigas queridas, Mariana, Maria Eduarda e Ana, minha gratidão por cada olhar de incentivo, por cada gesto de carinho. Vocês fizeram dessa jornada um caminho mais leve e pleno de significado, com nossas conversas e apoio.

Aos meus amigos e colegas de curso, agradeço o companheirismo e a força que me deram ao longo do caminho. Vocês foram parte essencial dessa trajetória.

Um agradecimento especial ao professor Adilson, pela dedicação incansável, paciência e apoio durante todo este projeto. Sua orientação foi um pilar fundamental para o desenvolvimento deste trabalho e para o meu crescimento pessoal e acadêmico.

*Na simplicidade do teu trabalho habitual,
nos detalhes monótonos de cada dia, tens
que descobrir o segredo – para tantos
escondido – da grandeza e da novidade: o
Amor.*

(Sulco, n. 489)

REZENDE, D. F.. **Uma Ferramenta de Teste de Conformidade para Modelos de Pilha.** 2024. 77f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2024.

RESUMO

Os avanços tecnológicos têm promovido uma transformação significativa em diversas áreas da sociedade, proporcionando inúmeros benefícios e oportunidades. No entanto, à medida que as tecnologias se tornam cada vez mais integradas às nossas vidas e em muitos setores críticos, torna-se crucial garantir a confiabilidade desses sistemas, bem como a privacidade de informações, até a prevenção de acidentes por falhas de sistemas. Diante desses desafios, é essencial desenvolver metodologias de teste que detectem falhas de software de forma precisa. Uma abordagem promissora é a combinação de testes de conformidade com métodos formais, especialmente para sistemas reativos. Este trabalho tem como objetivo desenvolver e aplicar uma abordagem de verificação de conformidade utilizando modelos IOVPTS para sistemas reativos com memória de pilha, através da relação de conformidade *ioco-like*. O intuito é garantir que uma implementação se comporte conforme especificado, especialmente em sistemas que exigem operações de empilhamento, aumentando a precisão e a confiabilidade do processo de teste.

Palavras-chave: Verificação de conformidade. Sistemas reativos. Testes formais.

REZENDE, D. F.. **A Conformance Testing Tool for Pushdown Model**. 2024. 77p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2024.

ABSTRACT

Technological advancements have transformed the modern society, providing numerous benefits and opportunities. However, as these technologies become increasingly intertwined with our lives and critical sectors, ensuring their reliability has become essential, as well as the privacy of information to prevent accidents caused by failures. In the face of these challenges, it has become necessary to find solutions to guarantee that the software testing process detects faults more accurately. One such approach that has shown promise is conformance testing, combined with formal methods, particularly for reactive systems. This study focuses on enhancing conformance testing techniques to optimize and improve the efficiency of the testing process for these systems. This work aims to develop and apply a conformance verification approach for IOVPTS models, dealing with reactive systems with stack memory and using an *ioco-like* conformance relation. Here we want to guarantee that an implementation behaves as specified, particularly for systems that require stack operations, giving more accuracy and reliability to the testing process.

Keywords: Conformance testing. Reactive systems. Formal testing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Níveis de abstração do sistema. Fonte: Elaborado pela autora.	20
Figura 2 – Um FSM com 3 estados e 4 transições - com entradas e saídas associadas. Fonte: Elaborado pela autora.	24
Figura 3 – Um exemplo de LTS. Fonte: Bonifácio; Moura, 2016 [14].	30
Figura 4 – Funcionamento do ar condicionado Fonte: Elaborado pela autora. . . .	34
Figura 5 – IOLTS do funcionamento de um ar condicionado Fonte: Elaborado pela autora.	35
Figura 6 – A VPTS \mathcal{S}_1 , with $L_c = \{b\}$, $L_r = \{c, t\}$, $L_i = \emptyset$. Fonte: Bonifácio, 2023 [19].	38
Figura 7 – Relação de conformidade Fonte: Elaborado pela autora.	41
Figura 8 – Arquitetura de Teste Fonte: Tretmans, 2008 [1].	43
Figura 9 – A IUT não está em conformidade com a SPEC. Fonte: Elaborado pela autora.	43
Figura 10 – IUT e SPEC tem o mesmo comportamento. Fonte: Elaborado pela autora.	44
Figura 11 – Arquitetura da estrutura da ferramenta. Fonte: Elaborado pela autora.	49
Figura 12 – Upload de arquivos na ferramenta. Fonte: Elaborado pela autora. . .	53
Figura 13 – Arquivos .txt para especificação dos modelos IOVPTs. Fonte: Elaborado pela autora.	54
Figura 14 – Erro devido à ausência de upload de arquivos na ferramenta. Fonte: Elaborado pela autora.	54
Figura 15 – Resultado de não conformidade entre a IUT e SPEC na ferramenta. Fonte: Elaborado pela autora.	55
Figura 16 – Resultado de conformidade entre a IUT e SPEC na ferramenta. Fonte: Elaborado pela autora.	55
Figura 17 – Arquivo .txt com modelo IOVPTS descrito. Fonte: Elaborado pela autora.	57
Figura 18 – iovpts da máquina de venda automática de chá e café. Fonte: Elaborado pela autora.	60
Figura 19 – Complemento do iovpts da máquina de venda automática de chá e café. Fonte: Elaborado pela autora.	60
Figura 20 – specification.txt	65
Figura 21 – Implementações distintas lado a lado. Fonte: Elaborado pela autora. . .	66
Figura 22 – Implementações distintas lado a lado. Fonte: Elaborado pela autora. . .	67
Figura 23 – Execução do estudo de caso com a ferramenta de conformidade. Fonte: Elaborado pela autora.	68
Figura 24 – Execução da implementação 1 do estudo de caso com a ferramenta de conformidade. Fonte: Elaborado pela autora.	69

Figura 25 – Execução da implementação 2 do estudo de caso com a ferramenta de conformidade. Fonte: Elaborado pela autora.	69
Figura 26 – Execução da implementação 3 do estudo de caso com a ferramenta de conformidade. Fonte: Elaborado pela autora.	69
Figura 27 – Execução da implementação 4 do estudo de caso com a ferramenta de conformidade. Fonte: Elaborado pela autora.	69

LISTA DE TABELAS

Tabela 1 – Tabela de Estados e Transições	64
---	----

LISTA DE ABREVIATURAS E SIGLAS

FSM	<i>Finite State Machine</i>
LTS	<i>Labeled Transition System</i>
IOLTS	<i>Input-Output Labeled Transition System</i>
VPTS	<i>Visibly Pushdown Transitions System</i>
IOVPTS	<i>Input-Output Visibly Pushdown Transitions System</i>
SPEC	<i>Specification</i>
IUT	<i>Implementation Under Test</i>
ioco	<i>Input-Output Conformance</i>
TP	<i>Test Purposes</i>
MBT	<i>Model-Based Testing</i>

SUMÁRIO

1	INTRODUÇÃO	15
2	FUNDAMENTAÇÃO	17
2.1	Testes para Sistemas Reativos	17
2.1.1	Sistemas Reativos	17
2.1.2	Teste de Software	19
2.2	Teste Baseado em Modelos	20
2.2.1	Especificações formais ou Criação do Modelo Formal	22
2.2.2	Propósitos de Testes	24
2.2.3	Geração e Execução de Casos de Teste	25
2.2.4	Cobertura de Teste	26
2.2.5	Completude de Conjuntos de Teste	27
2.2.6	Verificação de Conformidade	28
2.3	Modelos Formais	29
2.3.1	LTS	29
2.3.2	IOLTS	32
2.3.3	VPTS	35
2.3.4	IOVPTS	39
2.4	Verificação de Conformidade	40
2.4.1	Relação <i>ioco</i>	44
2.4.2	Relação <i>ioco-like</i>	45
3	FERRAMENTA PARA RELAÇÃO <i>IOCO-LIKE</i>	48
3.1	Desenvolvimento da Ferramenta	48
3.1.1	Estrutura e Implementação	49
3.1.2	Bibliotecas Utilizadas	51
3.1.3	Preparação do Ambiente para Execução da Ferramenta	52
3.2	A Ferramenta Prática	53
3.2.1	Arquivos de Entrada	55
3.2.2	Arquivo de Saída	57
4	UM ESTUDO DE CASO: MÁQUINA DE VENDAS	58
4.1	Descrição do problema	58
4.2	Especificação da Máquina de Venda	59
4.3	Aplicação da Ferramenta	64
4.3.1	Modelagem dos arquivos de entrada	65

4.3.2	Testando Conformidade	68
4.3.3	Análise dos Resultados Gerados	69
4.4	Resultado	72
5	CONCLUSÃO	74
	REFERÊNCIAS	76

1 INTRODUÇÃO

Sistemas reativos são caracterizados pela interação contínua com o ambiente. Em geral, sistemas reativos são críticos, gerando significativo impacto em áreas como de segurança, saúde, e infraestrutura. Por isso, falhas na detecção de erros nesses sistemas podem causar consequências drásticas. O controle de dispositivos e o processamento de informações sensíveis são alguns dos diversos sistemas reativos, onde uma falha pode resultar em graves consequências. Por exemplo, falhas num sistema de controle de tráfego, resultando em acidentes de trânsito; em sistemas que lidam com dados financeiros, resultando na exposição de informações confidenciais. Considerando a importância vital da detecção de falhas em sistemas reativos críticos, fica evidente o desafio de aprimorar as abordagens de testes para tais sistemas.

Uma das atividades cruciais no desenvolvimento de sistemas dessa natureza são os testes, em especial os testes baseados em modelos que usam formalismos adequadas e garantem maior precisão. Na área de teste baseado em modelos, uma das abordagens usadas para sistemas reativos é a verificação de conformidade [2]. O objetivo é garantir que o comportamento de um sistema reativo esteja de acordo com o comportamento de sua respectiva especificação. Uma abordagem para se verificar conformidade em sistemas reativos tradicionais usa a relação clássica *ioco* (Conformidade de Entrada/Saída)¹. Nesse cenário, o formalismo usado para especificar tais sistemas é modelo IOLTS (Sistema de Transição Rotulado de Entrada/Saída)², um sistema de transição rotulado com entradas e saídas, permitindo a análise formal e a verificação de conformidade sobre estes modelos.

No entanto, para capturar cenários mais complexos, um modelo com memória auxiliar foi proposto, os IOVPTSSs (Sistemas de Transição com Pilha Visível de Entrada/Saída)³ [3]. Essa complexidade adicionada pelo uso de uma memória de pilha impacta consideravelmente na complexidade da relação de conformidade adotada e, consequentemente, no processo de verificação realizado entre implementações e suas respectivas especificações.

Dessa forma, este trabalho busca aprimorar os métodos de verificação de conformidade aplicados a sistemas reativos, visando desenvolver uma ferramenta de suporte que viabilize sua aplicação prática. Além disso, a pesquisa culmina na utilização dessas abordagens em um estudo de caso, demonstrando a relevância e a aplicabilidade dos métodos propostos na verificação de sistemas críticos complexos.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 apre-

¹ Do inglês *Input/Output Conformance*.

² Do inglês *Input Output Labeled Transition Systems*.

³ Do inglês *Input/Output-Visible Pushdown Transition Systems*.

senta a fundamentação teórica sobre os sistemas reativos assíncronos e as técnicas de verificação de conformidade. A discussão inclui as características dos sistemas reativos e seus respectivos formalismos adotados na modelagem dos sistemas. O Capítulo 3 apresenta a ferramenta desenvolvida para realizar a verificação de conformidade em sistemas modelados com o formalismo de IOVPTS. Por fim, o Capítulo 5 contém as conclusões deste trabalho, destacando as contribuições realizadas e os desafios enfrentados.

2 FUNDAMENTAÇÃO

Este capítulo se concentra nos fundamentos necessários para a compreensão de sistemas reativos e os diferentes métodos de teste de software, com um foco particular nos testes baseado em modelos.

A Seção 2.1 descreve os sistemas reativos, suas características e os desafios relacionados ao desenvolvimento desses sistemas. Além disso, essa seção discute as técnicas gerais de teste de software, destacando a importância dessas práticas no ciclo de desenvolvimento de sistemas. Diferentes métodos de teste, como caixa-branca, caixa-preta e caixa-cinza, são abordados, cada um com seus respectivos objetivos e níveis de abstração. A Seção 2.2 aborda MBT (Teste Baseado em Modelo)¹, uma técnica que formaliza o processo de teste utilizando modelos como especificações para gerar casos de teste e verificar a conformidade do sistema. Já a Seção 2.3 detalha os modelos formais utilizados nos testes baseado em modelos, com ênfase em diferentes sistemas de transição rotulados. Por fim, a Seção 2.4 apresenta os métodos de verificação de conformidade, discutindo como garantir que o sistema implementado esteja de acordo com o modelo especificado usando relações de conformidade apropriadas.

Este capítulo, portanto, estabelece uma base sólida para o desenvolvimento de técnicas rigorosas aplicáveis à verificação e validação de sistemas reativos complexos.

2.1 Testes para Sistemas Reativos

Este capítulo explora os conceitos sobre sistemas reativos, as dificuldades que tais sistemas exigem no processo de teste, bem como os fundamentos associados a área de teste para sistemas reativos assíncronos.

2.1.1 Sistemas Reativos

Um sistema reativo é um sistema computacional que produz respostas a eventos externos fornecidos como estímulo. Esses estímulos devem ser processados gerando uma reação do sistema, seja realizando uma ação, emitindo uma mensagem ou mesmo atualizando seu estado interno [2, 3]. Tais sistemas funcionam normalmente em ciclos de execução contínuos, mantendo uma interação constante com o ambiente.

Os sistemas reativos se tornaram cada vez mais comuns entre os sistemas computacionais, seja em soluções tecnológicas simples ou em aplicações industriais críticas. Em todos os lugares sistemas do mundo real estão sendo governados por comportamentos

¹ Do inglês *Model Based Testing*.

reativos, onde os sistemas interagem com um ambiente externo recebendo estímulos de entrada e produzindo saídas em resposta. Esses eventos externos podem ser de vários tipos, incluindo eventos físicos, como um toque na tela de um dispositivo, eventos lógicos, como uma solicitação de um usuário, ou eventos de sistema, como um cronômetro que dispara.

Há inúmeros sistemas reativos que também são críticos, onde uma falha pode causar danos significativos às pessoas, à propriedade ou ao ambiente. Estes sistemas podem ser encontrados em uma variedade de setores, incluindo aeronáutica e aeroespacial, em aplicações tais como em sistemas de controle de voo, sistemas de navegação e sistemas de segurança de voo; e na saúde, tais como em sistemas de monitoramento de pacientes, sistemas de suporte à vida e sistemas de diagnóstico.

Alguns desses sistemas reativos críticos também são caracterizados pela necessidade de responder a eventos com restrição de tempo, como por exemplo, os sistemas de controle de tráfego aéreo (ATC). Neste caso, além de críticos e reativos, tais sistemas devem ser capazes de responder rapidamente a mudanças de posição das aeronaves e a outros eventos, como emergências [4]. Nesses sistemas são cruciais as restrições de tempo para respostas, pois a corretude da saída do sistema depende não apenas do resultado correto, mas também no tempo correto. Uma resposta atrasada em um sistema de tempo real pode ter consequências desastrosas, comprometendo a segurança, a funcionalidade ou mesmo o desempenho geral do sistema [5].

Em virtude das consequências drásticas que falhas nesses sistemas podem causar, testes mais rigorosos, usando abordagem formal, são usados para garantir a confiabilidade e segurança desses sistemas. Para que os requisitos de software sejam então atendidos ao longo do processo de desenvolvimento, atividades de verificação e validação (V&V), em geral, são aplicadas.

A etapa de verificação avalia se um software implementa corretamente uma funcionalidade específica [6], isto é, verifica se a funcionalidade está de acordo com a especificação. As técnicas de verificação incluem revisões de código, análise estática e testes unitários, focando na identificação de erros na estrutura, lógica e implementação do código. A verificação é essencial para assegurar que o sistema está sendo construído conforme as especificações definidas. Por outro lado, o processo de validação analisa se o software atende aos requisitos especificados durante a fase de desenvolvimento [6], isto é, se o sistema atende às necessidades do usuário. A validação está associada à implementação do software e envolve atividades de teste, como testes de integração, testes de sistema e testes de aceitação. Esses testes verificam se os requisitos funcionais levantados estão presentes no sistema.

O teste de software é uma ferramenta essencial nas atividades de V&V. Técnicas e métodos de geração e execução de testes são desenvolvidos com o intuito de identificar

falhas e inconsistências, tanto em código quanto em especificações. Os resultados dos testes fornecem informações valiosas para aprimorar o processo de desenvolvimento e garantir a qualidade final do software.

2.1.2 Teste de Software

O teste de software é uma das principais etapas do processo de desenvolvimento de sistemas. Ao longo de décadas o teste de software tem sido aplicado para garantir qualidade no desenvolvimento de sistemas, e inúmeras pesquisas têm sido realizadas com intuito de aumentar a eficiência e a eficácia dos testes para lidar com sistemas cada vez mais complexos [7]. Dessa forma, para garantir a qualidade do software, os testes têm como objetivo identificar falhas e evitar problemas para os usuários.

Existem diversos tipos de testes, cada um com objetivos específicos. Os testes de estresse, por exemplo, são projetados para avaliar como um sistema se comporta sob condições extremas - sob grandes volumes de dados. Já os testes de velocidade e performance medem a eficiência e o tempo de resposta do sistema, identificando possíveis gargalos e áreas para otimização. Além disso, testes de segurança focam na identificação de vulnerabilidades e na proteção contra ameaças externas. Cada tipo de teste desempenha um papel crucial na garantia da qualidade e robustez dos sistemas de software [8].

De acordo com cada objetivo existem diferentes testes. O teste unitário, por exemplo, contempla o teste de uma única classe ou método em uma implementação. Já o teste de integração garante que os componentes funcionem corretamente em conjunto, enquanto o teste de sistema engloba o teste do sistema desenvolvido como um todo [9].

Outros tipos de teste também podem ser aplicados, tais como o teste de robustez, que busca falhas sob condições adversas tais como entradas inesperadas, indisponibilidade de recursos ou falhas de *hardware* e rede, o teste de performance, que valida o sistema sob grande quantidade de requisições, e o teste de usabilidade, que detecta problemas de interface que dificultam a interação do usuário com o sistema [9].

Cada tipo de teste utiliza diferentes níveis de abstração do sistema em análise.

1. Testes de caixa-branca concentram-se na análise do funcionamento interno do sistema, com base no conhecimento da sua estrutura. Esses testes buscam identificar falhas estruturais de baixo nível, permitindo uma avaliação detalhada da lógica, estruturas de dados e algoritmos. Essa abordagem possibilita a detecção precisa de problemas internos que podem não ser evidentes em testes mais superficiais.
2. Testes de caixa-preta focam no comportamento e nas funcionalidades do sistema sem considerar sua implementação ou estrutura interna. O sistema é avaliado apenas por meio da sua interface com o ambiente externo, e a validação é realizada com

base nas respostas do sistema a diferentes estímulos externos [1]. Essa técnica é particularmente eficaz para verificar se o sistema atende aos requisitos funcionais e comportamentais esperados, independentemente de como foi implementado.

3. Testes de caixa-cinza combinam elementos dos testes de caixa-branca e caixa-preta [8]. Nesse caso, a funcionalidade do sistema é analisada tanto a partir da especificação funcional quanto da estrutura lógica interna, como o código fonte. Isso possibilita uma avaliação abrangente que considera tanto as funcionalidades externas quanto detalhes da implementação interna. Quando há acesso a alguma informação interna do sistema, por exemplo, como interfaces públicas de módulos ou documentação específica, mas não ao código-fonte completo, é possível explorar o sistema além dos testes de caixa-preta, utilizando um conhecimento parcial da implementação.

A Figura 1 ilustra de forma concisa os diversos níveis de abstração de um sistema.



Figura 1 – Níveis de abstração do sistema.

Fonte: Elaborado pela autora.

2.2 Teste Baseado em Modelos

O Teste Baseado em Modelos (MBT)² se destaca como uma abordagem rigorosa e ideal para enfrentar os desafios do teste de *software* em sistemas críticos e reativos [1]. Dois fatores importantes para sistemas críticos reativos são a garantir a confiabilidade e segurança no desenvolvimento. Nesse contexto, aplicamos testes de caixa-preta e caixa-cinza conforme a situação, sendo que o teste de caixa-preta é mais comum em sistemas dessa natureza [10]. Entre as diversas técnicas de teste de caixa-preta, que não exigem conhecimento do código-fonte, destaca-se o MBT. Esta abordagem oferece rigor e formalismo,

² Do inglês *Model-Based Testing*.

fornecendo uma metodologia sistemática para identificar falhas e garantir o comportamento correto desses sistemas. O MBT é particularmente eficaz em sistemas críticos, pois permite uma validação abrangente e detalhada das especificações e dos comportamentos esperados, contribuindo assim para a alta confiabilidade e segurança desses sistemas.

No MBT, o comportamento desejado do sistema é definido por meio de um modelo formal, que serve tanto como uma especificação quanto como uma base para criar e executar testes para implementações candidatas [1]. Utilizando modelos formais para descrever esses comportamentos desejados, o MBT garante maior rigor no processo de teste e tenta evitar as inconsistências e ambiguidades nos sistemas.

Através das técnicas de MBT conjuntos de teste podem ser gerados com base na especificação para então serem aplicados a uma implementação. Uma IUT (Implementação Sob Teste)³, pode ser um *hardware* ou *software*, um sistema embutido ou um sistema com sensores e atuadores [1]. Já uma especificação corresponde ao modelo formal de abstração que caracteriza e descreve os comportamentos desejáveis/indesejáveis de uma implementação [11]. A especificação pode representar uma parte de um sistema, com detalhes suficientes para descrever precisamente as características a serem testadas [10].

Uma forma de garantir que uma implementação atenda aos requisitos especificados é através da verificação de conformidade. Essa técnica assegura que o sistema implementado se comporte de acordo com a especificação e que falhas ou comportamentos indesejáveis sejam detectados, seguindo alguma relação de conformidade estabelecida entre a IUT e a especificação. O processo de detecção de falhas, em geral, possui as seguintes etapas:

1. construção do Modelo Formal: encontrar um modelo formal que descreva o comportamento esperado do sistema. Este modelo serve como uma especificação formal que define o funcionamento do sistema.
2. definição de TPs (Propósitos de Teste)⁴:
 - TPs estabelecem objetivos de teste, tais como verificar funcionalidades específicas, desempenho, ou detectar falhas potenciais. Os propósitos de teste orientam a geração dos casos de teste a partir do modelo de especificação.
 - TPs definem um modelo de falha usado para identificar possíveis falhas e comportamentos indesejados no sistema.
3. geração de casos de teste:
 - técnicas e ferramentas de MBT são usadas para gerar casos de teste a partir do modelo formal de especificação. Esses testes incluem os dados de entrada e

³ Do inglês *Implementation Under Test*.

⁴ Do inglês *Test Purposes*.

os comportamentos esperados, conforme definidos pelos propósitos de teste e pelo modelo de falhas.

- uma cobertura de teste avalia o quanto os casos de teste gerados são capazes de detectar falhas em partes importantes do modelo.
4. execução dos casos de teste: aplicar os casos de teste gerados a uma IUT. Cada caso de teste é executado no sistema, e as saídas geradas são registradas.
 5. verificação de conformidade: comparar as saídas reais de uma IUT com as saídas esperadas e definidas pelo modelo. Esta etapa pode ser automatizada para verificar rapidamente se o comportamento do sistema está conforme o especificado.

De forma geral, com base na especificação e nos propósitos de teste ou no modelo de falha, um conjunto de testes é criado para definir os comportamentos esperados e possíveis falhas do sistema. Cada caso de teste é aplicado à IUT. A saída gerada pelo sistema é então comparada com a saída esperada, e quando houver discrepâncias, uma falha foi detectada [10].

2.2.1 Especificações formais ou Criação do Modelo Formal

Na abordagem de teste usando modelos formais é preciso que algum tipo de modelo seja adotado para especificar o comportamento dos sistemas. Os modelos são representações abstratas de um sistema que captura seus aspectos essenciais e funcionalidades de forma simplificada. Além disso, modelos formais servem como uma ferramenta poderosa para analisar o comportamento do sistema de maneira mais profunda e eficaz, facilitando a identificação de falhas e a garantia da qualidade do *software*.

A escolha do modelo depende das características específicas do sistema e dos objetivos do teste. Quanto menor o grau de formalismo, mais intuitivo e fácil de lidar, porém podem apresentar ambiguidades e interpretações subjetivas. Já os modelos mais formais se utilizam de uma base matemática com linguagens bem definidas e precisas para representar um sistema que, por sua vez, possibilita análises mais precisas onde ambiguidades, incompletudes e inconsistências são mais facilmente descobertas e corrigidas [12].

Um exemplo de modelo formal amplamente utilizado no MBT são as FSM (Máquinas de Estado Finitas)⁵, devido à sua simplicidade, expressividade e capacidade de representar comportamentos dos sistemas [9].

As FSM representam o comportamento do sistema através de:

⁵ Do inglês *Finite State Machines*.

1. Estados: diferentes situações em que o sistema pode se encontrar. Por exemplo, em um sistema de autenticação, os estados podem incluir “usuário não autenticado”, “usuário autenticado”, e “tentativa de autenticação”.
2. Transições: mudanças de estado em resposta a eventos. Por exemplo, uma transição pode ocorrer quando um usuário insere credenciais corretas, movendo o controle do estado “usuário não autenticado” para “usuário autenticado”.
3. Ações: atividades executadas ao realizar uma transição. Por exemplo, ao transitar para o estado “usuário autenticado”, a ação pode envolver a exibição de uma mensagem de boas-vindas.

Uma FSM é um modelo usado para descrever sistemas que possuem um número finito de estados e que fazem transições entre esses estados em resposta a estímulos ou eventos. Um estímulo numa FSM é representado como um *input*, enquanto uma saída produzida é representada como um *output*.

Definição 1 ([13]). *Uma FSM é formalmente definida por $M = (X, Y, S, s_0, \delta, \lambda)$, onde:*

1. X é um alfabeto de entrada finito;
2. Y é um alfabeto de saída finito;
3. S é um conjunto finito de estados;
4. s_0 é o estado inicial $s_0 \in S$;
5. $\lambda : X \times S \rightarrow Y$ representa as funções de saída; e
6. $\delta : X \times S \rightarrow S$ representa as funções de transição.

Note que tal máquina é determinística e completa. Uma FSM é chamada completa se, para cada estado s de M , existe uma transição a partir de s com o símbolo de entrada a , para todo $a \in X$. Uma FSM determinística não permite duas transições diferentes saindo do mesmo estado com símbolos de entrada idênticos.

Abaixo, na Figura 2, temos um exemplo de uma FSM que modela o comportamento de um sistema simples. A FSM possui 3 estados e 4 transições. O estado inicial é q_0 , e os estados finais são q_1 e q_2 . O sistema pode realizar transições entre os estados em resposta aos eventos de entrada a e b , gerando os eventos de saída x e y .

Na abordagem de teste com modelos formais, a chave reside na adoção de um modelo para especificar o comportamento do sistema sob teste. Esses modelos servem como representações abstratas, capturando os aspectos essenciais e funcionalidades de forma simplificada. Embora as FSMs sejam um modelo poderoso de especificação, não são as

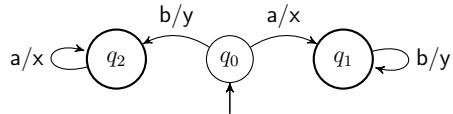


Figura 2 – Um FSM com 3 estados e 4 transições - com entradas e saídas associadas.

Fonte: Elaborado pela autora.

únicas opções. Outros modelos, como IOLTS⁶ e IOVPTS⁷, oferecem maior expressividade e flexibilidade para modelar sistemas complexos.

2.2.2 Propósitos de Testes

Propósitos de teste (TPs)⁸ estabelecem objetivos específicos que orientam o processo de teste, indicando quais aspectos devem ser avaliados durante o teste de um sistema e definindo os critérios para determinar se o sistema está funcionando corretamente. Em um cenário de teste de caixa-preta, onde não temos acesso à estrutura interna da IUT, os propósitos de teste são essenciais para criar casos de teste eficazes. Eles ajudam a identificar quais sequências de ações e respostas (entradas e saídas) precisam ser verificadas para garantir que a IUT esteja se comportando conforme o esperado [1].

Quando a IUT é tratada como uma caixa-preta, o testador ou “ambiente artificial” T e a IUT I são conectados por um canal bidirecional sem memória [14]. Nesse modelo, o ambiente artificial T interage com a IUT I enviando símbolos de ação (entrada) e recebendo respostas (saídas) através do canal. Os propósitos de teste orientam a criação dos casos de teste baseados nas interações permitidas e esperadas entre T e I . Cada propósito de teste define uma situação ou condição específica que deve ser testada, assegurando que todas as funcionalidades e comportamentos relevantes da IUT sejam verificadas [9].

Ao utilizar um *Test Purpose* como um modelo de falhas (*Failure Model*), os testes gerados são projetados para explorar cenários onde o sistema pode falhar, garantindo uma avaliação mais abrangente de sua robustez e capacidade de recuperação [9]. Um modelo de falhas é uma abstração que captura os possíveis defeitos e comportamentos incorretos do sistema. Esse modelo é uma representação estruturada das diversas maneiras pelas quais o sistema pode falhar ou se comportar de maneira inesperada em resposta a estímulos ao sistema. Com base no modelo de falhas, casos de teste específicos são gerados para explorar e detectar esses comportamentos de falha. O modelo não apenas valida o comportamento esperado, mas também se concentra na identificação de vulnerabilidades e situações de falha que podem ocorrer durante a operação do sistema [9].

A utilização de um modelo de falha permite uma avaliação mais abrangente da capacidade do sistema de lidar com situações adversas e se recuperar de falhas de forma

⁶ Do inglês *Input/Output Labeled Transition Systems*.

⁷ Do inglês *Input/Output-Visible Pushdown Transition System*.

⁸ Do inglês *Test Purposes*.

adequada. Ao incorporar esses cenários de falha na geração de casos de teste, é possível verificar como o sistema reage em diferentes condições e avaliar a eficácia dos mecanismos de recuperação e tolerância a falhas implementados.

Ao desenvolver e utilizar modelos de falha, é importante considerar o equilíbrio entre a cobertura de teste desejada e os recursos disponíveis. Nem todas as possíveis falhas podem ser modeladas e testadas devido a restrições de tempo e recursos. Portanto, é necessário fazer escolhas dos cenários de falha mais críticos que devem ser priorizados nos testes, levando em consideração os objetivos do projeto e as expectativas dos usuários finais. Isso garante que o sistema seja testado de forma abrangente, cobrindo tanto o comportamento normal quanto as situações excepcionais priorizadas. Na geração de testes o modelo de falha melhora a robustez e a confiabilidade do sistema ao identificar e corrigir problemas antes de impactarem o usuário final [9].

Uma técnica comum para gerar casos de teste baseados em *Test Purposes* é o uso do produto de modelos. Essa abordagem envolve o modelo da IUT com o *Test Purpose*, criando um produto resultante de ambos [9]. Este produto de modelos é essencialmente um novo modelo que representa o comportamento combinado da IUT e as propriedades específicas do *Test Purpose*. Isso permite a geração de casos de teste direcionados especificamente para verificar as propriedades detalhadas no *Test Purpose*.

Após a criação do produto de modelos, ferramentas de MBT podem ser utilizadas para gerar automaticamente casos de teste a partir deste produto [9]. Esses casos de teste são projetados para cobrir os aspectos do *Test Purpose* dentro do contexto da IUT. O processo de geração envolve a exploração do produto de modelos para identificar sequências de entrada e saída que verificam as propriedades específicas descritas no *Test Purpose* [9].

Os casos de teste resultantes são então executados para validar o comportamento da IUT em relação aos requisitos definidos no *Test Purpose*. Esta abordagem assegura um teste focado e eficiente, verificando propriedades críticas do sistema de forma sistemática e automatizada [9].

2.2.3 Geração e Execução de Casos de Teste

A geração automática de casos de teste é uma etapa essencial na abordagem de MBT (Teste Baseado em Modelos)⁹. Os casos de teste podem ser gerados a partir do modelo formal do sistema usando ferramentas especializadas. Esses testes são configurados para incluir dados de entrada e comportamentos esperados, conforme especificado pelos propósitos de teste e pelo modelo de falhas.

No MBT, a geração de casos de teste é fundamentada em modelos que representam o comportamento do sistema. Tais modelos podem ser elaborados usando diferentes

⁹ Do inglês *Model-Based Testing*.

formalismos, como autômatos finitos, que descrevem o sistema através de uma série de estados e transições. Outros formalismos incluem LTS¹⁰, IOLTS¹¹ e IOVPTS¹², que podem incorporar elementos temporais, simbólicos e outras características específicas do sistema em teste [9].

Já a geração de testes executáveis é um processo mais complexo do que a mera gerando de dados de entrada. Nessa etapa é necessária a inclusão de informações de validação, como os valores esperados de saída da IUT. Para isso, o modelo deve descrever a relação entre entradas e saídas esperadas da IUT, garantindo uma cobertura completa. Este processo envolve não apenas a escolha de valores de entrada, mas também a criação de casos de teste que incluem verificações detalhadas, assegurando que o comportamento do sistema esteja alinhado com as expectativas definidas pelos propósitos de teste e pelo modelo de falhas [9].

2.2.4 Cobertura de Teste

A cobertura de teste trata dos diferentes aspectos do sistema que foram testados por um conjunto de casos de teste. O objetivo é medir a abrangência dos casos de teste em relação ao modelo de falhas do sistema. Em MBT, isso inclui verificar quantos estados, transições, caminhos, entre outros elementos, foram exercitados pelos testes [9], avaliando a amplitude dos testes realizados.

Considerando a cobertura de falhas, dois aspectos importantes são avaliados:

Eficiência: A geração de casos de teste deve ser eficiente, ou seja, os testes devem cobrir o máximo possível das falhas com o mínimo possível de testes.

Eficácia: Os casos de teste gerados devem ser eficazes, ou seja, devem ser capazes de identificar as falhas do sistema.

Em sistemas reativos, a cobertura pode ser avaliada com base em um modelo de falhas [10]. A cobertura baseada no modelo de falhas concentra-se em verificar quão bem o conjunto de testes cobre os cenários de falha especificados pelo modelo de falhas. Em vez de focar apenas em cobrir o código ou as funcionalidades, a cobertura é medida com base em como os testes expõem as falhas que foram identificadas como críticas pelo modelo.

Neste contexto, os casos de teste são gerados especificamente para explorar as falhas identificadas no modelo. Isso significa que os testes são projetados com o objetivo de forçar o sistema a entrar em estados ou executar transições que revelam falhas conforme definido pelo modelo de falhas. Por exemplo, se o modelo identifica que o sistema pode

¹⁰ Do inglês *Labelled Transition Systems*.

¹¹ Do inglês *Input-Output Labelled Transition Systems*.

¹² Do inglês *Input-Output Value Partition Testing Systems*.

falhar ao lidar com uma determinada sequência de eventos, os testes são projetados para forçar essa sequência e verificar se o sistema falha conforme esperado.

O objetivo é garantir que o sistema não apenas execute suas funções corretamente sob condições normais, mas também que se comporte de maneira adequada e identifique as falhas quando confrontado com as condições descritas no modelo de falhas. Assim, os testes demonstram a presença ou ausência de falhas específicas com base em uma especificação formal que descreve como o sistema deve se comportar em situações de falha.

Também podemos utilizar, em sistemas reativos, modelos com outros objetivos. Um dos desafios que sistemas dessa natureza podem enfrentar é de definir o momento adequado para encerrar os testes [9]. Para avaliar se o sistema foi testado de forma suficiente, é possível utilizar métricas de cobertura de teste. Essas métricas não só ajudam a avaliar a exaustividade dos testes, como também orientam na seleção dos casos de teste, assegurando que os requisitos e comportamentos esperados sejam adequadamente abordados, o que ajuda a limitar o número total de testes necessários [10]. Assim, a geração de testes pode ser realizada, e a análise do sistema é garantida de maneira eficiente e manejável.

Diversos métodos de geração de casos de teste podem ser utilizados para alcançar uma boa cobertura de teste. No entanto, a complexidade inerente aos sistemas de *software* torna a tarefa de testar exaustivamente um sistema praticamente impossível. Como Edsger W. Dijkstra afirmou, “O teste só pode provar a presença de erros, mas nunca a sua ausência”. Isso se deve a várias razões práticas, como a complexidade infinita dos sistemas reais, a limitação de recursos para executar todos os testes possíveis, e a impossibilidade de prever todos os possíveis estados e interações do sistema. Embora a exaustividade de testes possa aumentar significativamente a confiança na qualidade do *software*, não elimina a possibilidade de erros. Dada a impossibilidade de garantir a ausência total de erros, mesmo com métodos extensivos de geração de teste, é crucial considerar o conceito de Completude de Conjuntos de Teste.

2.2.5 Completude de Conjuntos de Teste

Um conjunto de testes é considerado completo para uma especificação quando é capaz de fornecer uma cobertura completa de falhas [2]. Em outras palavras, o conjunto de testes deve ser capaz de identificar todas as discrepâncias entre o comportamento esperado (definido pela especificação) e o comportamento real (da implementação).

No entanto, é importante notar que a completa cobertura de teste não implica necessariamente a detecção de todas as possíveis falhas no sistema. Para obter conjuntos completos de teste podem ser adotadas premissas mais flexíveis. No contexto de testes, dois conceitos importantes são:

Soundness: Todas as implementações corretas e algumas incorretas podem passar nos testes, mas nenhuma implementação correta deve falhar.

Exhaustiveness: Todas as implementações incorretas são identificadas como falhas.

Para modelos FSM, a completude dos testes é frequentemente abordada por meio da cobertura de todos os estados e das transições definidas no modelo [15]. A principal função desse tipo de abordagem é garantir que cada estado e cada transição sejam testados, pelo menos, uma vez. Para isso, focamos na cobertura de estados e transições específicas para assegurar que o comportamento do sistema em cada ponto possível seja verificado.

Em modelos assíncronos, como os (IO)LTS, o processo de teste é mais complexo, logo a discussão sobre completude dos testes também se torna mais complexo. Nesses casos, a completude envolve a cobertura não apenas dos estados e transições, mas também de possíveis sequências de ações entre os estados [16]. Em sistemas onde a ordem das ações pode alterar significativamente o comportamento, validar todas as sequências possíveis de eventos torna-se impraticável para garantir que o sistema se comporte corretamente, independentemente da ordem em que as ações ocorram. No entanto, garantir certos padrões de comportamentos e/ou propriedades estabelecidas são de extrema importância para se obter uma cobertura de falhas.

Nos modelos IOLTS e IOVPTS, a completude dos testes pode ser baseada em um modelo de falhas específico. Nesse caso, a completude não garante que qualquer falha seja encontrada na IUT, mas assegura que todas as falhas definidas pelo modelo de falhas proposto sejam detectadas, considerando as condições e restrições assumidas. Assim, a completude na cobertura de teste é definida como a extensão em que um conjunto de testes explora sistematicamente todos os valores de entrada válidos e inválidos, caminhos de controle e estados possíveis do sistema. Este conceito visa minimizar o risco de falhas não detectadas, assegurando uma análise abrangente do comportamento do sistema.

2.2.6 Verificação de Conformidade

A verificação de conformidade, no contexto de sistemas reativos, é uma técnica que visa garantir que o comportamento do sistema implementado esteja em concordância com a sua especificação. Logo, o objetivo é assegurar que o sistema implementado se comporte como esperado, de acordo com sua especificação correspondente.

A verificação de conformidade desempenha um papel crucial no MBT, assegurando que o sistema testado se alinha às especificações e requisitos estabelecidos. Em MBT, os modelos não apenas guiam a geração de casos de teste, mas também servem como referência para validar a conformidade do sistema com os requisitos especificados [9]. O processo de verificação de conformidade envolve a comparação entre o comportamento observado do sistema e o comportamento esperado definido pelo modelo.

Na verificação de conformidade, duas propriedades fundamentais podem ser garantidas [17]:

1. *Soundness*: Um sistema é considerado *sound* quando qualquer comportamento do sistema também é um comportamento encontrado na especificação. Em outras palavras, um sistema *sound* nunca executa ações que não estejam descritas na especificação. Isso assegura que o sistema não realiza comportamentos não autorizados pela especificação.
2. *Completeness*: Um sistema é considerado *complete* quando todo comportamento permitido pela especificação pode ser efetivamente realizado pelo sistema implementado. Assim, a especificação deve capturar integralmente todos os comportamentos desejados, sem omissões ou ambiguidades, assegurando que o sistema possa atingir todos os comportamentos que são autorizados pela especificação.

Na verificação de conformidade, a robustez do MBT não depende apenas da geração e execução de casos de teste, mas também da qualidade dos modelos utilizados. As propriedades de soundness e completeness garantem que o sistema testado está alinhado com suas especificações, proporcionando uma base sólida para a validação do comportamento do *software*. No entanto, para que tais processos sejam realmente eficazes, é imprescindível que os modelos empregados sejam precisos e representem fielmente as especificações do sistema. Isso garante que os testes realizados sejam relevantes e que os resultados obtidos reflitam a realidade do comportamento do *software* em relação às expectativas definidas.

2.3 Modelos Formais

As técnicas e métodos de teste baseado em modelos possuem formalismos apropriados para lidarem com essa etapa de forma rigorosa. Entre esses formalismos existem alguns sistemas de transição que capturam o comportamento de sistemas reativos tais como os IOLTSs (Sistemas de Transição Rotulado de Entrada/Saída)¹³ e os IOVPTSSs (Sistemas de Transição com Pilha Visível de Entrada/Saída)¹⁴.

2.3.1 LTS

Os LTSs (Sistemas de Transição Rotulados)¹⁵ são um modelo formal utilizado para descrever o funcionamento/comportamento de sistemas. Este formalismo é composto por transições rotuladas com ações ou eventos que podem ocorrer no sistema, representando

¹³ Do inglês *Input Output Labeled Transition Systems*.

¹⁴ Do inglês *Input/Output-Visible Pushdown Transition Systems*.

¹⁵ Do inglês *Labeled Transition Systems*.

as operações ou mudanças no sistema [14]. Este modelo é composto por um conjunto de estados e uma relação de transição entre esses estados, em que cada transição é associada a um símbolo de ação. As transições entre estados em um LTS ocorrem através de eventos rotulados, entradas e saídas, independentes. As entradas representam a execução de uma ação recebida e as saídas indicam a resposta do sistema à um estímulo [12].

Um LTS pode ser determinístico, quando dada uma entrada específica e um estado atual, há exatamente uma transição que o sistema pode seguir [15], ou não determinístico, quando um único evento causa várias transições possíveis ou quando uma transição não requer nenhuma entrada, isto é, a ação é independente do ambiente. O símbolo τ representa esse tipo de transição.

O LTS é descrito formalmente na Definição 2 a seguir. Posteriormente, uma estrutura semântica é introduzida para atribuir significado a essa definição.

Definição 2 ([14]). *Um LTS é formalmente definido pela tupla $\mathcal{S} = \langle S, s_0, L, T \rangle$, onde:*

1. *S é um conjunto finito de estados;*
2. *$s_0 \in S$ é o estado inicial;*
3. *L é um conjunto finito de rótulos, ou ações e $\tau \notin L$ é o símbolo da ação interna;*
4. *$T \subseteq S \times L_\tau \times S$ é um conjunto de transições.*

O símbolo τ é usado para representar qualquer ação que causa apenas uma mudança interna de estados.

Exemplo 1. A Figura 3 representa o LTS $\mathcal{S} = \langle S, s_0, L, T \rangle$, onde $S = \{s_0, s_1, s_2, s_3, s_4\}$, $L = \{b, c, t\}$ e as setas indicam as transições.

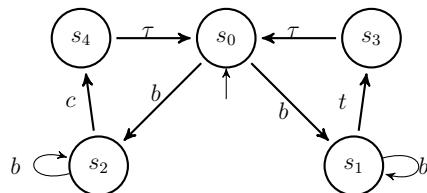


Figura 3 – Um exemplo de LTS.
Fonte: Bonifácio; Moura, 2016 [14].

Observe que o LTS é não determinístico. Dado que, de s_0 com a entrada b , o controle pode passar para o estado s_1 ou o estado s_2 . Adicionalmente, o sistema pode incluir transições internas, representadas pelo símbolo τ . Como de s_3 ou s_4 para s_0 . As transições mencionadas permitem que o sistema transite entre estados sem a necessidade de uma entrada externa ou que haja ambiguidade na transição para uma mesma entrada partindo de um estado, o que é conhecido como não-determinismo. A presença de não

determinismo aumenta tanto a complexidade quanto a adaptabilidade do sistema modelado pelo LTS.

O comportamento de um sistema é determinado pela semântica do LTS que o modela, dado pelo conjunto de traços (*traces or behaviors*). Antes de abordar a definição de traço como uma sequência observável de ações, é fundamental introduzir o conceito de caminhos (*paths*). Caminhos são sequências de estados conectados por transições. Um caminho de um LTS se inicia no estado inicial e segue por uma série de transições, cada transição é rotulada com uma ação específica.

Para definir formalmente um caminho, primeiro assumimos a seguinte definição de homomorfismo. Um homomorfismo é uma função que mapeia elementos de um alfabeto para cadeias (ou palavras) sobre outro alfabeto:

Definição 3. *Sejam A e B alfabetos. Um homomorfismo de A para B é uma função $h : A \rightarrow B^*$ [14].*

Neste contexto, h é a função homomórfica que mapeia cada símbolo do alfabeto A para uma palavra (sequência de símbolos) sobre o alfabeto B . B^* representa o conjunto de todas as palavras possíveis (incluindo a palavra vazia ϵ) que podem ser formadas com os símbolos de B .

Agora, a semântica de LTS pode ser definida.

Definição 4 ([14]). *Seja $\mathcal{S} = \langle S, s_0, L, T \rangle$ um LTS e $p, q \in S$. Assuma $\sigma = \sigma_1, \dots, \sigma_n \in L_\tau^*$. Dizemos que σ é:*

1. *um caminho de p para q se existirem estados $r_i \in S, 0 \leq i \leq n$ e, adicionalmente, tivermos $(r_{i-1}, \sigma_i, r_i) \in T, 1 \leq i \leq n$, com $r_0 = p$ e $r_n = q$;*
2. *um caminho observável de p para q se μ é um caminho de p para q e $\sigma = h_\tau(\mu)$.*

Assim, um caminho σ de p para q é uma sequência de símbolos que representa a mudança do sistema do estado p para o estado q [14]. Observe que um caminho pode conter transições associadas ao símbolo interno τ . No entanto, um caminho observável é obtido removendo os símbolos internos τ de um caminho qualquer [14]. Transições internas são eventos que ocorrem sem interagir diretamente com o ambiente externo e, portanto, não são perceptíveis para um observador externo.

A movimentação do sistema do estado p para o estado q pode ser captada por um caminho observável, que mostra as transições visíveis ao ambiente externo. Dessa forma, a inclusão de transições internas τ em um caminho não altera a relação entre p e q ; as transições internas são invisíveis para um observador externo, mas não interferem na definição de um caminho entre estados.

Um *trace* é representado por $\sigma = \{a_1, a_2, \dots, a_n\}$, onde σ é uma sequência de ações observáveis que podem ser realizadas ao seguir os caminhos do sistema. Cada traço representa uma sequência específica de ações permitidas pelas regras de transição do LTS, iniciando-se no estado inicial.

Definição 5 ([14]). *Seja $\mathcal{S} = \langle S, s_0, L, T \rangle$ um LTS e $p \in S$. O conjunto de traços de p é $\text{tr}(p) = \{\sigma \mid p \xrightarrow{\sigma}\}$, e o conjunto de traços observáveis de p é $\text{otr}(p) = \{\sigma \mid p \xrightarrow{\sigma}\}$. A semântica de \mathcal{S} é o conjunto $\text{tr}(s_0)$, e a semântica observável de \mathcal{S} é o conjunto $\text{otr}(s_0)$.*

Podemos escrever $\text{tr}(\mathcal{S})$ para $\text{tr}(s_0)$ e $\text{otr}(\mathcal{S})$ para $\text{otr}(s_0)$. Se \mathcal{S} não possui transições rotuladas com τ então $\text{otr}(\mathcal{S}) = \text{tr}(\mathcal{S})$ [14]. Assuma que $(s, \tau, s) \notin T$ e que $s_0 \rightarrow s$ vale para qualquer $s \in S$ [14]. Assim temos que:

1. não há transições internas diretas de um estado s para si mesmo usando uma ação interna τ ;
2. há uma transição válida do estado inicial s_0 para qualquer estado s no conjunto de estados S .

Essa restrição impede que um estado s , ao executar uma ação interna τ , retorne imediatamente ao mesmo estado s . Entretanto, não impede a formação de ciclos rotulados por transições τ , ou seja, sequências repetitivas de ações internas τ que podem ocorrer em diferentes estados, levando a um estado de *livelock* [14]. Essa propriedade ocorre em um sistema concorrente ou reativo onde os processos ou componentes continuam a executar ações, mas não conseguem fazer progresso efetivo em direção à conclusão de suas tarefas, ficando em um ciclo de ações que não levam a um estado desejado. Assim, embora não haja transições diretas (s, τ, s) para o mesmo estado s , ainda é possível que ações internas τ em diferentes estados formem ciclos improdutivos [14].

Para evitar *livelock* ou ambiguidades entre estados, um LTS determinístico deve ser adotado.

Definição 6 ([14]). *Um LTS $\mathcal{S} = \langle S, s_0, L, T \rangle$ é determinístico se $s_0 \xrightarrow{\sigma} s_1$ e $s_0 \xrightarrow{\sigma} s_2$ implicarem em $s_1 = s_2$, para todos $s_1, s_2 \in S$ e todo $\sigma \in L^*$.*

2.3.2 IOLTS

O IOLTS (Sistema de Transição Rotulado de Entrada/Saída)¹⁶ é um formalismo que descreve o comportamento de um sistema através de transições e estados. As transições são rotuladas com as ações que causam as transições. Esse modelo é uma extensão dos LTSSs [1]. Porém, nos IOLTSs [14] os rótulos, ou ações, são particionados em entradas e saídas, representando as interações do sistema com o ambiente.

¹⁶ Do inglês *Input Output Labeled Transition System*.

Denotamos por $\mathcal{S}_{\mathcal{J}} = \langle S, s_0, L, T \rangle$ o LTS subjacente de \mathcal{J} . Assim, $\mathcal{S}_{\mathcal{J}}$ é a estrutura formal que descreve o comportamento fundamental de \mathcal{J} .

Definição 7 ([14]). *Um IOLTS é definido, formalmente, por $\mathcal{J} = (S, s_0, L_I, L_U, T)$, onde*

1. L_I é um conjunto finito não-vazio de ações de entrada;
2. L_U é um conjunto finito não-vazio de ações de saída;
3. $L_I \cap L_U = \emptyset$, e $L = L_I \cup L_U$ é o conjunto de ações.
4. $\mathcal{S}_{\mathcal{J}} = \langle S, s_0, L, T \rangle$ é o LTS subjacente de \mathcal{J} .

Nesse modelo, os estados representam eventos e contextos; as transições são as mudanças entre estados desencadeadas por estímulos do ambiente; os rótulos se referem aos símbolos representativos de um dado estímulo ou resposta.

Um IOLTS pode apresentar estados quiescentes, caracterizados pela ausência de ações de saída $x \in L_U$ ou internas τ [1]. Formalmente, um estado s é quiescente se nenhuma ação $x \in L_U \cup \{\tau\}$ estiver definida em s . A quiescência é denotada pelo símbolo δ , de modo que $\delta \notin L_\tau$ [10]. Em uma situação real, a quiescência pode indicar a demora na produção de uma resposta, ou que o tempo foi excedido ou que a implementação não está sendo capaz de responder adequadamente.

Definição 8 ([10]). *Dado um IOLTS $\mathcal{S} = (S, s_0, L_I, L_U, T)$, temos que*

1. a função **out** representa as saídas produzidas a partir de um determinado estado, dada por

$$\text{out}(V) = \bigcup_{s \in V} \left\{ l \in L_U \mid s \xrightarrow{l} \right\}.$$

Para um conjunto de estados V , $\text{out}(V)$ é a união de todos os rótulos de saída $l \in L_U$ que podem ser produzidos a partir de qualquer estado s em V . Logo, $s \xrightarrow{l}$ significa que existe uma transição a partir do estado s rotulada por l .

2. a função **inp** representa as entradas aceitas a partir de um determinado estado, dada por

$$\text{inp}(V) = \bigcup_{s \in V} \left\{ l \in L_I \mid s \xrightarrow{l} \right\}.$$

Similarmente, para um conjunto de estados V , $\text{inp}(V)$ é a união de todos os rótulos de entrada $l \in L_I$ que podem ser aceitos a partir de qualquer estado s em V . Assim, $s \xrightarrow{l}$ novamente indica que existe uma transição a partir de s com o rótulo l .

Um estado $s \in S$ é quiescente se $\text{out}(s) = \emptyset$.

Logo, quando um estado s do modelo não produz nenhuma saída, s é considerado quiescente. Nessa situação, a implementação não realiza transições rotuladas com saídas para outros estados. Para capturar e lidar com esse comportamento, uma transição self-loop rotulada por δ , onde $\delta \notin L_\tau$, é adicionada ao estado quiescente [1]. Formalmente, a transição $s \xrightarrow{\delta} s$, ou (s, δ, s) é adicionada em T .

A adição da transição rotulada com δ torna a ação observável, tal como qualquer outra transição de saída do sistema. Esse recurso é útil em testes de conformidade, pois se quer avaliar não apenas o comportamento ativo (saídas geradas), mas também os estados de inatividade. Ao explicitar a quiescência como uma transição, tal comportamento pode ser capturado pelos testes como uma saída válida, o que facilita a verificação de conformidade.

Exemplo 2. O funcionamento básico de um ar-condicionado, ilustrado pela Figura 4, pode ser um exemplo de sistema modelado por um IOLTS, uma vez que ambos envolvem a entrada de dados e a saída de ações correspondentes. A Figura 5 apresenta um modelo IOLTS para este sistema. Simplificando, quando uma pessoa liga um ar-condicionado e define a temperatura desejada, o aparelho precisa manter o ambiente a essa temperatura. O ar-condicionado começa a ler a temperatura do ambiente (read temp) para verificar se está mais alta do que a temperatura definida. Se a temperatura do ambiente estiver mais alta do que a definida, o ar-condicionado irá resfriar o ambiente: push on. Se estiver mais baixa, pode entrar em modo de espera para economizar energia: sleep. Enquanto estiver ligado, o ar-condicionado continua verificando a temperatura do ambiente para saber quando resfriar push on ou esperar sleep. Essa interação contínua com o ambiente é o que valida essa comparação ao modelo IOLTS, onde a entrada é a leitura da temperatura (read temp) e a saída é a ação de resfriar (push on) ou de esperar (sleep).

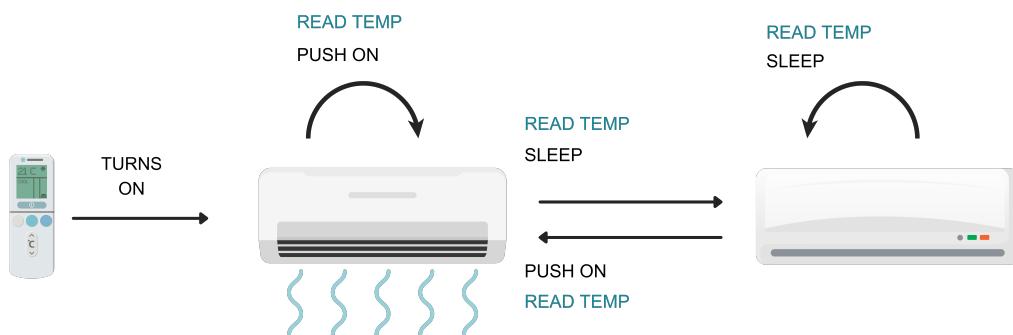


Figura 4 – Funcionamento do ar condicionado
Fonte: Elaborado pela autora.

Com base no formalismo de IOLTS, técnicas de geração de testes, verificação de conformidade, análise de cobertura de falhas e completude de conjuntos de testes, podem

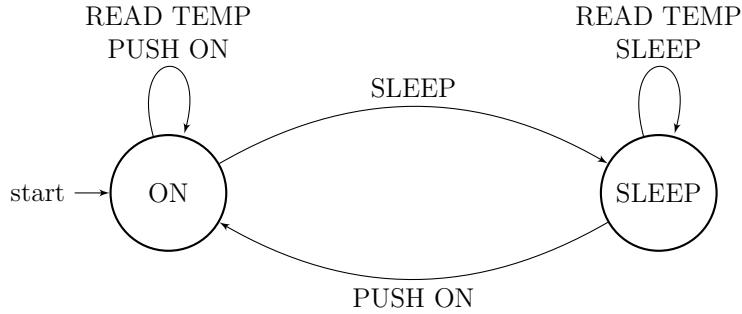


Figura 5 – IOLTS do funcionamento de um ar condicionado

Fonte: Elaborado pela autora.

ser aplicados.

Testes baseados nesse modelo tem sido amplamente utilizados como um framework formal para verificar se uma IUT está em conformidade com uma especificação fornecida, de acordo com um determinado modelo de falha e uma relação de conformidade específica [14]. A ideia geral é que os comportamentos observados numa IUT sejam comparados ao comportamento modelado pela especificação. Quando algum comportamento distinto, de acordo com a relação de conformidade, é identificado, uma falha é então detectada [18]. Este processo é realizado fornecendo as entradas (estímulos) para a IUT e para a sua respectiva especificação, e comparando as saídas (observáveis) geradas por ambas, a fim de identificar possíveis falhas.

A semântica de um IOLTS define seu comportamento, especificamente através de um conjunto de traços (*traces*) — sequências de entradas e saídas que o IOLTS modela. Esse conceito forma a base para estabelecer relações de conformidade entre modelos, permitindo a comparação das ações executadas em um mesmo processo.

Um *trace* é representado por $\sigma = \{a_1, a_2, \dots, a_n\}$, onde σ é uma sequência de ações que descreve a transição do estado inicial s_0 para o estado s . Como visto, em um LTS os traços são apenas sequências de ações, enquanto em um IOLTS os traços observáveis são sequências de entradas e saídas. Essa distinção permite capturar a interação mais rica entre o sistema modelado e seu ambiente, considerando tanto as ações do sistema quanto as respostas do ambiente.

A semântica de um IOLTS é definida pela semântica do seu LTS subjacente.

Definição 9 ([14]). *A semântica de um IOLTS \mathcal{J} é o conjunto $\text{otr}(\mathcal{J}) = \text{otr}(\mathcal{S}_{\mathcal{J}})$.*

2.3.3 VPTS

O modelo VPTS (Sistema de Transição de Pilha Visível)¹⁷ é um formalismo com memória potencialmente infinita [19]. Este formalismo é composto por um conjunto de

¹⁷ Do inglês *Visible Pushdown Transition System*.

estados, uma pilha (que pode ser observada e manipulada externamente) e um conjunto de transições rotuladas que descrevem as operações realizadas na pilha. VPTSs permitem representar trocas assíncronas de mensagens entre o sistema e o ambiente, onde as saídas podem ocorrer de maneira independente das entradas, ou seja, as mensagens de saída são tratadas como eventos separados [19].

Definição 10 ([19]). *Um VPTS é definido formalmente por $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T \rangle$, onde:*

1. *S é um conjunto finito de estados;*
2. *$S_{in} \subseteq S$ é o conjunto de estados iniciais;*
3. *L é um alfabeto;*
4. *$\varsigma \notin L$ é um símbolo especial que indica ação interna;*
5. *Γ é o alfabeto da pilha (também conhecido como alfabeto de empilhamento), onde $\perp \notin \Gamma$ é um símbolo especial que indica fundo da pilha;*
6. *$T = T_c \cup T_r \cup T_i$, onde $T_c \subseteq S \times L_c \times \Gamma \times S$, $T_r \subseteq S \times L_r \times \Gamma_\perp \times S$ e $T_i \subseteq S \times (L_i \cup \varsigma) \times \sharp \times S$, onde $\sharp \notin \Gamma_\perp$ é um símbolo reservado.*

Com a complexidade adicional devida a memória de pilha, os VPTSs permitem uma variedade maior de comportamentos. As transições entre estados podem agora envolver diferentes operações na pilha. Suponha que $t = (p, x, Z, q)$ seja uma transição de T [19]:

1. Chamamos de **transição de push** quando $t \in T_c$, o que significa que o sistema lê uma entrada x enquanto se move do estado p para q em S , e empurra o símbolo Z para a pilha.
2. Chamamos de **transição de pop** se $t \in T_r$. Nesse caso, o sistema lê uma entrada $x \in L_r$ enquanto muda de estado de p para q em S , e remove o símbolo Z da pilha. Observe que uma transição de *pop* pode ocorrer quando a pilha está vazia, mantendo a pilha inalterada se o símbolo de *pop* for \perp .
3. Chamamos de **transição simples** quando $t \in T_i$ e $x \in L_i$. Nesse caso, a transição t lê x enquanto se move de p para q , sem alterar a pilha.
4. Chamamos de **transição interna** quando $t \in T_i$ e $x = \varsigma$. Uma transição interna não altera a pilha, e também não lê nenhum símbolo da entrada.

A semântica de um VPTS, determina o comportamento do sistema modelado, é definida pela maneira como as configurações são interligadas através das transições e

como essas transições formam traces que representam o comportamento do sistema. As configurações fornecem o estado do sistema, as transições detalham como mudar entre esses estados, e os traces oferecem uma visão global do comportamento do sistema ao longo do tempo. As configurações são pares formados por um estado p e o conteúdo da pilha α , como mostra a Definição 11.

Definição 11 ([19]). *Uma configuração de um VPTS $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T \rangle$ é um par $(p, \alpha) \in S \times (\Gamma^* \cup \{\perp\})$, onde p é um estado e α é uma sequência de símbolos representando o conteúdo da pilha.*

1. Quando $p \in S_{in}$ e $\alpha = \perp$, (p, α) é uma configuração inicial de \mathcal{S} .
2. O conjunto de todas as configurações de \mathcal{S} é representado por $\mathcal{C}_{\mathcal{S}}$.

As transições entre as configurações são definidas por regras que especificam como o estado e a pilha podem ser modificados.

Definição 12 ([19]). *Se $(q, \alpha) \in \mathcal{C}_{\mathcal{S}}$ e $\ell \in L_{\varsigma}$, escrevemos $(p, \alpha) \xrightarrow{\ell} (q, \beta)$ quando há uma transição $(p, \ell, Z, q) \in T$, tal que:*

1. *Transição de Empilhamento ou Push ($\ell \in L_c$): Adiciona o símbolo Z ao topo da pilha, $\beta = Z\alpha$.*
 - A transição (s, x, Z, q) é rotulada como x/Z_+ .
2. *Transição de Desempilhamento ou Pop ($\ell \in L_r$): Remove o símbolo do topo da pilha, com duas possíveis variações:*
 - a) $Z \neq \perp$ e $\alpha = Z\beta$: Z é o símbolo que será removido do topo da pilha, e α representa o estado atual da pilha. Após a transição, β será o estado da pilha após remover Z .
 - b) $Z = \alpha = \beta = \perp$ (pilha vazia): Essa variação lida com o caso especial em que a pilha está vazia. Assim, a pilha permanece inalterada.
 - A transição (s, x, Z, q) é rotulada como x/Z_- .
3. *Transição Interna ($\ell \in L_i \cup \{\varsigma\}$): A pilha permanece inalterada, $\alpha = \beta$.*
 - A transição (s, x, \sharp, q) é rotulada apenas como x .

Um **movimento simples** de \mathcal{S} é representado por $(p, \alpha) \xrightarrow{\ell} (q, \beta)$ quando uma transição $(p, \ell, Z, q) \in T$ é usada neste movimento. Após este movimento, (q, β) também é uma configuração de \mathcal{S} : $(q, \beta) \in \mathcal{C}_{\mathcal{S}}$.

Um trace é uma sequência de rótulos de transição que leva de uma configuração inicial até outra configuração.

Definição 13 ([19]). *Seja $\sigma = l_1, \dots, l_n$ uma palavra em L_ς^* .*

1. *Dizemos que σ é um **trace** de (p, α) até (q, β) se existem configurações intermediárias $(r_i, \alpha_i) \in \mathcal{C}_S$, $0 \leq i \leq n$, tais que $(r_{i-1}, \alpha_{i-1}) \xrightarrow{l_i} (r_i, \alpha_i)$, com $(r_0, \alpha_0) = (p, \alpha)$ e $(r_n, \alpha_n) = (q, \beta)$.*
 - *Um **trace** σ de (p, α) para (q, β) é representado por $(p, \alpha) \xrightarrow{\sigma} (q, \beta)$.*
2. *Dizemos que (p, α) até (q, β) é um **trace observável** se é uma sequência $\sigma \in L^*$ obtida removendo-se as transições internas ς de um trace μ de (p, α) até (q, β) .*
 - *Um **trace observável** de (p, α) para (q, β) , é representado por $(p, \alpha) \xrightarrow{\sigma} (q, \beta)$.*

O trace se inicia em (p, α) e termina em (q, β) , e a configuração (q, β) é dita **alcançável** a partir de (p, α) . Isso significa que existe um trace que leva o sistema de uma configuração inicial de \mathcal{S} até (q, β) . Quando (q, β) é alcançável em \mathcal{S} , então é possível começar em uma configuração inicial de \mathcal{S} e seguir uma sequência de transições para chegar em (q, β) .

Exemplo 3. A Figura 6 representa um VPTS \mathcal{S} com os seguintes elementos:

1. *Estados: $S = \{s_0, s_1\}$, sendo $S_{in} = \{s_0\}$ o conjunto de estados iniciais.*
2. *Rótulos: $L_c = \{b\}$, $L_r = \{c, t\}$, $L_i = \emptyset$ e $\Gamma = \{Z\}$.*
3. *Transição push: (s_0, b, Z, s_0) .*
4. *Transições pop: (s_0, c, Z, s_1) , (s_0, t, Z, s_1) , (s_1, c, Z, s_1) , (s_1, t, Z, s_1) .*
5. *Transição interna: $(s_1, \varsigma, \sharp, s_0)$.*

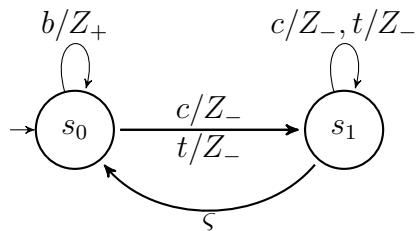


Figura 6 – A VPTS \mathcal{S}_1 , with $L_c = \{b\}$, $L_r = \{c, t\}$, $L_i = \emptyset$.
Fonte: Bonifácio, 2023 [19].

O comportamento de \mathcal{S} diz que o símbolo b ocorre tantas vezes quanto necessário, empilhando o símbolo Z . Em seguida, ao menos um c ou t correspondente deve ocorrer, e

então Z é desempilhado, ou então vários símbolos c e t ocorrerem enquanto a pilha não estiver vazia. Na sequência, este processo se reinicia com \mathcal{S} voltando o controle para o estado s_0 , através de um rótulo internal ς .

2.3.4 IOVPTS

O IOVPTS (Sistemas de Transição com Pilha Visível de Entrada/Saída)¹⁸ é uma extensão do VPTS. O formalismo VPTS pode ser utilizado para modelar sistemas com uma memória potencialmente infinita e com a capacidade de interagir de forma assíncrona com um ambiente externo [19]. Em tais situações, pode-se tratar alguns rótulos de ações como símbolos que o VPTS “recebe” do ambiente e outros rótulos de ações como símbolos que o VPTS “retorna” para o ambiente [19]. O IOVPTS atende a essa necessidade, diferenciando símbolos de ações de entrada e de saída.

Definição 14 ([19]). *O IOVPTS é definido por $\mathcal{J} = \langle S, S_{in}, L_I, L_U, \Gamma, T \rangle$, onde:*

1. L_I é um conjunto finito de ações de entrada;
2. L_U é um conjunto finito de ações de saída;
3. $L_I \cap L_U = \emptyset$, e $L = L_I \cup L_U$ é o conjunto de ações; e
4. $\langle S, S_{in}, L, \Gamma, T \rangle$ é o VPTS subjacente associado a \mathcal{J} .

Denotamos a classe de todos os IOVPTS com alfabeto de entrada L_I e alfabeto de saída L_U por $IOVP(L_I, L_U)$. Observe que, em qualquer referência a um modelo IOVPTS, podemos substituí-lo pelo seu VPTS subjacente [19].

A interpretação de uma transição de um IOVPTS $\mathcal{J} = \langle S, S_{in}, L_I, L_U, \Gamma, T \rangle$, é que para $t = (s, x, Z, s')$, a partir de um estado atual $s \in S$, a transição é caracterizada por um evento, que pode ser uma ação de entrada $x \in L_I$ ou de saída, $y \in L_U$. Com a execução da transição um símbolo $Z \in \Gamma$ pode ser empilhado ou desempilhado, de acordo com o tipo de transição (*push*, *pop* ou interna), conduzindo o sistema a um novo estado $s' \in S$.

No IOVPTS, além das operações de empilhar e desempilhar como no VPTS, o conjunto de ações são particionados, introduzindo a noção **chamadas** (*calls*) e **retornos** (*returns*) para que interações mais complexas com o ambiente possam ser modeladas [3]. O IOVPTS então permite a modelagem de sistemas que envolvem interações de entrada/saída e, simultaneamente, o controle sobre o fluxo de chamadas e retornos, resultando em maior expressividade na representação de sistemas complexos.

¹⁸ Do inglês *Input/Output-Visible Pushdown Transition System*.

1. **Transições de *call*** ($t \in T_c$): Uma transição de chamada é uma transição *push* onde o sistema move-se do estado p para o estado q , empilhando um símbolo específico que indica o início de uma nova chamada. Formalmente, uma transição de chamada é representada como $t = (p, x, Z, q)$, onde $x \in L$, $Z \in \Gamma$, e o símbolo Z empilhado representa o contexto da chamada.
2. **Transições de *return*** ($t \in T_r$): Uma transição de retorno é uma transição *pop* onde o sistema move-se do estado p para o estado q , desempilhando um símbolo específico que indica o retorno de uma chamada anterior. Formalmente, uma transição de retorno é representada como $t = (p, x, Z, q)$, onde $x \in L_r$, $Z \in \Gamma$, e o símbolo Z desempilhado representa o contexto do retorno.

A semântica de um IOVPTS é definida como o conjunto de seus traces observáveis, ou seja, os traces observáveis de seu VPTS subjacente.

Definição 15 ([19]). *Seja $\mathcal{J} = \langle S, S_{in}, L_I, L_U, \Gamma, T \rangle$ um IOVPTS. A semântica de \mathcal{J} é o conjunto $otr(\mathcal{J}) = otr(\mathcal{S}_{\mathcal{J}})$, onde $\mathcal{S}_{\mathcal{J}}$ é o VPTS subjacente associado a \mathcal{J} .*

Além disso, ao se referir a um IOVPTS \mathcal{J} , a notação $\xrightarrow{\mathcal{J}}$ e $\xRightarrow{\mathcal{J}}$ devem ser entendidas como $\xrightarrow{\mathcal{S}}$ e $\xRightarrow{\mathcal{S}}$, respectivamente, onde \mathcal{S} é o VPTS subjacente associado a \mathcal{J} .

2.4 Verificação de Conformidade

A noção de conformidade se refere à ideia de que uma implementação ou sistema deve estar de acordo com a sua respectiva especificação, ou seja, o comportamento da implementação não deve divergir do comportamento esperado descrito na especificação. Para definir a relação de conformidade é necessário que as componentes dessa relação estejam claramente definidas. Tais componentes incluem:

1. *Implementação*: é o sistema que deve ser testado (IUT¹⁹). Em testes de caixa preta, uma implementação é testada de acordo com os seus comportamentos e interações com o ambiente, sem o conhecimento da funcionamento interno. A única maneira de interagir com a implementação é por meio de suas interfaces, fornecendo *inputs* e observando os *outputs*. O objetivo do teste é verificar a correção do comportamento da IUT em suas interfaces [1].
2. *Especificação*: é uma descrição formal ou informal de como o sistema ou componente - implementação - deve se comportar [9]. A especificação define as expectativas, requisitos e restrições que a implementação deve atender. Em testes baseados em modelos a especificação é expressa em alguma linguagem ou modelo formal [1]. O conjunto de todos os comportamentos (ou palavras) pertencentes ao modelo ou a

¹⁹ Implementation under test

essa linguagem pode ser denotado por $SPEC$. Logo, para uma especificação $\mathcal{S} \in SPEC$ queremos verificar se o comportamento da $IUT \mathcal{I}$ está em conformidade com \mathcal{S} [1].

3. *Relação de conformidade:* A relação de conformidade tem como objetivo assegurar que o comportamento de um sistema esteja de acordo com a sua respectiva especificação. A definição de uma relação de conformidade passa pela estrutura das componentes, por exemplo, os modelos e sua expressividade para especificar tais componentes, e o objetivo do teste. Logo, para verificar se uma IUT está em conformidade com uma especificação, é preciso que uma relação de conformidade seja formalmente definida, tal como a relação *ioco* [1].

Definição 16 ([1]). *Seja um conjunto de modelos MOD e uma especificação SPEC. Supondo que qualquer IUT possa ser modelada por um objeto formal I_{IUT} pertencente a MOD.*

1. *Definimos uma relação de implementação $\mathbf{imp} \subseteq MOD \times SPEC$.*
2. *Um modelo de implementação I é considerado correto em relação a uma especificação $s \in SPEC$ se $(i, s) \in \mathbf{imp}$.*

Neste contexto, a Figura 7 ilustra a relação entre a implementação, a especificação e a conformidade. A implementação do sistema é representada por uma caixa preta, sendo o

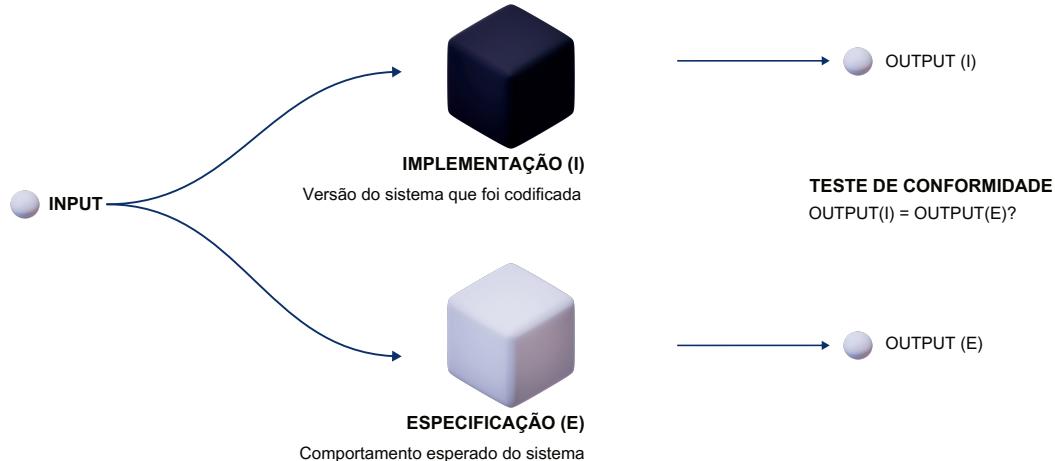


Figura 7 – Relação de conformidade
Fonte: Elaborado pela autora.

funcionamento interno desconhecido do testador. Em contraste, a especificação do sistema é representada por uma caixa branca, indicando que o comportamento esperado do sistema é conhecido pelo testador. A especificação define claramente o que o sistema deve fazer para cada *input* possível, fornecendo uma referência precisa para avaliar a correção da implementação.

Tanto a implementação quanto a especificação recebem entradas (*inputs*) e geram saídas (*outputs*). Entradas são gatilhos (*triggers*) que fazem o sistema executar certas ações ou acionam mudanças de estado [16]. Em um contexto de testes, estas são definidas como eventos ou condições que desencadeiam respostas específicas do sistema. Saídas são as respostas geradas pelo sistema em reação às entradas recebidas. Essas saídas devem ser validadas para garantir que o sistema se comporte como esperado quando submetido a diversas condições de entrada.

O comportamento de um sistema é determinado pela sequência de entradas e as saídas correspondentes [16]. Por isso, é essencial analisar toda a sequência de interações entre entradas e saídas para entender o comportamento geral do sistema e garantir que esteja de acordo com suas especificações [16]. Assim, o teste de conformidade, em vez de analisar apenas uma resposta isolada a uma única entrada, deve considerar o comportamento do sistema, isto é, como as saídas são produzidas em relação a uma série de entradas. Essa sequência determina se o sistema está funcionando de acordo com as especificações.

Na Figura 7, essa comparação de comportamento é representada pela comparação do *output* da implementação (*output(I)*) com o *output* esperado pela especificação (*output(E)*). Se ambos os *outputs* forem iguais para todos os *inputs* possíveis, podemos concluir que a implementação está em conformidade com a especificação.

Estabelecida a relação de conformidade, a verificação de conformidade é o processo de validar se uma implementação satisfaz todos os requisitos definidos na especificação formal. Este processo deve certificar que a implementação do sistema está de acordo com sua respectiva especificação. O teste de conformidade visa garantir que o software atenda exatamente às expectativas e requisitos previamente definidos [20]. A Figura 8 ilustra o processo de verificação com suas respectivas etapas de modelagem, geração e execução dos testes.

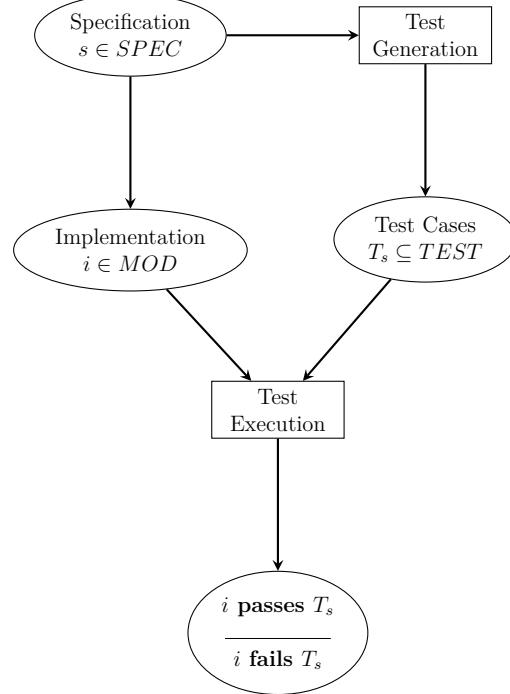


Figura 8 – Arquitetura de Teste

Fonte: Tretmans, 2008 [1].

O *Teste de Conformidade* envolve a avaliação, por meio de testes, se uma implementação está em conformidade, com respeito à relação de implementação imp , com sua especificação.

Definição 17 ([1]). *O teste de conformidade é então definido como a busca por um conjunto de testes T_s tal que:*

$$\forall i \in MOD : i \text{ } imp \text{ } s \Leftrightarrow i \text{ } passes \text{ } T_s$$

Logo, para uma implementação i imp s , dizemos que i implementa a especificação s se, e somente se, i passa pelos testes de T_s , ou seja, i passes T_s . A Figura 9 ilustra o teste de conformidade entre uma IUT e uma especificação.

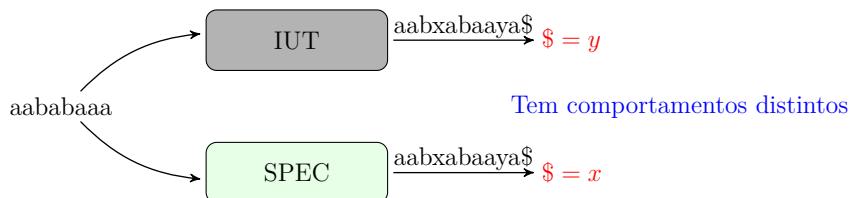


Figura 9 – A IUT não está em conformidade com a SPEC.

Fonte: Elaborado pela autora.

A figura mostra a sequência de entradas fornecida tanto para a IUT quanto para a SPEC, representada por “aababaaa”, e a forma como essas entradas são processadas. As

setas no diagrama demonstram como as entradas são mapeadas para as saídas, destacando a correspondência ou discrepância entre o comportamento observado e esperado. As saídas resultantes da aplicação da entrada são “aabxabaayay” na IUT e “aabxabaayax” na SPEC. Ao verificar a conformidade entre as saídas produzidas nota-se a diferença no último caractere, indicando que a implementação não está em conformidade com a especificação.

Já no exemplo da Figura 10, o resultado é semelhante ao anterior, exceto pelo último caractere, que é igual tanto na IUT quanto na SPEC, indicando que a implementação está em conformidade com a especificação para essa sequência de entrada.

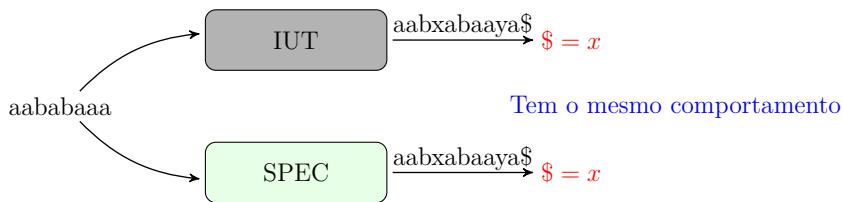


Figura 10 – IUT e SPEC tem o mesmo comportamento.

Fonte: Elaborado pela autora.

2.4.1 Relação *ioco*

A conformidade *ioco* (Conformidade de Entrada/Saída)²⁰ é uma das relações mais usadas para a verificação de sistemas reativos em ambientes assíncronos. A relação de conformidade *ioco* determina quando uma IUT se comporta de acordo com uma especificação, quando ambas são modeladas por IOLTSs, através de sequências de estímulos aplicadas a IUT que produzem saídas previstas também na especificação [1]. Este conceito é fundamental em áreas como engenharia de software, sistemas embarcados e telecomunicações, onde a precisão e a confiabilidade são cruciais.

Podemos formalizar a relação de conformidade *ioco* como segue. Uma IUT, denotada por \mathcal{I} , é considerada *ioco-conforme* a uma especificação \mathcal{S} se, e somente se, para cada sequência de entradas que leva a um estado em \mathcal{S} , as saídas observadas na IUT estão contidas no conjunto de saídas esperadas pela especificação. A Definição 18 expressa formalmente a relação *ioco*.

Definição 18 ([1]). *Seja $\mathcal{S} = \langle S, s_0, L_I, L_U, T \rangle$ um IOLTS de especificação, e $\mathcal{I} = \langle Q, q_0, L_I, L_U, R \rangle$ um IOLTS da IUT, e $L = L_I \cup L_U$, o alfabeto de entradas e saídas.*

1. $\text{out}(V) = \bigcup_{s \in V} \{\ell \in L_U \mid s \xrightarrow{\ell}\}$, para todo $V \subseteq S$.

Esta expressão define o conjunto de saídas possíveis a partir de um conjunto de estados V do sistema. Para cada estado s em V , ela coleta todas as saídas ℓ que podem ser geradas a partir de s através de transições rotuladas por ℓ , incluindo

²⁰ Do inglês *Input/Output Conformance*.

a saída δ , que representa uma saída que pode ser gerada por uma transição não observável (ou seja, uma transição que não produz uma saída visível). O resultado é a união de todas essas saídas para todos os estados em V .

2. $s \text{ after } \sigma = \{q \mid s \xrightarrow{\delta} q\}$, para todo $s \in S$, $\sigma \in L^*$.

Esta função define o conjunto de estados que podem ser alcançados a partir do estado s após a execução de uma sequência de transições rotuladas por σ . Ou seja, $s \text{ after } \sigma$ é o conjunto de todos os estados q que podem ser alcançados a partir de s ao seguir a sequência de transições σ .

3. $\mathcal{I} \text{ ioco } \mathcal{S} \text{ se, e somente se, } \text{out}(q_0 \text{ after } \sigma) \subseteq \text{out}(s_0 \text{ after } \sigma)$, para todo $\sigma \in \text{otr}(\mathcal{S})$.

Esta condição formaliza a relação de conformidade ioco. Diz que a implementação \mathcal{I} é ioco-conforme à especificação \mathcal{S} se, para toda sequência de transições observáveis σ que leva a um estado na especificação \mathcal{S} (denotado como $\text{otr}(\mathcal{S})$), as saídas observadas a partir do estado inicial da implementação q_0 após a sequência σ estão contidas nas saídas observadas a partir do estado inicial da especificação s_0 após a mesma sequência σ .

De forma intuitiva, uma IUT (\mathcal{I}) está em conformidade com uma especificação (\mathcal{S}) se, após uma sequência de estímulos, as saídas produzidas pela implementação correspondem ou estão contidas nas saídas esperadas pela especificação, considerando apenas as saídas observáveis. Essa abordagem assegura que, mesmo que a implementação possa apresentar comportamentos adicionais ou não especificados, ela não pode produzir saídas que não são permitidas pela especificação. Assim, garantimos a conformidade e a integridade do sistema em relação ao que foi definido na especificação [14].

A classe de todos os modelos IOLTS com o alfabeto de entrada L_I e de saída L_U é denotado por $\mathcal{IO}(L_I, L_U)$. Assim, uma implementação $\mathcal{I} \in \mathcal{IO}(L_I, L_U)$ é dita *ioco-conforme* a uma especificação $\mathcal{S} \in \mathcal{IO}(L_I, L_U)$ se qualquer sequência de estímulos de entrada derivado de \mathcal{S} e executado em \mathcal{I} resulta numa mesma saída prevista por \mathcal{S} [1].

2.4.2 Relação *ioco-like*

Para verificar se IUTs, descritas como IOVPTSs, estão em conformidade com uma especificação IOVPTS dada, uma nova relação de conformidade, baseada na relação *ioco* sobre IOLTSs, é definida. A ideia da relação *ioco-like* é então verificar se, dadas uma especificação \mathcal{S} e uma IUT \mathcal{I} , dizemos que \mathcal{I} está em conformidade com \mathcal{S} quando, para qualquer comportamento observável σ de \mathcal{S} , qualquer símbolo de saída que \mathcal{I} possa emitir após percorrer σ está, necessariamente, entre os símbolos de saída que \mathcal{S} também pode emitir após percorrer o mesmo σ [3]. Observe que esta intuição é similar a da relação *ioco* para IOLTSs, porém agora os modelos, IOVPTSs, possuem uma memória auxiliar de pilha. A Definição 19 descreve precisamente a relação *ioco-like*.

Definição 19 ([3]). Seja $\mathcal{S} = \langle S, S_{in}, L_I, L_U, \Gamma, T \rangle$ e $\mathcal{J} = \langle Q, Q_{in}, L_I, L_U, \Delta, R \rangle$ dois IOVPTSSs, com $L = L_I \cup L_U$, temos

1. a função $\text{out}: \mathcal{P}(C_{\mathcal{S}}) \rightarrow L_U$ tal que

$$\text{out}(V) = \bigcup_{(s, \alpha) \in V} \left\{ \ell \in L_U \mid (s, \alpha) \xrightarrow{\ell} \right\}$$

A função out recebe como entrada um conjunto V de configurações $C_{\mathcal{S}}$, tal que cada configuração é um par (s, α) , onde s é um estado do sistema \mathcal{S} e α é o conteúdo da pilha. A função então retorna a união dos símbolos de saída $\ell \in L_U$ que podem ser produzidos a partir de qualquer configuração (s, α) pertencente a V . A notação $(s, \alpha) \xrightarrow{\ell}$ denota que, a partir da configuração (s, α) , o sistema pode realizar uma transição que resulta na produção do símbolo de saída ℓ .

2. a função $\text{after}: C_{\mathcal{S}} \times L^* \rightarrow \mathcal{P}(C_{\mathcal{S}})$ tal que

$$(s, \alpha) \text{ after } \sigma = \{(q, \beta) \mid (s, \alpha) \xrightarrow{\sigma} (q, \beta)\}, \text{ para todo } (s, \alpha) \in C_{\mathcal{S}}, \sigma \in L^*.$$

A função after toma uma configuração (s, α) , onde s é um estado de \mathcal{S} e α é o conteúdo da pilha (de Γ^*), e uma sequência σ de símbolos do alfabeto L^* . A função retorna um conjunto de configurações (q, β) , onde q é um estado de \mathcal{S} e β é o conteúdo da pilha resultante após a execução da sequência σ a partir de (s, α) . A função after basicamente computa o conjunto de todas as configurações (q, β) alcançáveis a partir de uma configuração inicial (s, α) após executar a sequência de ações σ . A sequência de transições que leva o sistema de (s, α) para (q, β) consumindo σ é denotada por $(s, \alpha) \xrightarrow{\sigma} (q, \beta)$.

3. \mathcal{J} ioco-like a \mathcal{S} se para todo $\sigma \in \text{otr}(\mathcal{S})$, $q_0 \in Q_{in}$, $\ell \in \text{out}((q_0, \perp) \text{ after } \sigma)$ existe algum $s_0 \in S_{in}$ tal que $\ell \in \text{out}((s_0, \perp) \text{ after } \sigma)$.

Para que \mathcal{J} seja considerado ioco-like conforme a \mathcal{S} , duas condições devem ser atendidas para cada sequência de ações observáveis σ de \mathcal{S} :

- a) $\sigma \in \text{otr}(\mathcal{S})$: σ é um traço observável de \mathcal{S} , ou seja, uma sequência de ações que pode ser realizada pela especificação.
- b) para qualquer estado inicial q_0 de \mathcal{J} e símbolo de saída ℓ , se ℓ pode ser emitido por \mathcal{J} após executar σ , então também deve ser possível emitir ℓ em \mathcal{S} após executar σ .
 - $\ell \in \text{out}((q_0, \perp) \text{ after } \sigma)$ significa que \mathcal{J} pode emitir a saída ℓ depois de processar σ a partir do estado inicial (q_0, \perp) .
 - a expressão $\ell \in \text{out}((s_0, \perp) \text{ after } \sigma)$ afirma que a mesma saída ℓ deve ser possível para \mathcal{S} a partir do estado inicial (s_0, \perp) .

A função *after* é essencial para entender como uma sequência de ações σ pode transformar o estado e o conteúdo da pilha de um sistema de transição visível. Esse rastro de execução ajuda a determinar a sequência de ações que altera o estado e a pilha do sistema, uma aspecto fundamental para determinar a conformidade da implementação com a especificação.

Já a função *out* permite identificar quais símbolos de saída podem ser produzidos pelo sistema após a execução dessa sequência de ações. Essa característica também é crucial para verificar se as saídas geradas pela implementação estão de acordo com as saídas permitidas pela especificação após a sequência de ações.

Assim, informalmente, dizemos que \mathcal{J} (a implementação sob teste) é considerada conforme a \mathcal{S} (a especificação) se, para toda sequência de ações observáveis σ que pode ser realizada pela especificação, qualquer saída que a implementação \mathcal{J} emita após processar essa sequência também deve ser uma saída que a especificação \mathcal{S} pode emitir após processar a mesma sequência, garantindo que as saídas produzidas pela implementação estejam sempre de acordo com as saídas permitidas pela especificação - isto é, as saídas produzidas pela implementação correspondem ou estão contidas nas saídas esperadas pela especificação.

3 FERRAMENTA PARA RELAÇÃO *IOCO-LIKE*

O número de sistemas reativos usados vem crescendo cada vez mais, aumentando assim também a demanda pela atividade de teste em sistemas dessa natureza, especialmente sistemas reativos mais complexos que envolvem a necessidade de memória auxiliar. Verificar a conformidade entre modelos que especificam o comportamento desses sistemas é essencial para assegurar que as implementações estejam de acordo com as especificações, particularmente em ambientes que demandam alta confiabilidade.

Em um trabalho mais recente [21], um método de verificação de conformidade foi proposto para sistemas reativos usando memória de pilha. A relação de conformidade desenvolvida estende a tradicional relação *ioco* e usa como formalismo base os modelos IOVPTSSs. Apesar do avanço teórico, não havia até então uma ferramenta prática dedicada especificamente à verificação de conformidade *ioco-like* para esses sistemas. A criação de tal ferramenta é crucial, pois permitiria a análise automatizada e precisa da conformidade entre implementações e suas especificações.

Com base nesse método, o objetivo deste trabalho é implementar uma ferramenta que realize a verificação de conformidade entre uma especificação e uma implementação usando modelos IOVPTSSs. A ferramenta utiliza um cenário de teste *white-box*, onde a estrutura da implementação é conhecida, permitindo uma análise detalhada de conformidade. Os algoritmos que verificam, de forma eficiente, se a implementação está em conformidade com a especificação foram desenvolvidos. Além dos vereditos de conformidade, a ferramenta também gera, se necessário, casos de teste que evidenciem os vereditos de não conformidade.

Essa ferramenta representa um avanço significativo na área de testes baseados em modelos (MBT), ao fornecer uma abordagem formal para a verificação de sistemas reativos com memória infinita, um aspecto ainda pouco explorado na literatura [21].

3.1 Desenvolvimento da Ferramenta

A ferramenta desenvolvida para este projeto foi implementada na linguagem Python 3.9.11, escolhida pela sua flexibilidade, facilidade de uso e ampla disponibilidade de bibliotecas que facilitam o desenvolvimento de sistemas. A organização dos arquivos e estruturação do código, bem como as principais bibliotecas utilizadas no desenvolvimento da ferramenta são detalhadas a seguir.

3.1.1 Estrutura e Implementação

O desenvolvimento da ferramenta está organizada em módulos de acordo com suas funcionalidades, conforme ilustrado na Figura 11. O módulo principal, destacado em verde, é a parte central das operações. Os demais módulos estão representados em cada caixa branca.

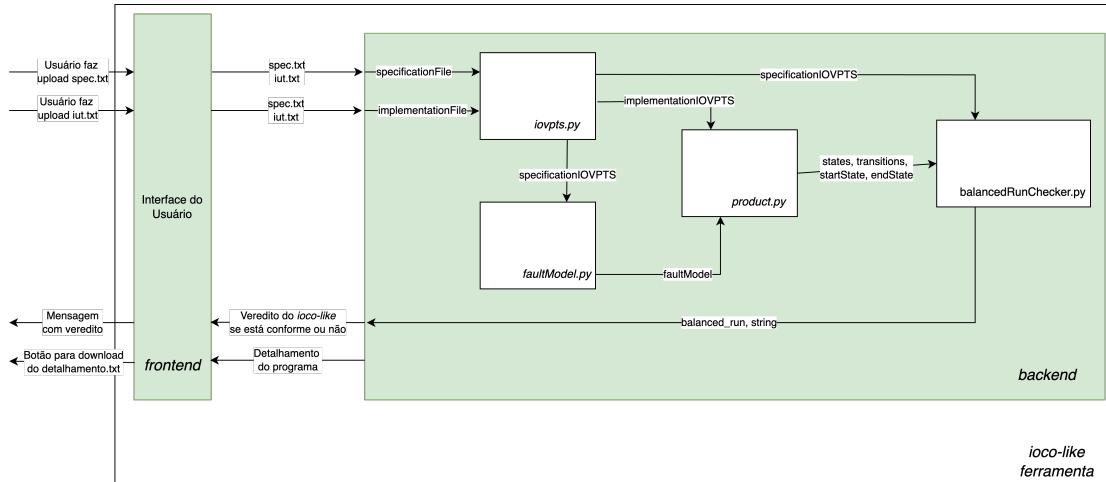


Figura 11 – Arquitetura da estrutura da ferramenta.

Fonte: Elaborado pela autora.

A seguir cada módulo é descrito em detalhe:

main: Este módulo é responsável pela entrada de dados da ferramenta e contém as chamadas para os módulos auxiliares. Entre as funções do *main* estão:

1. Leitura de Arquivos de Entrada: O script lê os modelos de especificação e implementação fornecidos pelo usuário. Esses modelos são transformados em objetos da classe *iovpts* para serem processados.
2. Instanciação e chamada das funções auxiliares:
 - Geração do Modelo de Falhas: A função responsável pela geração do modelo de falhas é chamada pela *main*, que, a partir da especificação do sistema, constrói o modelo que simula possíveis falhas ou desvios na execução.
 - Cálculo do Produto: A *main* invoca a função que calcula o produto entre o modelo de falhas e a implementação do sistema, necessário para o processo de verificação.
 - Verificação de Conformidade: A *main* também chama a função de verificação de conformidade entre a implementação e a especificação.

3. Interface com o Usuário: O módulo utiliza a biblioteca *Streamlit* para criar uma interface interativa, que permite ao usuário fornecer as entradas necessárias e visualizar os resultados do teste de conformidade.

iovpts: Este módulo define e gerencia os modelos construídos. A classe IOVPTS permite a definição dos componentes essenciais do sistema, incluindo símbolos de empilhamento e desencadeamento, símbolos internos, de entrada e saída, e estados, além de possibilitar a adição de transições entre estados. Também existe uma função “read iovpts file” que lê a configuração do IOVPTS a partir de um arquivo, interpretando e extraíndo os dados necessários para construir um objeto IOVPTS. A função processa o conteúdo do arquivo, configura os componentes do modelo e define o estado inicial com base nas informações fornecidas.

faultModel: Implementa a lógica para gerar o modelo de falhas com base na especificação do sistema. Para isso, inicialmente, adiciona um estado especial chamado “fail” e um símbolo de pilha “*” para representar situações de falha. Durante a geração do modelo, para cada estado do sistema original, o código verifica as transições e eventos que podem levar a esse estado. Quando detecta eventos que resultam em falhas, cria transições adicionais que levam ao estado “fail”. Quando existem eventos não manipulados nas transições, também são criadas transições que direcionam ao estado de falha. O modelo de falhas resultante é então exibido e pode ser salvo em uma lista para uso futuro.

product: Este módulo calcula o produto entre um modelo de falhas e uma implementação de IOVPTS. O produto essencialmente encontra a interseção entre as transições da implementação e do modelo de falhas, identificando os caminhos e comportamentos comuns. O método “compute_product” realiza a computação, analisando as transições dos modelos de falhas e implementação, e atualiza os conjuntos de estados e transições do produto. Além disso, o arquivo inclui funções para calcular e salvar informações detalhadas sobre estados, símbolos de pilha e transições do produto, bem como para converter essas informações em formatos utilizáveis e legíveis. As funções “compute_estados”, “compute_pilha” e “compute_transicoes” transformam e organizam as informações do produto, preparando-as para visualização ou armazenamento. Assim, posteriormente, os resultados do produto computado podem ser analisados para a verificação de conformidade.

balancedRunChecker: O módulo implementa um verificador de execuções balanceadas para autômatos visuais com pilha, inspirado na verificação de conformidade *ioco-like*. As transições do autômato são representadas por três estruturas principais: os vetores “In” e “Out”, e a matriz “R”.

- O vetor “In” armazena as transições que ocorrem quando um símbolo específico é empilhado na pilha, enquanto o vetor “Out” contém as transições associadas ao desempilhamento de símbolos. Essas estruturas ajudam a rastrear as transições associadas a cada estado e símbolo da pilha.
- A matriz “R” é usada para registrar as transições possíveis entre pares de estados, considerando tanto as transições internas quanto as transições que envolvem empilhamento e desempilhamento de símbolos. A matriz então permite que seja verificado se há uma sequência balanceada de transições entre dois estados específicos.

Essa função verifica se existe uma execução balanceada entre o estado inicial e o estado final do modelo resultante. Uma execução balanceada ocorre quando as operações de empilhar e desempilhar símbolos são realizadas corretamente, ou seja, cada transição de chamada (*push*) deve ter uma transição correspondente de retorno (*pop*), garantindo que as operações na pilha estejam corretamente emparelhadas. O módulo utiliza recursão para explorar as possíveis transições e identificar os caminhos que podem formar uma execução balanceada, retornando se uma execução balanceada foi encontrada ou se há uma discrepância no comportamento do sistema. Quando há alguma discrepancia nas transições *push/pop*, significa que um erro na implementação em relação à especificação foi identificado.

3.1.2 Bibliotecas Utilizadas

Diversas bibliotecas da linguagem Python foram utilizadas no desenvolvimento da ferramenta. Dentre as principais bibliotecas se destacam:

pandas: Facilita a manipulação de dados estruturados. Foi utilizada especialmente para processar as transições entre estados durante o cálculo do produto entre o modelo de falhas e a implementação. Especificamente, o pandas pode ser útil para organizar grandes quantidades de dados em DataFrames, o que permite uma interação eficiente ao manipular e verificar o conjunto de estados e transições. Essa funcionalidade facilita a verificação da conformidade ao lidar com múltiplos estados e transições de forma organizada e eficiente.

deepcopy: Realiza cópias profundas de estruturas complexas de dados, como listas de estados e transições. Isso é essencial para garantir que as alterações feitas em uma cópia da estrutura de dados não afetem o estado original. Em especial, no módulo faultModel, a função deepcopy é usada repetidamente para clonar os estados e as transições do sistema original (sem falhas) para construir o modelo de falhas, mantendo o modelo original intacto para comparação posterior.

logging: Registra os eventos durante a execução do programa, facilitando o processo de depuração e análise dos resultados. O uso do logging facilita a identificação de erros e o rastreamento do fluxo do programa, especialmente quando há várias transições e interações entre estados. No módulo balancedRunChecker, o logging auxilia no monitoramento da verificação de execuções balanceadas, permitindo acompanhar o progresso das verificações e identificar problemas com mais facilidade.

streamlit: Cria uma interface gráfica interativa que facilita a interação entre o usuário e a ferramenta. Essa interface foi projetada para tornar a interação com a ferramenta simples e acessível, oferecendo um design intuitivo que inclui título, subtítulo, instruções, área para upload de arquivos, botões de ação e mensagens de feedback. Assim, através dessa interface, o usuário pode fornecer arquivos de entrada, visualizar o progresso das verificações e receber o resultado dos testes de conformidade. O uso do Streamlit transforma o processo de verificação em algo mais acessível, permitindo que usuários com menos experiência em linha de comando possam interagir com a ferramenta de maneira eficiente.

3.1.3 Preparação do Ambiente para Execução da Ferramenta

Para garantir que a ferramenta seja executada corretamente em seu ambiente local, siga os passos detalhados abaixo para configurar o ambiente ou no macOS ou no Windows. As bibliotecas necessárias são instaladas utilizando o gerenciador de pacotes `pip`, que acompanha a instalação do Python.

1. Verifique a versão do Python e do `pip`.

- a) No macOS, abra o Terminal e execute:

- `python3 -version`
- `pip3 -version`

- b) No Windows, abra o Prompt de Comando e execute:

- `python -version`
- `pip -version`

Se os comandos retornarem as versões, o Python está instalado. Verifique se a versão é superior a 3.9; caso contrário, será necessário atualizar. Para atualização ou instalação, siga as instruções disponíveis no site oficial ¹.

No Windows, certifique-se de selecionar a opção “Add Python to PATH” durante a instalação para garantir que o Python seja reconhecido em qualquer diretório.

No MacOS, a instalação padrão geralmente já inclui o Python no PATH.

¹ Site Oficial: <https://www.python.org/downloads/>

2. Instale as bibliotecas necessárias:

a) No macOS, no Terminal, execute:

- pip3 install pandas
- pip3 install streamlit

b) No Windows, no Prompt de Comando, execute:

- pip install pandas
- pip install streamlit

Depois que as bibliotecas necessárias estiverem instaladas, siga os seguintes passos para executar a ferramenta:

1. Abra o Terminal (macOS) ou o Prompt de Comando (Windows).
2. Navegue até o diretório onde o arquivo `main.py` está localizado.
3. Execute o seguinte comando para iniciar a ferramenta utilizando o Streamlit: `streamlit run main.py`

3.2 A Ferramenta Prática

A interface gráfica da ferramenta, apresentada na Figura 12, permite ao usuário realizar o *upload* de dois modelos, acessar um menu de ajuda e utilizar um botão para verificar conformidade.

The screenshot shows a web-based application titled "ioco-like". At the top, there is a dropdown menu labeled "Ajuda". Below the title, a message reads: "Por favor, faça o upload dos arquivos de especificação e implementação em formato .txt. Ambos os arquivos são necessários para verificar a conformidade ioco-like." Two file upload fields are present: one for "Especificação" and one for "Implementação", both with a "Drag and drop file here" placeholder and a "Browse files" button. At the bottom, a large blue button labeled "Verificar Conformidade" is visible.

Figura 12 – Upload de arquivos na ferramenta.
Fonte: Elaborado pela autora.

Já a Figura 13 mostra dois exemplos de modelos IOVPTs que o usuário deve fornecer para a verificação de conformidade. A descrição detalhada desses arquivos encontra-se na Secção 3.2.1.

specificationIOVPTS.txt	implementationIOVPTS.txt
a,d,g,j	a,d,g,j
b,e,h,k	b,e,h,k
c,f,i,l	c,f,i,l
a,d,g,j	a,d,g,j
b,c,e,f,h,i,k,l	b,c,e,f,h,i,k,l
Z,X,Y,W	Z,X,Y,W
q0,q1,q2,q3,q4,q5,q6,q7,q8,q9	p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11
q0,a,Z,q1	p0,a,Z,p1
q1,b,Z,q2	p1,b,Z,p2
q2,c,Z,q3	p2,c,Z,p3
q3,d,X,q4	p3,d,X,p10
q4,e,X,q5	p10,e,X,p5
q5,f,Y,q6	p5,f,Y,p11
q6,g,Y,q7	p11,g,Y,p6
q7,h,Y,q8	p6,h,Y,p0
q8,i,W,q9	p2,g,X,p7
q9,j,W,q0	p7,h,Y,p8
q2,g,X,q5	p8,i,W,p3
q4,i,Z,q3	p3,k,W,p6
q7,k,W,q6	p4,l,Z,p9
#	#
q0	p0
-	-

Figura 13 – Arquivos .txt para especificação dos modelos IOVPTs.

Fonte: Elaborado pela autora.

Após carregar os arquivos, o usuário deve clicar no botão “Verificar Conformidade” para a ferramenta processar as entradas. Caso o usuário tente prosseguir sem fornecer ambos os arquivos, um aviso de erro é exibido, como mostrado na Figura 14.

Por favor, carregue ambos os arquivos de especificação e implementação.

Figura 14 – Erro devido à ausência de upload de arquivos na ferramenta.

Fonte: Elaborado pela autora.

Após o processo de verificação a ferramenta exibe o resultado ao usuário. Se for detectada uma não conformidade, a ferramenta gera um traço de falha que aponta onde a implementação diverge da especificação, permitindo uma análise detalhada dos erros encontrados, como mostrado na Figura 15. Caso o resultado indique conformidade, apenas uma mensagem informando o veredito é exibido, conforme ilustrado na Figura 16.

Em ambos os casos, seja de conformidade ou de não conformidade, a ferramenta disponibiliza um botão para que o usuário faça o download de um arquivo com o detalhamento completo da execução. A explicação sobre o conteúdo desse arquivo está disponível na Secção 3.2.2.

Testando o IUT descrito no arquivo IUTa.txt contra a especificação (SPEC) no arquivo iovpts-spec.txt

O IUT não está em conformidade com a especificação. Um caso de teste que mostra essa condição é:
cof,coi,deb,coi,dco

[Baixar Detalhamento Completo](#)

Figura 15 – Resultado de não conformidade entre a IUT e SPEC na ferramenta.

Fonte: Elaborado pela autora.

Testando o IUT descrito no arquivo implementationIOVPTS.txt contra a especificação (SPEC) no arquivo specificationIOVPTS.txt

O IUT está em conformidade ioco-like com a especificação.

[Baixar Detalhamento Completo](#)

Figura 16 – Resultado de conformidade entre a IUT e SPEC na ferramenta.

Fonte: Elaborado pela autora.

3.2.1 Arquivos de Entrada

A ferramenta recebe como entrada dois modelos IOVPTS: um representando a especificação e o outro a implementação do sistema a ser testado. A especificação descreve o comportamento esperado do sistema, enquanto a implementação representa o sistema sob teste. Ambos os modelos devem ser fornecidos em arquivos de texto no formato .txt, conforme a estrutura abaixo descrita. Essa mesma descrição dos arquivos está disponível no botão “Ajuda” da ferramenta.

1. **Linha 1:** Define as ações **CALL**, ações de chamada ou call actions, que empilham símbolos.
2. **Linha 2:** Define as ações **RETURN**, ações de retorno ou return actions, que desempilham símbolos.
3. **Linha 3:** Define as ações **INTERNAL**, ações internas ou internal actions, que não mexem na pilha.
4. **Linha 4:** Define as ações **INPUT**, eventos de entrada, representam eventos que o sistema recebe do ambiente externo.
5. **Linha 5:** Define as ações **OUTPUT**, eventos de saída, são eventos que o sistema gera como para o ambiente externo.

6. **Linha 6:** Define os **símbolos de pilha**, símbolos que podem ser empilhados e desempilhados pelo autômato.
 - a) O símbolo especial @ é utilizado para transições simples ou internas.
 - b) O símbolo * representa a base da pilha, ou o estado de pilha vazia.
7. **Linha 7:** Define os **estados do modelo**.
8. **Linhas 8 até x:** Definem as **transições** do modelo, onde cada transição é descrita na forma “s, a, Z, q”:
 - a) s: Estado de origem.
 - b) a: Ação realizada.
 - c) Z: Símbolo de pilha que está sendo manipulado.
 - d) q: Estado de destino.
9. **Linha (x+1):** Contém o símbolo # que indica o fim da descrição de transições.
10. **Linha (x+2):** Contém o **estado inicial** do modelo.
11. **Linha (x+3):** Contém o símbolo - para indicar o fim da descrição do modelo.

Na descrição dos modelos, vale ressaltar algumas observações importantes:

1. Sempre quando houver múltiplos itens (símbolos, estados, ações, etc) por linha, separe por vírgula.
2. O final de cada linha NÃO deve conter símbolos adicionais, tais como ponto, ponto e vírgula, etc.
3. Se um conjunto for vazio (por exemplo, o modelo não tem ações internas), uma linha em branco deve ser deixada na descrição.

A Figura 17 apresenta um exemplo de arquivo de entrada com a descrição de um modelo IOVPTS.

a,b,c	# CALLs
d,e,f	# RETURNS
x,y	# INTERNALs
i1,i2	# INPUTs
o1,o2	# OUTPUTs
A,B,C	# Símbolos de pilha
s0,s1,s2,s3	# Estados do modelo
s0,a,@,s1	# Transição
s1,d,A,s2	# Transição
s2,x,@,s3	# Transição
s3,i1,@,s0	# Transição
s0,o1,@,s3	# Transição
#	# Indica o estado inicial
s0	# Estado inicial
-	# Símbolo de fim de arquivo

Figura 17 – Arquivo .txt com modelo IOVPTS descrito.
Fonte: Elaborado pela autora.

3.2.2 Arquivo de Saída

A ferramenta gera um arquivo de saída “detalhamento.txt”, que contém uma descrição detalhada do processo de verificação de conformidade entre a implementação e a especificação. O arquivo detalha as transições do produto gerado entre o modelo de falhas e a implementação, listando cada estado composto, os símbolos envolvidos nas transições e o traço de falha, caso seja encontrado. Além disso, as transições e estados do produto são organizados utilizando índices e dicionários para simbolizar as pilhas e estados envolvidos, facilitando a análise do resultado. Por fim, o arquivo apresenta o veredito final, indicando se a IUT está em conformidade *iooco-like* com a especificação. Caso contrário, exibe o traço de falha detectado.

O objetivo de fornecer esse arquivo de saída detalhado ao usuário é permitir uma análise profunda do processo de verificação de conformidade. Com a descrição completa dos estados, transições e do modelo de falhas, o usuário pode compreender como a implementação se comporta em relação à especificação. Isso facilita a identificação de possíveis desvios e erros específicos, oferecendo uma visão clara de onde a implementação falha, quando for o caso. Além disso, o traço de falha gerado permite que o usuário investigue passo a passo a execução que levou a não conformidade, auxiliando na correção da implementação ou no ajuste da especificação.

4 UM ESTUDO DE CASO: MÁQUINA DE VENDAS

O estudo de caso apresentado nesta seção tem como objetivo realizar um teste de conceito da ferramenta desenvolvida. Além disso, outro objetivo não menos importante é resolver um problema clássico da literatura, porém com o foco voltado para aspectos reativos assíncronos do sistema e sua necessidade de memória para uma modelagem mais precisa. O estudo de caso prático abordado neste trabalho é de uma máquina automática de vendas de bebidas.

A máquina de vendas foi escolhida por sua natureza reativa assíncrona, podendo ser especificada através dos IOVPTSs. Além disso, sistemas de venda automática exigem gerenciamento de estados complexos e memória para lidar com diferentes entradas, tais como quantias em dinheiro e seleção de produtos, e com a produção de diferentes saídas, como a entrega de produtos ou devolução de troco.

A modelagem deste tipo de sistema com IOVPTS se torna adequada pois envolve tanto a partição de entradas e saídas quanto a necessidade de memória para controlar a quantidade de dinheiro inserido e realizar cálculos de troco. Este estudo de caso oferece uma oportunidade de demonstrar como o formalismo de IOVPTS pode ser aplicado para capturar com precisão o comportamento esperado do sistema, ao mesmo tempo que permite a verificação de conformidade.

4.1 Descrição do problema

O estudo de caso adotado está baseado num trabalho que lida com aspectos relacionados a verificação formal do sistema [22] e no estudo de caso apresentado no trabalho [21]. No entanto, o objetivo do primeiro trabalho era verificar propriedades usando lógica temporal em sistemas modelados com UML. Já o objetivo deste trabalho se assemelha mais ao do segundo estudo. Assim, o sistema objeto de estudo é uma máquina automática de vendas de bebidas e este trabalho pretende realizar a verificação de conformidade *ioco-like* em IUTs candidatas para uma especificação do sistema, utilizando modelos IOVPTS.

Neste projeto o problema foi adaptado para se ajustar ao contexto específico da análise realizada e para facilitar a implementação e validação do sistema. Por isso, o sistema foi simplificado para aceitar apenas um tipo de moeda. A funcionalidade de verificação de estoque também foi removida. Por outro lado, a opção de cancelamento do pedido a qualquer momento antes que a bebida seja selecionada foi adicionada a descrição.

O sistema da máquina automática de bebidas foi então projetado para aceitar moedas, fornecer troco quando necessário, e liberar a bebida selecionada. A máquina

oferece dois tipos de bebidas: chá, que custa 2 reais, e café, com o valor de 3 reais. Todas as transações são realizadas exclusivamente com moedas de 1 real.

Para obter uma bebida, o usuário deve inserir uma moeda por vez e escolher a bebida desejada. A máquina, assumindo que o estoque é ilimitado e as bebidas estão sempre disponíveis, entrega automaticamente o produto após a escolha. Em caso de cancelamento da compra antes da seleção da bebida, a máquina devolve integralmente o valor inserido. Além disso, se o usuário inserir mais moedas do que o necessário, a máquina calcula e entrega o troco corretamente.

Este estudo de caso tem como foco capturar a lógica das interações em tempo real entre o usuário e a máquina, incluindo o gerenciamento da inserção de moedas, a seleção de produtos, a devolução de troco e o cancelamento de operações. O formalismo IOVPTS é particularmente adequado para este cenário, uma vez que o modelo deve lidar com uma pilha visível para controlar as transições relacionadas à entrada de moedas e suas correspondentes ações na pilha, como empilhar créditos ou depurar o saldo ao entregar a bebida ou troco.

Portanto, o desafio deste estudo é garantir que uma implementação candidata do sistema reflita adequadamente as regras de negócios subjacentes, gerencie eficientemente a entrada de moedas e responda corretamente a diferentes ações do usuário, como cancelar a compra ou selecionar uma bebida, conforme a especificação fornecida.

4.2 Especificação da Máquina de Venda

A especificação IOVPTS para a máquina de vendas é apresentada na Figura 18.

Abaixo, apresentamos um complemento ao iovpts modelado anteriormente, no qual foi adicionada a funcionalidade de permitir ao usuário, caso a quantia inserida não seja suficiente, não apenas inserir mais moedas, mas também trocar a escolha de bebida ou cancelar a operação. Na Figura 19, é possível visualizar as transições que representam essas novas opções no sistema que levam de volta ao estado S2.

Na Tabela 1 também podemos ver o detalhamento das transições de estados que foram implementadas para acomodar as novas funcionalidades. Esta tabela ilustra como o sistema lida com diferentes ações do usuário, como a inserção de moedas adicionais, a escolha de bebidas, o cancelamento da operação e a devolução do troco.

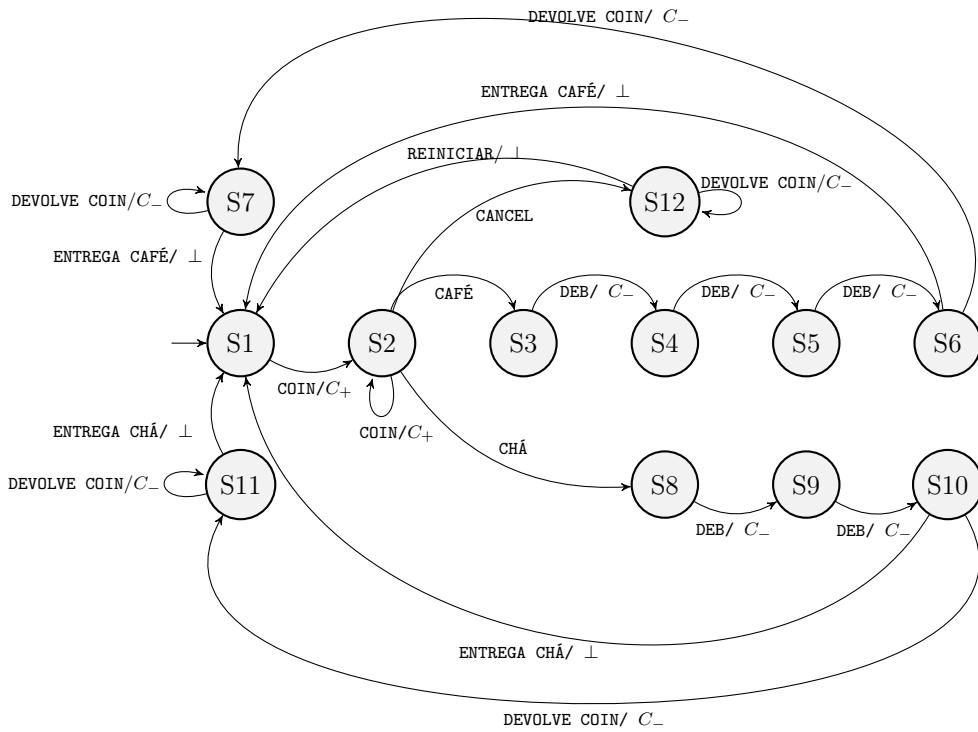


Figura 18 – iovpts da máquina de venda automática de chá e café.

Fonte: Elaborado pela autora.

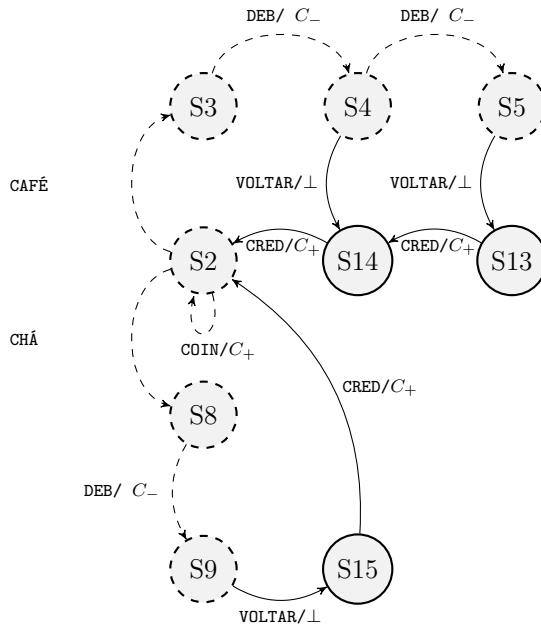


Figura 19 – Complemento do iovpts da máquina de venda automática de chá e café.

Fonte: Elaborado pela autora.

A descrição detalhada de cada componente da especificação IOVPTS é dada a seguir.

1. Estados - A máquina de vendas pode ser representada por uma série de estados que refletem seu comportamento em diferentes momentos da interação com o usuário:

- **S1:** Estado inicial, onde a máquina está em standby. Para sair desse estado, o usuário deve inserir uma moeda.
- **S2:** Estado onde a máquina acumula o valor das moedas inseridas, empilhando-as. Após esse estágio, o usuário pode optar por escolher uma bebida (chá ou café) ou cancelar a operação.
- **S3, S4, S5:** Estados onde a máquina debita o valor acumulado para verificar se há moedas suficientes para a compra de café.
- **S6:** Estado em que a máquina entrega o café e retorna ao estado inicial ou começa a devolver o troco, caso necessário.
- **S7:** Estado onde a máquina realiza a devolução do troco e, em seguida, entrega o café, retornando ao estado inicial.
- **S8, S9:** Estados onde a máquina debita o valor acumulado para verificar se há moedas suficientes para a compra de chá.
- **S10:** Estado em que a máquina entrega o chá e volta ao estado inicial ou inicia a devolução do troco.
- **S11:** Estado onde a máquina realiza a devolução do troco e, em seguida, entrega o chá, retornando ao estado inicial.
- **S12:** Estado onde a máquina devolve o valor total inserido, pois a operação foi cancelada pelo usuário.
- **S13, S14:** Estados onde a máquina devolve o valor inserido até o momento, pois o usuário não inseriu moedas suficientes para a compra do café.
- **S15:** Estado onde a máquina devolve o valor inserido até o momento, pois o usuário não inseriu moedas suficientes para a compra do chá.

2. Transições - As transições representam as mudanças entre estados, disparadas por ações do usuário ou da máquina (sistema):

- **T0 ($S1 \rightarrow S2$):** O usuário insere uma moeda para iniciar (coin/C_+).
- **T1 ($S2 \rightarrow S2$):** O usuário continua inserindo moedas (coin/C_+).
- **T2 ($S2 \rightarrow S3$):** O usuário escolhe café (café), e a máquina começa a debitar o valor necessário.
- **T3 ($S3 \rightarrow S4, S4 \rightarrow S5, S5 \rightarrow S6$):** A máquina debita o valor acumulado (deb/C_-) até que o montante suficiente seja atingido para a compra do café.
- **T4 ($S6 \rightarrow S1$):** A máquina entrega o café ($\text{entrega café}/\perp$).

- **T5 ($S_6 \rightarrow S_7$, $S_7 \rightarrow S_7$)**: A máquina devolve as moedas que não foram debitadas (troco).
- **T6 ($S_7 \rightarrow S_1$)**: A máquina retorna ao estado inicial após finalizar a entrega do café (*entrega café/⊥*).
- **T8 ($S_2 \rightarrow S_8$)**: O usuário escolhe chá (*chá*), e o máquina inicia o processo de débito.
- **T9 ($S_8 \rightarrow S_9$, $S_9 \rightarrow S_{10}$)**: A máquina debita o valor acumulado (*deb/C₋*) até que o montante seja suficiente para o chá.
- **T10 ($S_{10} \rightarrow S_1$)**: A máquina entrega o chá (*entrega chá/⊥*).
- **T11 ($S_{10} \rightarrow S_{11}$, $S_{11} \rightarrow S_{11}$)**: A máquina devolve as moedas restantes (troco).
- **T12 ($S_{11} \rightarrow S_1$)**: A máquina retorna ao estado inicial após a entrega do chá (*entrega chá/⊥*).
- **T13 ($S_2 \rightarrow S_{12}$)**: O usuário escolhe cancelar (*cancel*), e a máquina inicia o retorno do valor inserido.
- **T14 ($S_{12} \rightarrow S_{12}$)**: A máquina devolve todas as moedas inseridas.
- **T15 ($S_{12} \rightarrow S_1$)**: A máquina verifica se a pilha está vazia e retorna ao estado inicial.
- **T16 ($S_5 \rightarrow S_{13}$)**: A máquina verifica se a pilha está vazia e transita para um estado de crédito, para refazer o débito anterior.
- **T17 ($S_{13} \rightarrow S_{14}$)**: A máquina credita o valor que havia sido debitado na verificação de preço (*cred/C₊*).
- **T18 ($S_{14} \rightarrow S_2$)**: O valor debitado anteriormente é creditado novamente (*cred/C₊*).
- **T19 ($S_9 \rightarrow S_{15}$)**: A máquina verifica se a pilha está vazia e transita para um estado de crédito para desfazer o débito.
- **T20 ($S_{15} \rightarrow S_2$)**: O valor debitado é creditado novamente (*cred/C₊*).
- **T21 ($S_4 \rightarrow S_{14}$)**: A máquina verifica se a pilha está vazia e transita para um estado de crédito, para refazer o débito anterior.

3. Ações de Entrada - As ações de entrada são as interações do usuário com a máquina:

- **COIN**: O usuário insere uma moeda.
- **CAFÉ**: O usuário seleciona café.
- **CHÁ**: O usuário seleciona chá.
- **CANCEL**: O usuário cancela a transação.

4. Ações de Saída - As ações de saída representam as respostas da máquina:

- ENTREGA CHÁ: A máquina entrega a chá.
- ENTREGA CAFÉ: A máquina entrega a café.
- DEVOLVE COIN: A máquina devolve uma moeda.
- CRED: A máquina inclui o crédito que havia sido debitado.
- VOLTAR: A máquina volta para o estado de coleta de moedas ou cancelar ou escolher bebida.
- REINICIAR: A máquina volta para o estado inicial.
- DEB: A máquina desempilha moedas após a escolha da bebida.

5. Símbolos na Modelagem de IOVPTS

- **Símbolos de Chamada (“empilhar”):**

- COIN: A cada moeda inserida, o valor correspondente é adicionado à pilha de moedas acumuladas, representando o saldo total.
- CRED COIN: Quando é necessário desfazer uma operação (undo), o valor que havia sido debitado é creditado novamente, ou seja, a moeda é adicionada de volta à pilha de moedas acumuladas, restaurando o saldo anterior.

- **Símbolos de Retorno (“desempilhar”):**

- DEB: Após a escolha da bebida, a pilha de moedas é “desempilhada” para verificar se o valor inserido corresponde ao necessário para a compra.
- DEVOLVE COIN: Quando o usuário cancela a transação ou insere mais moedas que o necessário, o valor acumulado na pilha é “desempilhado” para devolver o montante.

- **Símbolos internos:**

- CAFÉ: A escolha do café é uma ação neutra, não afetando diretamente a pilha de moedas, já que não envolve cálculo monetário imediato.
- CHÁ: A escolha do chá também é uma ação neutra, sem impacto direto na pilha, pois não há envolvimento imediato de valores monetários.
- ENTREGA CAFÉ: A entrega do café não altera a pilha; é uma ação que finaliza a transação, sem modificar o valor acumulado.
- ENTREGA CHÁ: A entrega do chá, assim como a do café, não afeta a pilha de moedas e marca o fim da transação.
- VOLTAR: Um evento que volta para o estado anterior.
- REINICIAR: Um evento que volta para o estado inicial.

Estado Atual	Ação de Entrada	Símbolo	Próximo Estado	Ação de Saída
S1	COIN	C_+	S2	-
S2	COIN	C_+	S2	-
S2	CAFÉ	-	S3	-
S3	-	C_-	S4	DEB
S4	-	C_-	S5	DEB
S5	-	C_-	S6	DEB
S6	-	\perp	S1	ENTREGA CAFÉ
S6	-	C_-	S7	DEVOLVE COIN
S7	-	C_-	S7	DEVOLVE COIN
S7	-	\perp	S1	ENTREGA CAFÉ
S8	CHÁ	-	S9	-
S9	-	C_-	S10	DEB
S10	-	\perp	S1	ENTREGA CHÁ
S10	-	C_-	S11	DEVOLVE COIN
S11	-	C_-	S11	DEVOLVE COIN
S11	-	\perp	S1	ENTREGA CHÁ
S2	CANCEL	-	S12	-
S12	-	C_-	S12	DEVOLVE COIN
S12	-	\perp	S1	REINICIAR
S5	-	\perp	S13	VOLTAR
S13	-	C_+	S14	CRED
S14	-	C_+	S2	CRED
S9	-	\perp	S15	VOLTAR
S15	-	C_+	S2	CRED

Tabela 1 – Tabela de Estados e Transições

4.3 Aplicação da Ferramenta

Esta seção apresenta a verificação de conformidade entre Unidades em Teste (IUTs) candidatas, utilizando a ferramenta desenvolvida. O foco será a aplicação prática da ferramenta no estudo de caso da máquina de vendas, conforme descrito anteriormente.

O estudo de caso foi utilizado para verificar a conformidade entre a especificação e a implementação por meio da ferramenta desenvolvida. O processo é dividido em três etapas principais:

- Modelagem dos arquivos de entrada:** Nesta etapa, são criados os arquivos de especificação e implementação conforme o modelo IOVPTS, preparando-os para a verificação.
- Execução da verificação de conformidade:** Após o carregamento dos arquivos na ferramenta, realiza-se a verificação para avaliar se a implementação está em conformidade com a especificação proposta.
- Análise dos resultados gerados:** Os resultados da verificação são analisados para identificar quaisquer discrepâncias ou falhas de conformidade, permitindo uma compreensão clara da eficácia da ferramenta.

Através dessa aplicação, será possível ilustrar o funcionamento da ferramenta na detecção de discrepâncias e falhas de conformidade, proporcionando uma visão clara de seu funcionamento e dos resultados obtidos.

4.3.1 Modelagem dos arquivos de entrada

O primeiro passo é a modelagem dos arquivos de entrada, conforme descrito na Seção 3.2.1. Esses arquivos representam a especificação e a implementação do sistema, contendo todas as ações, estados, transições e símbolos relevantes para o modelo.

Modelo de Especificação

Abaixo, na Figura 20 é apresentado o arquivo de entrada utilizado como especificação.

```

COIN,CRED
RETURNCOIN,DEB
COFFEE,TEA,RETURNCOFFEE,RETURNTREA,CANCEL,VOLTAR,REINICIAR
COIN,COFFEE,TEA,CANCEL
RETURNCOFFEE,RETURNTREA,RETURNCOIN,CRED,VOLTAR,REINICIAR
C,@,*
S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15
S1,COIN,C,S2
S2,COIN,C,S2
S2,COFFEE,@,S3
S3,DEB,C,S4
S4,DEB,C,S5
S5,DEB,C,S6
S6,RETURNCOFFEE,*,S1
S6,RETURNCOIN,C,S7
S7,RETURNCOIN,C,S7
S7,RETURNCOFFEE,*,S1
S2,TEA,@,S8
S8,DEB,C,S9
S9,DEB,C,S10
S10,RETURNTREA,*,S1
S10,RETURNCOIN,C,S11
S11,RETURNCOIN,C,S11
S11,RETURNTREA,*,S1
S2,CANCEL,@,S12
S12,RETURNCOIN,C,S12
S12,REINICIAR,*,S1
S5,VOLTAR,*,S13
S13,CRED,C,S14
S14,CRED,C,S2
S9,VOLTAR,*,S15
S15,CRED,*,S2
S4,VOLTAR,*,S14
#
S1
-

```

Figura 20 – specification.txt

Modelos de Implementações

Abaixo, nas Figuras 21a, 21b, 22a e 22b são apresentados dois arquivos de entrada utilizados como implementação.

```

COIN,CRED
RETURNCOIN,DEB
COFFEE,TEA,RETURNCOFFEE,RETURNTSEA,CANCEL,VOLTAR,REINICIAR
COIN,COFFEE,TEA,CANCEL
RETURNCOFFEE,RETURNTSEA,RETURNCOIN,CRED,VOLTAR,REINICIAR
C,@,*
S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15
S1,COIN,C,S2
S2,COIN,C,S2
S2,COFFEE,@,S3
S3,DEB,C,S4
S4,DEB,C,S5
S5,DEB,C,S6
S6,RETURNTSEA,*,S1
S6,RETURNCOIN,C,S7
S7,RETURNCOIN,C,S7
S7,RETURNTSEA,*,S1
S2,TEA,@,S8
S8,DEB,C,S9
S9,DEB,C,S10
S10,RETURNCOFFEE,*,S1
S10,RETURNCOIN,C,S11
S11,RETURNCOIN,C,S11
S11,RETURNCOFFEE,*,S1
S2,CANCEL,@,S12
S12,RETURNCOIN,C,S12
S12,REINICIAR,*,S1
S5,VOLTAR,*,S13
S13,CRED,C,S14
S14,CRED,C,S2
S9,VOLTAR,*,S15
S15,CRED,*,S2
S4,VOLTAR,*,S14
#
S1
-

```

```

COIN,CRED
RETURNCOIN,DEB
COFFEE,TEA,RETURNCOFFEE,RETURNTSEA,CANCEL,VOLTAR,REINICIAR
COIN,COFFEE,TEA,CANCEL
RETURNCOFFEE,RETURNTSEA,RETURNCOIN,CRED,VOLTAR,REINICIAR
C,@,*
S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15
S1,COIN,C,S2
S2,COIN,C,S2
S2,COFFEE,@,S3
S3,DEB,C,S4
S4,DEB,C,S6
S6,RETURNTSEA,*,S1
S6,RETURNCOFFEE,*,S1
S6,RETURNCOIN,C,S7
S7,RETURNCOIN,C,S7
S7,RETURNCOFFEE,*,S1
S2,TEA,@,S8
S8,DEB,C,S9
S9,DEB,C,S10
S10,RETURNTSEA,*,S1
S10,RETURNCOIN,C,S11
S11,RETURNCOIN,C,S11
S11,RETURNCOFFEE,*,S1
S2,CANCEL,@,S12
S12,RETURNCOIN,C,S12
S12,REINICIAR,*,S1
S13,CRED,C,S14
S14,CRED,C,S2
S9,VOLTAR,*,S15
S15,CRED,*,S2
S4,VOLTAR,*,S14
#
S1
-

```

(a) implementation1.txt

(b) implementation2.txt

Figura 21 – Implementações distintas lado a lado.

Fonte: Elaborado pela autora.

No exemplo da Figura 21a, a principal diferença entre os arquivos de especificação e implementação está na troca das ações RETURNCOFFEE e RETURNTSEA em determinados estados. Isso deve resultar em um comportamento diferente do esperado. Temos:

1. Especificação: A ação RETURNCOFFEE é acionada após a escolha do café. A ação RETURNTSEA é acionada após a escolha do chá.
2. Implementação: A ação RETURNCOFFEE é acionada após a escolha do chá. A ação RETURNTSEA é acionada após a escolha do café.

No exemplo da Figura 21b, a principal diferença entre os arquivos de especificação e implementação é a remoção do estado S5, o que altera o valor do café de 3 reais para 2 reais. Isso deve resultar em um comportamento diferente do esperado. Temos:

1. Especificação: Quando selecionado COFFEE passa pelas transições S3 → S4 → S5 → S6.
2. Implementação: Quando selecionado COFFEE passa pelas transições S3 → S4 → S6.

```

COIN,CRED
RETURNCOIN,DEB
COFFEE,TEA,RETURNCOFFEE,RETURNTSEA,CANCEL,VOLTAR,REINICIAR
COIN,COFFEE,TEA,CANCEL
RETURNCOFFEE,RETURNTSEA,RETURNCOIN,CRED,VOLTAR,REINICIAR
C,@,*  

S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15  

S1,COIN,C,S2  

S2,COIN,C,S2  

S2,COFFEE,@,S3  

S3,DEB,C,S4  

S4,DEB,C,S5  

S5,DEB,C,S6  

S6,RETURNCOFFEE,*,S1  

S6,RETURNCOIN,C,S7  

S7,RETURNCOFFEE,*,S1  

S2,TEA,@,S8  

S8,DEB,C,S9  

S9,DEB,C,S10  

S10,RETURNTSEA,*,S1  

S10,RETURNCOIN,C,S11  

S11,RETURNCOIN,C,S11  

S11,RETURNTSEA,*,S1  

S2,CANCEL,@,S12  

S12,RETURNCOIN,C,S12  

S12,REINICIAR,*,S1  

S5,VOLTAR,*,S13  

S13,CRED,C,S14  

S14,CRED,C,S2  

S9,VOLTAR,*,S15  

S15,CRED,*,S2  

S4,VOLTAR,*,S14  

#  

S1  

-

```

```

COIN,CRED
RETURNCOIN,DEB
COFFEE,TEA,RETURNCOFFEE,RETURNTSEA,CANCEL,VOLTAR,REINICIAR
COIN,COFFEE,TEA,CANCEL
RETURNCOFFEE,RETURNTSEA,RETURNCOIN,CRED,VOLTAR,REINICIAR
C,@,*  

S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15  

S1,COIN,C,S2  

S2,COIN,C,S2  

S2,COFFEE,@,S3  

S3,DEB,C,S4  

S4,DEB,C,S5  

S5,DEB,C,S6  

S6,RETURNCOFFEE,*,S1  

S6,RETURNCOIN,C,S7  

S7,RETURNCOIN,C,S7  

S7,REINICIAR,*,S1  

S2,TEA,@,S8  

S8,DEB,C,S9  

S9,DEB,C,S10  

S10,RETURNTSEA,*,S1  

S10,RETURNCOIN,C,S11  

S11,RETURNCOIN,C,S11  

S11,RETURNTSEA,*,S1  

S2,CANCEL,@,S12  

S12,RETURNCOIN,C,S12  

S12,REINICIAR,*,S1  

S5,VOLTAR,*,S13  

S13,CRED,C,S14  

S14,CRED,C,S2  

S9,VOLTAR,*,S15  

S15,CRED,*,S2  

S4,VOLTAR,*,S14  

#  

S1  

-

```

(a) implementation3.txt

(b) implementation4.txt

Figura 22 – Implementações distintas lado a lado.

Fonte: Elaborado pela autora.

No exemplo da Figura 22a, a principal diferença entre os arquivos de especificação e implementação é a remoção da transição de loop do estado S7, que tinha a função de devolver moedas de troco caso ainda houvesse moedas na pilha. Essa mudança pode resultar, dependendo da entrada, na devolução de menos moedas do que o devido. Temos:

1. Especificação: Quando selecionado COFFEE, debita o valor do café, retorna o troco correto e entrega o café.
2. Implementação: Quando selecionado COFFEE, debita o valor do café, retorna uma moeda e caso ainda tenha moedas na pilha a implementação trava, retornando vazio.

No exemplo da Figura 22b, a principal diferença entre os arquivos de especificação e implementação é que, após devolver as moedas referentes ao troco do café, o sistema não entrega o café, apenas reinicia a máquina. Isso deve resultar em um comportamento diferente do esperado. Temos:

1. Especificação: Quando selecionado COFFEE após retornar o troco a ação RETURNCOFFEE é acionada.
2. Implementação: Quando selecionado COFFEE após retornar o troco a ação REINICIAR é acionada.

4.3.2 Testando Conformidade

Após a modelagem dos arquivos de entrada, esses são carregados na ferramenta para a verificação de conformidade. O usuário pode selecionar os arquivos desejados, adicionar ambos (especificação e implementação), e clicar no botão “Verificar Conformidade”. A ferramenta executa a comparação entre a especificação e a implementação. A Figura 23 apresenta a execução do teste de conformidade, utilizando a ferramenta em conjunto com a especificação mostrada na Figura 20 e a implementação ilustrada na Figura 21a.

ioco-like

Verificação de Conformidade

Ajuda ▾

Por favor, faça o upload dos arquivos de especificação e implementação em formato `.txt`. Ambos os arquivos são necessários para verificar a conformidade ioco-like.

Escolha o arquivo de Especificação

Upload
Drag and drop file here
Browse files

specification.txt 0.7KB X

Escolha o arquivo de Implementação

Upload
Drag and drop file here
Browse files

implementation1.txt 0.7KB X

Especificação carregada: specification.txt

Implementação carregada: implementation1.txt

Verificar Conformidade

Testando o IUT descrito no arquivo `implementation1.txt` contra a especificação (SPEC) no arquivo `specification.txt`

O IUT não está em conformidade com a especificação. Um caso de teste que mostra essa condição é:
COIN,COIN,TEA,DEB,DEB,RETURNCOFFEE

Baixar Detalhamento Completo

Figura 23 – Execução do estudo de caso com a ferramenta de conformidade.
Fonte: Elaborado pela autora.

O mesmo procedimento foi realizado com os demais arquivos de entrada. Os resultados das quatro execuções são exibidos nas Figuras 24, 25, 26 e 27, que serão analisados na próxima subseção.

O IUT não está em conformidade com a especificação. Um caso de teste que mostra essa condição é: COIN,COIN,TEA,DEB,DEB,RETURNCOFFEE

Figura 24 – Execução da implementação 1 do estudo de caso com a ferramenta de conformidade.

Fonte: Elaborado pela autora.

O IUT não está em conformidade com a especificação. Um caso de teste que mostra essa condição é: COIN,COIN,COFFEE,DEB,DEB,RETURNCOFFEE

Figura 25 – Execução da implementação 2 do estudo de caso com a ferramenta de conformidade.

Fonte: Elaborado pela autora.

O IUT está em conformidade ioco-like com a especificação.

Figura 26 – Execução da implementação 3 do estudo de caso com a ferramenta de conformidade.

Fonte: Elaborado pela autora.

O IUT não está em conformidade com a especificação. Um caso de teste que mostra essa condição é: COIN,COIN,COIN,COIN,COFFEE,DEB,DEB,DEB,RETURNCOIN,REINICIAR

Figura 27 – Execução da implementação 4 do estudo de caso com a ferramenta de conformidade.

Fonte: Elaborado pela autora.

A ferramenta gera uma saída detalhada, onde são apontadas as diferenças de comportamento entre a implementação e a especificação, conforme as transições modeladas.

4.3.3 Análise dos Resultados Gerados

A ferramenta gera, conforme mencionado, uma saída detalhada do que ocorreu no programa em um arquivo chamado detalhamento.txt. Devido à extensão desse arquivo, explicaremos aqui apenas os resultados obtidos. Esses resultados são essenciais

para identificar e corrigir falhas na implementação, assegurando que o sistema esteja em conformidade com o comportamento especificado.

Implementação 1

O resultado final desse teste de conformidade foi:

Veredito: O IUT não está em conformidade com a especificação.

Um caso de teste que mostra essa condição é:

COIN, COIN, TEA, DEB, DEB, RETURNCOFFEE

A ferramenta detecta um rastro de execução que revela a divergência entre as ações RETURNCOFFEE e RETURNTEA em estados equivalentes.

- Na especificação, quando o usuário realiza as ações COIN, COIN, TEA, a saída gerada é DEB, DEB, RETURNTEA.
- Na implementação, ao executar as mesmas ações, o usuário recebe DEB, DEB, RETURNCOFFEE.

O *output* da implementação é RETURNTEA, enquanto o da especificação é RETURNCOFFEE. Observa-se que o *output* da implementação não é equivalente ao da especificação e também não está contido nele, o que demonstra a violação do comportamento *ioco-like*. O rastro possibilita a análise da transição em que o problema ocorreu. A análise detalhada dos rastros de execução torna-se crucial para corrigir desvios que estão comprometendo a funcionalidade do sistema.

- Especificação: t6: S6 – RETURNCOFFEE/ → S1
Indica que, ao atingir o estado S6, a ação esperada é RETURNCOFFEE.
- Implementação: t6: S6 – RETURNTEA/ → S1
Aqui, a implementação realiza uma troca de ação, levando a um comportamento inesperado.

Implementação 2

O resultado final desse teste de conformidade foi:

Veredito: O IUT não está em conformidade com a especificação.

Um caso de teste que mostra essa condição é:

COIN, COIN, COFFEE, DEB, DEB, RETURNCOFFEE

A ferramenta identifica um rastro de execução que evidencia a divergência de comportamentos em estados equivalentes. Esse rastro permite a análise da transição em que ocorreu o problema. Especificamente:

- Na especificação, quando o usuário realiza as ações COIN, COIN, COFFEE, a saída é DEB, DEB, e então, a máquina trava, retornando vazio.
- Na implementação, ao realizar as mesmas ações, o usuário obtém DEB, DEB, RETURNCOFFEE.

Dessa forma, como não está contido em vazio, podemos afirmar que o *output* da implementação não está contido no *output* da especificação. O *output* da implementação RETURNCOFFEE não é equivalente e não está contido no da especificação “vazio”, resultando em uma condição de não conformidade.

Implementação 3

O resultado final desse teste de conformidade foi:

Veredito: O IUT está em conformidade com a especificação.

Nesse caso, intuitivamente, percebemos uma falta de conformidade, já que o comportamento da implementação difere do da especificação. Contudo, ao analisar um rastro, que poderia comprovar uma inconformidade, podemos avaliar se trata de uma inconformidade *ioco-like* ou apenas de uma diferença de comportamento. Abaixo, apresentamos um exemplo de um possível rastro que ilustra essa situação:

- Na especificação, quando o usuário executa as ações COIN, COIN, COIN, COIN, COIN, COFFEE, a resposta da máquina é DEB, DEB, DEB, RETURNCOIN, RETURNCOIN, RETURNCOFFEE.
- Na implementação, ao realizar as mesmas ações, o usuário obtém DEB, DEB, DEB, RETURNCOIN, e a máquina trava, retornando “vazio”.

Dessa forma, o *output* da implementação, que é “vazio”, está contido no *output* da especificação, que é RETURNCOFFEE. Por essa razão, mesmo que os comportamentos sejam distintos, eles são considerados em conformidade segundo a abordagem *ioco-like*.

É importante ressaltar que, ao executar a própria especificação como unidade em teste (IUT), também não foi identificado nenhum erro, o que confirma a validade do modelo de especificação. Além disso, alterações menores na implementação não resultaram em falhas, demonstrando que o teste é sensível apenas a desvios significativos no comportamento esperado.

Implementação 4

O resultado final desse teste de conformidade foi:

Veredito: O IUT não está em conformidade com a especificação.

Um caso de teste que mostra essa condição é:

COIN,COIN,COIN,COIN,COFFEE,DEB,DEB,DEB,RETURNCOIN,REINICIAR

Ao analisar o rastro de execução, podemos utilizá-lo novamente para comparar as respostas da máquina da especificação e da implementação.

- Na especificação, quando o usuário realiza as ações COIN, COIN, COIN, COIN, COFFEE, a saída gerada é DEB, DEB, DEB, RETURNCOIN, RETURNCOFFEE.
- Na implementação, ao executar as mesmas ações, o usuário recebe DEB, DEB, DEB, RETURNCOIN, REINICIAR.

O *output* da implementação é REINICIAR, enquanto o da especificação é RETURNCOFFEE. Nota-se que o *output* da implementação não é equivalente ao da especificação e também não está contido nele, o que evidencia a violação do comportamento *ioco-like*. Portanto, a implementação não é *ioco-like* em relação à especificação.

4.4 Resultado

A aplicação da ferramenta de verificação de conformidade nas diferentes implementações do estudo de caso da máquina de vendas permitiu identificar de maneira precisa a conformidade ou não conformidade *ioco-like* entre a especificação e as diversas implementações.

Para as implementações 1, 2 e 4, a ferramenta identificou de maneira clara e consistente as discrepâncias entre a especificação e as implementações modeladas. Essas divergências foram devidamente registradas no arquivo de saída gerado, confirmando a violação da conformidade *ioco-like*. Esse resultado não apenas atesta a eficácia da ferramenta, mas também ressalta sua capacidade de realizar diagnósticos detalhados, que são essenciais para a avaliação e correção de comportamentos que não estão em conformidade.

No que diz respeito à implementação 3, embora não tenha sido identificada a não conformidade, o processo foi igualmente documentado no arquivo de saída, incluindo o veredito de conformidade. Essa abordagem sistemática assegura que mesmo as implementações que atendem aos critérios da especificação sejam devidamente validadas conforme a conformidade *ioco-like*.

Os resultados obtidos ressaltam a eficácia da ferramenta na identificação de comportamentos incorretos em implementações modeladas com IOVPTS. Isso evidencia a habilidade da ferramenta em testar a conformidade em sistemas reativos que utilizam memória e pilha, sublinhando a importância de uma verificação rigorosa entre a especificação e a implementação.

Adicionalmente, o estudo de caso ilustra a aplicação prática da ferramenta, que pode ser utilizada em outros contextos para validar a correta implementação de sistemas baseados em modelos. Dessa forma, a ferramenta se apresenta como uma solução prática e eficiente para a verificação de conformidade, contribuindo significativamente para o desenvolvimento de sistemas mais robustos e confiáveis.

5 CONCLUSÃO

Este trabalho teve como principal objetivo desenvolver e aplicar uma abordagem de verificação de conformidade baseada em modelos IOVPTS para sistemas reativos com memória de pilha, utilizando a relação *ioco-like*. Uma ferramenta prática foi então implementada para testar a conformidade entre uma implementação e sua respectiva especificação.

O desenvolvimento da ferramenta englobou a construção de vários módulos para lidar com a criação de modelos formais, o produto de modelos, bem como o processo de verificação de execuções balanceadas. Assim, através da modelagem formal com operações de empilhamento e desempilhamento, foi possível testar e validar diferentes comportamentos dos sistemas, garantindo que o comportamento da implementação estivesse em conformidade com a especificação proposta.

A etapa de verificação de execuções balanceadas foi um aspecto central do processo, permitindo detectar situações em que operações de empilhamento e desempilhamento ocorriam de maneira incorreta. Esse aspecto característico em sistemas reativos assíncronos que exigem memória auxiliar, possibilitou que potenciais falhas fossem detectadas e corrigidas de forma antecipada, contribuindo para a melhoria da qualidade e confiabilidade do sistema sob teste.

Assim, este trabalho oferece uma contribuição significativa para a área de verificação formal de sistemas reativos com memória de pilha. O formalismo de IOVPTS provou ser essencial para criar uma base robusta de verificação, e a ferramenta desenvolvida se mostrou uma solução eficiente para a detecção de falhas de conformidade. Com aprimoramentos futuros, espera-se que essa ferramenta possa ser aplicada a um escopo mais amplo de sistemas, promovendo maior segurança e confiabilidade em sistemas reativos complexos.

Além dos módulos que lidam com a modelagem e verificação, uma contribuição importante foi o desenvolvimento de uma interface intuitiva e interativa. A interface permite que os usuários carreguem os arquivos de especificação e implementação, realizem a verificação de conformidade com facilidade e visualizem os resultados de maneira clara. Essa usabilidade estende o alcance da ferramenta, tornando-a acessível a pessoas com diferentes níveis de conhecimento técnico.

Contudo, a abordagem desenvolvida também apresentou algumas limitações. A complexidade computacional do processo de verificação aumenta significativamente à medida que o tamanho e a complexidade da implementação e da especificação crescem. Além disso, a ferramenta poderia ser aprimorada para lidar com cenários de teste mais realis-

tas e oferecer maior suporte a integrações com outros modelos de teste. Como trabalho futuro, sugere-se a aplicação de um estudo de caso mais realista e complexo, bem como a exploração de técnicas de otimização no cálculo de execuções balanceadas.

REFERÊNCIAS

- [1] TRETMANS, J. Model based testing with labelled transition systems. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 1–38, 2008. Disponível em: <https://doi.org/10.1007/978-3-540-78917-8_1>.
- [2] BONIFÁCIO, A. L.; MOURA, A. V. Test suite completeness and black box testing. *Software Testing, Verification and Reliability*, v. 27, n. 1-2, p. e1626, 2017. E1626 stvr.1626. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1626>>.
- [3] BONIFACIO, A. L.; MOURA, A. V. Conformance checking and pushdown reactive systems. *CLEI Electronic Journal*, v. 25, n. 2, March 2023. ISSN 0717-5000. Disponível em: <<https://doi.org/10.19153/cleiej.25.3.2>>.
- [4] THOMPSON, M. *Reactive Systems: Principles and Patterns*. Cambridge, UK: Cambridge University Press, 2020. ISBN 978-1-108-48400-2.
- [5] LAPLANTE, P. A.; OVASKA, S. J. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. 4th. ed. Hoboken, NJ: Wiley, 2017. ISBN 978-1-118-99000-5.
- [6] PRESSMAN, R. S. *Engenharia de Software: Uma Abordagem Profissional*. 7th. ed. Porto Alegre, Brasil: AMGH Editora Ltda, 2011. ISBN 978-85-364-2560-2.
- [7] YENIGUN, H.; YILMAZ, C.; ULRICH, A. Advances in test generation for testing software and systems. *International Journal on Software Tools for Technology Transfer (STTT)*, Springer, v. 18, n. 3, 2016. ISSN 1433-2779. Disponível em: <<https://doi.org/10.1007/s10009-016-0394-9>>.
- [8] TRETMANS, J. *Testing Techniques: Model-Based Testing with Labelled Transition Systems*. Berlin, Heidelberg: Springer, 2004. ISBN 978-3-540-78917-8. Disponível em: <https://doi.org/10.1007/978-3-540-78917-8_1>.
- [9] UTTING, M.; LEGEARD, B. *Practical Model-Based Testing: A Tools Approach*. 1st. ed. San Francisco, CA: Morgan Kaufmann Publishers, 2007. ISBN 978-0-12-372501-1. Disponível em: <<https://www.elsevier.com/books/practical-model-based-testing/utting/978-0-12-372501-1>>.
- [10] GOMES, C. S. Everest: Uma ferramenta para verificação de conformidade e geração de testes para modelos reativos. *Dissertação de Mestrado, Universidade Estadual de Londrina*, Londrina, Brasil, 2020.
- [11] KHAN, M. U. et al. Empirical studies omit reporting necessary details: A systematic literature review of reporting quality in model based testing. *Computer Standards and Interfaces*, v. 57, p. 35–50, 2018. Disponível em: <<https://doi.org/10.1016/j.csi.2018.07.005>>.
- [12] BROY, M. et al. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Berlin, Heidelberg: Springer-Verlag, 2005. (Lecture Notes in Computer Science). ISBN 978-3-540-26054-4.

- [13] BONIFACIO, A. L.; MOURA, A. V.; SIMAO, A. da S. A generalized model-based test generation method.
- [14] BONIFACIO, A. L.; MOURA, A. V. Testing asynchronous reactive systems: Beyond the ioco framework. *CLEI Electronic Journal*, v. 24, n. 13, July 2021. ISSN 0717-5000. Disponível em: <<https://doi.org/10.19153/cleiej.24.2.13>>.
- [15] SIPSER, M. *Introduction to the Theory of Computation*. 3rd. ed. Boston, MA: Cengage Learning, 2012. ISBN 978-1-305-25343-2.
- [16] UTTING, M.; LEGEARD, B. *Model-Based Testing for Embedded Systems*. 1st. ed. Waltham, MA: Elsevier, 2012. ISBN 978-0-12-415831-1.
- [17] KULL, A.; UTTING, M.; LEGEARD, B. Model-based testing of reactive systems: An industry perspective. *Software Testing, Verification and Reliability*, v. 22, n. 3, 2012. Disponível em: <<https://doi.org/10.1002/stvr.456>>.
- [18] BONIFÁCIO, A. L.; NASCIMENTO, C. Extração de propósitos de teste para modelos reativos. *Revista de Informática Teórica e Aplicada*, v. 29, n. 2, 2022. ISSN 0103-4308. Disponível em: <http://www.uel.br/cce/dc/wp-content/uploads/ProjetoTCC_CAROLINE_P_G_DONATO_NASCIMENTO.pdf>.
- [19] BONIFÁCIO, A. L.; NASCIMENTO. Language-based testing for pushdown reactive systems. *Journal of Systems and Software*, v. 196, 2023. Disponível em: <<https://doi.org/10.1016/j.jss.2023.111670>>.
- [20] MYERS, G. J. *The Art of Software Testing*. 2nd. ed. Hoboken, NJ: John Wiley and Sons, 2004. ISBN 978-0-471-46912-9.
- [21] BONIFACIO, A. L.; MOURA, A. V. Conformance checking and pushdown reactive systems. *CLEI Electronic Journal*, v. 25, n. 3, November 2022. ISSN 0717-5000. Disponível em: <<https://doi.org/10.19153/cleiej.25.3.2>>.
- [22] ROYER, J.-C. *Temporal Logic Verifications for UML: the Vending Machine Example*. Nantes, France, 2001. Disponível em: <<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1d3ebe62bb4a981ccf49d0a5727e844104d84340>>.