Conformidade em Sistemas Reativos: Formalismos como Ferramenta para Garantir a Confiabilidade

Denise Figueiredo de Rezende[1], **Adilson Luiz Bonifácio**[1]

[1]Departamento de Computação – Universidade Estadual de Londrina (UEL)
Caixa Postal 10.011 – CEP 86057-970 – Londrina – PR – Brasil

denise.rezende@uel.br, bonifacio@uel.br

Abstract. *Technological advances have driven a significant transformation in several areas of our society, bringing countless benefits and opportunities. However, as technologies become increasingly integrated with our lives and in many critical sectors, it becomes crucial to guarantee the reliability of these systems. This includes everything from ensuring the privacy of information to preventing accidents caused by failures in traffic systems. In the face of these challenges, it has become necessary to seek solutions that ensure that the software testing process detects defects more accurately. One way to do this is through conformance testing and formal methods testing, which have proven promising for testing reactive systems. This work aims to study and improve conformance testing techniques to optimize and increase the efficiency of the testing process.*

Resumo. *Os avanços tecnológicos têm impulsionado uma transformação significativa em diversas áreas da nossa sociedade, trazendo 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. Isso inclui desde a garantia de privacidade de informações até a prevenção de acidentes por falhas de sistemas de trânsito. Diante desses desafios, tornou-se necessário buscar soluções que garantam que o processo de teste de software detecte falhas com mais precisão. Uma via para isso são testes baseados em conformidade e formalismos, os quais tem se provado promissores para testes de sistemas reativos. Este trabalho visa estudar e aprimorar técnicas de testes de conformidade para otimizar e aumentar a eficiência do processo de teste.*

SUMÁRIO

1 INTRODUÇÃO

Sistemas reativos, tanto de software quanto de hardware, são caracterizados pela interação contínua com o ambiente externo. Há inúmeros sistemas reativos que também são críticos. Sistemas críticos são caracterizados pelo significativo impacto na segurança, saúde, infraestrutura da vida em sociedade. Por isso, falhas na detecção de erros nesses sistemas durante a fase de testes pode 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 gravíssimas consequências. Como por exemplo, num sistema de controle de tráfego, onde a ocorrência de falhas pode acarretar em acidentes de trânsito; em sistemas que lidam com dados financeiros, onde falhas não identificados podem levar a exposição de informações confidenciais. Considerando a importância crucial da detecção de falhas em sistemas reativos críticos, fica evidente o desafio de aprimorar as abordagens de testes para tais sistemas.

Uma das abordagens usadas para sistemas reativos é a verificação de conformidade [1]. O objetivo é garantir que o comportamento de um sistema reativo esteja de acordo com o comportamento de sua respectiva especificação. Uma abordagem para verificação de conformidade em sistemas reativos usa a relação IOCO (Input/Output Conformance), cujo objetivo é verificar se as saídas geradas pelo sistema são aquelas esperadas na especificação. O formalismo usado para especificar tais sistemas é modelo IOLTS (Input Output Labeled Transition Systems), um sistema de transição rotulado com entradas e saídas, permitindo a análise formal e a verificação de conformidade sobre estes modelos.

O IOVPTS (Input/Output-Visible Pushdown Transition System), por sua vez, estende o IOLTS, para abranger uma classe de modelos mais complexa, usando uma memória auxiliar [2]. Essa complexidade adicionada pelo uso de uma memória de pilha também impacta consideravelmente na relação de conformidade adotada para estes modelos e por consequência na forma como a verificação de conformidade é realizada entre implementações e suas respectivas especificações.

Nessa linha, este trabalho visa aprimorar os métodos de verificação de conformidade para sistemas reativos bem como desenvolver ferramentas de apoio e a aplicação prática dessas técnicas em estudos de caso.

A parte primordial desse projeto é a fundamentação, na seção 2. Descreve a fundamentação teórica dos modelos mais conhecidos.

2 FUNDAMENTAÇÃO

Este capítulo apresenta uma visão abrangente dos fundamentos necessários para a compreensão de sistemas reativos e os diferentes métodos de teste de software, com um foco particular no teste baseado em modelos. A estrutura do capítulo está dividida em várias seções, cada uma abordando tópicos essenciais para uma base sólida. Abaixo, uma breve descrição das seções:

Na seção inicial 2.1, são introduzidos os sistemas reativos, que são sistemas que respondem a estímulos externos em tempo real. A seção define o que são esses sistemas e discute suas características e desafios específicos. Na seção seguinte 2.2, são abordadas as técnicas gerais de teste de software, explicando a importância dos testes no ciclo de desenvolvimento e manutenção de sistemas.

Posteriormente na seção 2.3, detalha-se os testes de software baseados em modelos, uma abordagem que utiliza modelos formais para gerar casos de teste e verificar a conformidade do sistema. Dentro desse aspecto aborda-se: Especificações Formais; Geração de Casos de Teste; Cobertura de Teste; Modelo de Falha; Verificação de Conformidade.

Na seção seguinte são apresentados diversos modelos formais utilizados no teste baseado em modelos, cada um com suas características e aplicações específicas. Respectivamente: LTS (Labeled Transition Systems); IOLTS (Input/Output Labeled Transition; VPTS (Visible Pushdown Transition System); IOVPTS (Input/Output-Visible Pushdown Transition System). Sendo o foco desse trabalho a conformidade em IOLTS e IOVPTS.

Na última seção da fundamentação, foca-se nos métodos para garantir que o sistema implementado esteja conforme as especificações do modelo, apresentando o conceito geral de conformidade em testes de software e detalhando a relação de conformidade IOCO, uma metodologia específica para verificação de conformidade em modelos IOLTS.

2.1 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 [1, 2]. 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 computacio-

nais, seja em soluções tecnológicas simples ou em aplicações industriais críticas. Em todos os lugares sistemas do mundo real estão sendo governados por comportamentos 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 [3]. 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 [4].

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 aplicados.

O processo de verificação tem o objetivo de garantir a corretude do sistema em desenvolvimento de acordo com as especificações. Já no processo de validação a ideia é garantir que o sistema seja construído de acordo com as necessidades do usuário. Em geral, a validação está associada com a implementação do software e envolve a atividade de teste (integração, sistema e aceitação), verificando 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.

2.2 Teste de Software

A atividade de teste é parte essencial do processo de desenvolvimento de software. Os testes tem como objetivo garantir a qualidade do software, identificar falhas e evitar problemas aos usuários. Existem diversos tipos de teste, cada um com um objetivo específico, tais como teste de estresse (reação do sistema a grandes volumes de dados), velocidade e performance, segurança, entre outros [5].

Cada tipo de teste utiliza diferentes níveis de abstração do sistema a ser testado. Testes de caixa branca concentram-se no funcionamento interno do sistema, onde a estrutura do sistema é conhecida. Essa abordagem permite testes precisos para identificação de falhas na lógica, estruturas de dados e algoritmos. Em contraste, testes de caixa preta focam no comportamento do sistema e suas funcionalidades, sem se preocupar com a implementação ou estrutura interna do software. Essa técnica é ideal quando o objetivo é avaliar a conformidade dos comportamentos exibidos pelo sistema, independente da sua implementação interna. Um tipo de teste que permite uma variação entre esse dois tipos de teste são os testes de caixa-cinza [5]. A figura 1 ilustra de forma concisa os diversos níveis de abstração de um sistema.

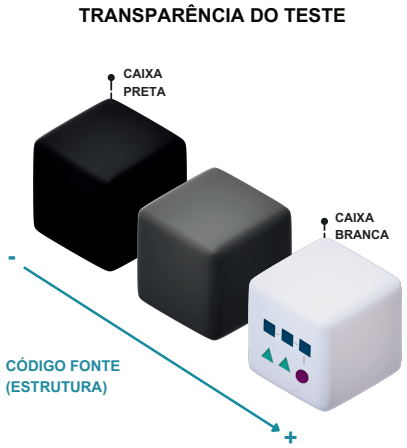

Figura 1 – Níveis de abstração do sistema

Um exemplo que usa a abordagem de caixa-cinza seria quando existe o acesso a algumas informações internas do sistema, como interfaces públicas de módulos ou documentação específica, mas não se tem acesso ao código-fonte completo. Nesse caso, é possível que se

explore o sistema além dos testes de caixa preta, usando um conhecimento parcial da implementação.

Em sistemas críticos reativos, como sistemas de controle de tráfego aéreo, testes usando abordagens rigorosas são indispensáveis para garantir a confiabilidade e segurança no desenvolvimento. A abordagem de teste baseado em modelos (MBT) tem sido então aplicada em sistemas dessa natureza, oferecendo rigor e formalismo, além de uma abordagem sistemática para identificar falhas e garantir o comportamento correto desses sistemas.

2.3 Teste Baseado em Modelos

O Teste Baseado em Modelo (MBT - Model-based testing) se destaca como uma abordagem rigorosa de teste ideal para superar os desafios inerentes ao teste de software em sistemas críticos reativos [6].

No MBT, o comportamento desejado de um sistema é capturado por algum modelo formal que serve de especificação e base para os testes sobre implementações candidatas. [6]. Ao descrever os comportamentos desejados de um sistema por meio de um modelo formal é possível garantir um rigor matemático, evitando inconsistências e ambiguidades, no processo de teste. A partir desses modelos, diferentes tipos de testes podem ser realizados, usando modelos específicos conforme aspectos de teste desejados.

2.3.1 Especificações formais

Na abordagem de teste usando modelos formais é preciso que algum tipo de modelo seja adotado para se 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 entender, comunicar e analisar sistemas complexos, facilitando a tomada de decisões e a resolução de problemas. Essa abstração permite 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 mais adequado depende das características específicas do sistema e dos objetivos do teste. Quanto menor o grau de formalismo de um modelo, 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 rigorosas e automatizadas, assegurando a consistência e confiabilidade do sistema em construção.

Um exemplo de modelo formal são as Máquinas de Estado Finitas (FSM - Finite State Machines), as quais representam o comportamento do sistema utilizando estados (diferentes situações em que o sistema pode se encontrar); transições (mudanças de estado em resposta a eventos) e ações (atividades executadas ao realizar uma transição). As FSM se destacam como um tipo de modelo formal amplamente utilizado no MBT, devido à sua simplicidade, expressividade e capacidade de representar comportamentos discretos.

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. Em uma FSM, um estímulo é representado como input, enquanto uma saída produzida é representada como output.

A FSM é definida formalmente por uma tupla [7] $\mathcal{A} = \langle S, s_0, A, \rho, F \rangle$, onde:

- S é um conjunto finito de estados;
- $s_0 \subseteq S$ é o estado inicial;
- A é um alfabeto finito não vazio;
- $\rho \subseteq S \times (A \cup \{\varepsilon\}) \times S$ é a relação de transação;
- $F \subseteq S$ é um conjunto de estados finais.

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 fazer a transição entre os estados de acordo com os eventos a e b.

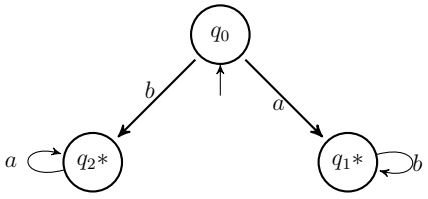

Figura 2 – Um FSM com 3 estados e 4 transições.

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 formal poderoso, elas não são a única opção. Outros

modelos, como IOLTS (Input/Output Labeled Transition Systems) e IOVPTS (Input/Output-Visible Pushdown Transition System), oferecem maior expressividade e flexibilidade para modelar sistemas complexos.

2.3.2 Geração de casos de teste

A geração de casos de teste é uma das atividades da abordagem MBT, crucial no processo de desenvolvimento de software, uma vez que bons conjuntos de teste gerados devem então ser aplicados a uma implementação sob teste (IUT – Implementation Under Test) para verificar a funcionalidade correta do sistema.

A produção de casos de teste utilizando MBT é fundamentada em modelos que retratam o comportamento previsto do sistema. Estes modelos podem ser elaborados através de vários formalismos, como os autômatos finitos, que descrevem o sistema como uma série de estados e transições. Existem também sistemas de transição que abrangem uma vasta gama de variações, como LTS, IOLTS, IOVPTS, com elementos temporais, simbólicos, entre outros.

Os casos de teste são gerados a partir dos modelos que representam o comportamento esperado do sistema, começando com a definição do escopo. Os especialistas em testes determinam quais aspectos do sistema serão testados. Para isso, o modelo é então analisado para identificar os diferentes caminhos de execução possíveis no sistema. Depois, a criação dos casos de teste requer a elaboração detalhada de cenários de teste que abordem diferentes fluxos de execução e condições.

2.3.3 Cobertura de teste

A cobertura de teste se refere à porcentagem do código ou funcionalidades do sistema que foram testadas por um conjunto de casos de teste. Aumenta a confiabilidade do sistema: Ao testar uma maior parte do código, é mais provável que falhas sejam identificadas e corrigidas, resultando em um sistema mais confiável. Duas características importantes para MBT são:

- Eficiência: A geração de casos de teste deve ser eficiente, ou seja, deve gerar o máximo possível de cobertura com o mínimo de testes possível.

- Eficácia: Os casos de teste gerados devem ser eficazes, ou seja, devem ser capazes de identificar falhas no sistema.

Diversos métodos de geração de casos de teste podem ser utilizados para alcançar uma boa cobertura de teste. Alguns dos métodos mais comuns incluem: Algoritmo de Dijkstra: Este algoritmo pode ser utilizado para gerar casos de teste que cobrem todos os caminhos

possíveis em um grafo e Exaustividade: É impossível testar todos os casos de uso possíveis de um sistema e garantir a ausência de falhas.

No entanto, é importante notar que a completa cobertura de teste não implica necessariamente a detecção de todas as possíveis falhas no sistema. Nesse contexto, podemos definir a completude na cobertura de teste como a extensão em que um conjunto de testes explora sistematicamente e de forma completa todos os valores de entrada válidos e inválidos, caminhos do fluxo de controle e estados possíveis do programa, para reduzir o risco de falhas não detectadas.

Observa-se que, à medida que se busca uma maior abrangência nos testes, ao se aproximar da completude de teste, o modelo utilizado tende a se tornar mais restrito, visando torná-lo mais controlável e/ou conhecido. No entanto, esse refinamento do modelo pode resultar em métodos mais complexos para sua geração. Por outro lado, quando o modelo é mais genérico, admitindo diferentes formas e comportamentos (parciais, não determinísticos, etc.), a geração de casos de teste torna-se mais desafiadora e a garantia de completude mais difícil de ser alcançada. Essa dinâmica entre a especificidade do modelo e a complexidade na geração de casos de teste evidencia a inter-relação entre completude e cobertura de sistemas.

2.3.4 Modelo de falha

Um modelo de falhas descreve os possíveis cenários de falha que podem ocorrer no sistema durante sua operação. 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 externos ou internos.

Os modelos de falhas permitem uma avaliação mais abrangente da capacidade do sistema de lidar com situações adversas e de se recuperar de falhas de forma adequada. Ao incorporar esses cenários de falha na geração de casos de teste é possível se verificar como o sistema reage em diferentes condições de falha 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.

2.3.5 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.

Dentro do contexto de completude e cobertura de sistemas, é importante ressaltar que a exigência de que o que está na implementação deve estar na especificação é unilateral, não bilateral. Enquanto essa diretriz estabelece a necessidade de consistência entre a implementação e a especificação, ela não impede que a implementação inclua funcionalidades adicionais. Assim, garante a integridade da especificação sem restringir a extensão das capacidades da implementação.

Na verificação de conformidade duas propriedades importantes podem ser garantidas, *soundness* (segurança) e completeness (completude). O sistema é dito ser *soundness* quando qualquer comportamento do sistema também é um comportamento encontrada na especificação. Logo, um sistema que é *soundness* nunca fará algo que não está descrito na especificação. Já o conceito de *completeness* garante que todo comportamento permitido pela especificação pode ser realizado pelo sistema. Logo, essa propriedade garante que a especificação capture completamente o comportamento desejado do sistema, sem omissões ou ambiguidades [8]

2.4 Modelos Formais

A fase de teste é uma etapa fundamental no processo do desenvolvimento de software. No caso de testes em sistemas reativos, existem alguns 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: os IOLTSs (Input Output Labeled Transition Systems) e os IOVPTSs (Input/Output-Visible Pushdown Transition Systems).

2.4.1 LTS

O LTS é um modelo formal utilizado para descrever o funcionamento/comportamento de sistemas. É composto por transições rotuladas com ações ou eventos que podem ocorrer no sistema, representando as operações ou mudanças que acontecem internamente. [7]

Em um LTS as transições entre estados não são rigidamente determinadas por eventos externos ou entradas específicas. Isso significa que a escolha de qual transição seguir a

partir de um determinado estado pode não depender exclusivamente de fatores externos, permitindo que múltiplas transições sejam possíveis para um mesmo evento. Dessa forma, as transições podem ocorrer de forma independente, com base em condições internas do sistema ou escolhas não determinísticas, oferecendo flexibilidade na representação de sistemas onde o comportamento não é completamente previsível ou sincronizado com eventos externos.

O LTS é definido formalmente por uma tupla $\mathcal{S} = \langle S, s_0, L, T \rangle$, onde: [7]

- S é um conjunto finito de estados;
- $s_0 \subseteq S$ é o estado inicial;
- L é um conjunto finito de rótulos, ou ações; $\tau \notin L$ é o símbolo da ação interna;
- $T \subseteq S \times L_\tau \times S$ é um conjunto de transições.

Exemplo:

Para o LTS abaixo, $\mathcal{S} = \langle S, s_0, L, T \rangle$, temos $S = \{s_0, s_1, s_2, s_3, s_4\}$, $L = \{b, c, t\}$ e setas que indicam transições.

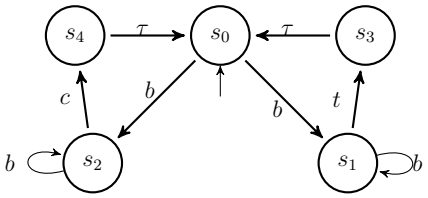

Figura 3 – Um LTS com 5 estados e 8 transições.

$$T = \{(s_0, b, s_1), (s_0, b, s_2), (s_1, b, s_1), (s_1, t, s_3)$$
$$(s_2, b, s_2), (s_2, c, s_4), (s_3, \tau, s_0), (s_4, \tau, s_0)\}$$

Neste exemplo, podemos observar claramente o não determinismo de um LTS. Por exemplo, a partir do estado s_0 com a entrada b, temos a opção de transitar tanto para o estado s_1 quanto para o estado s_2, o que demonstra que o comportamento do sistema não é totalmente previsível em todos os casos.

Além disso, percebemos que o sistema também possui transições internas, representadas pelo símbolo τ, que permitem retornar ao estado inicial a partir de estados específicos, como s_3 e s_4. Essas características de não determinismo e transições internas contribuem para a complexidade e adaptabilidade do sistema modelado pelo LTS.

2.4.2 IOLTS

O IOLTS é um formalismo matemático que descreve o comportamento de um sistema como uma série de transições entre estados com entradas e saídas. As transições são rotuladas com as ações que causam as transições.

O funcionamento básico de um ar-condicionado pode ser um exemplo de sistema que pode ser modelado utilizando IOLTS, uma vez que ambos envolvem a entrada de dados e a saída de ações correspondentes. 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 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. Se estiver mais baixa, ele pode entrar em um modo de espera para economizar energia. Enquanto estiver ligado, o ar-condicionado continua verificando a temperatura do ambiente para saber quando resfriar ou esperar.

Essa interação contínua com o ambiente é o que valida essa comparação ao modelo IOLTS, onde a entrada é a leitura da temperatura e a saída é a ação de resfriar ou de esperar. A imagem abaixo mostra um exemplo desse funcionamento.

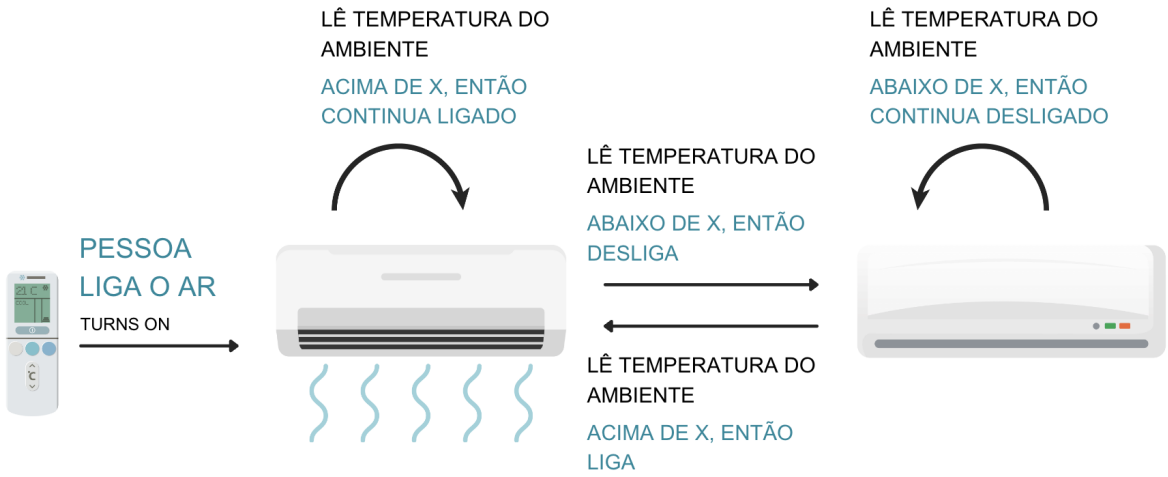

Figura 4 – Exemplo: Funcionamento do ar condicionado

Podemos então modelar esse sistema utilizando o IOLTS, como segue a imagem abaixo.

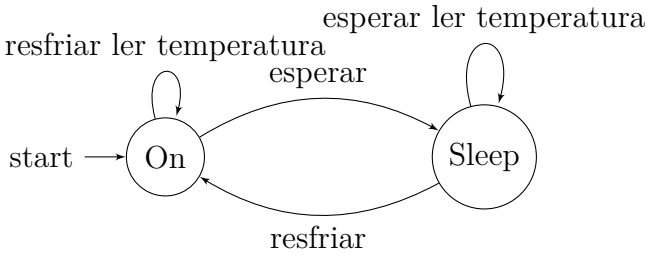

Figura 5 – Exemplo: IOLTS do funcionamento de um ar condicionado

O IOLTS é uma extensão dos LTSs [6], um sistema de transições rotuladas para descrever o comportamento de sistemas. Porém, nos IOLTSs [7] os rótulos, ou ações, são particionados em entradas e saídas, representando as interações do sistema com o ambiente.

O modelo IOLTS é definido, formalmente, por $M = (S, L_I, L_U, T, s_0)$, onde

- S é o conjunto contável de estados.
- L_I é o conjunto contável de rótulos de entradas.
- L_U é o conjunto contável de rótulos de saídas.
- T é uma relação do tipo $T \subseteq S \times (L \cup \tau) \times S$ que define o conjunto de transições, tal que $L = L_I \cup L_U$, $L = L_I \cup L_U \neq \emptyset$ e $L_I \cap L_U = \emptyset$.
- s_0 é o estado inicial.

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.

A partir deste modelo é possível: aplicar técnicas de geração de testes para criar casos de teste que contemplem diferentes cenários de interações de entrada e saída; verificação de conformidade; análise de cobertura de falhas; completude de conjuntos de testes; entre outros.

Testes baseados nesse modelo tem sido amplamente utilizados como um framework formal para verificar se uma implementação em teste (IUT) está em conformidade com uma especificação fornecida, de acordo com um determinado modelo de falha e uma relação de conformidade específica [7]. 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 encontrada [9].

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.

2.4.3 VPTS

O modelo VPTS (Visible Pushdown Transition System) consiste em um conjunto de estados, uma pilha (que pode ser observada e manipulada externamente) e um conjunto de transições rotuladas que descrevem as operações realizadas na pilha. Logo, um VPTS é definido formalmente por $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T \rangle$, onde:

- S é um conjunto finito de estados;
- $S_{in} \subseteq S$ é o conjunto de estados iniciais;
- L é um alfabeto;
- $\varsigma \notin L$ é um símbolo especial que indica ação interna;
- 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;
- $T = T_c \cup T_r \cup T_i$, 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.

O comportamento de um VPTS é dado pela noção de configuração.

Definição 1. *Seja $\mathscr{S} = \langle S, S_{in}, L, \Gamma, T \rangle$ um VPTS. Uma* configuração *de $\mathscr{S}$ é um par $(p, \alpha) \in S \times (\Gamma^\star\{\perp\})$. Quando $p \in S_{in}$ e $\alpha = \perp$, (p, α) é uma* configuração inicial *de $\mathscr{S}$. O conjunto de todas as configurações de $\mathscr{S}$ é dado por $\mathscr{C}_\mathscr{S}$. Seja $(q, \alpha) \in \mathscr{C}_\mathscr{S}$ e $\ell \in L_\varsigma$, escrevemos $(p, \alpha) \xrightarrow{\ell} (q, \beta)$ se existe uma transição $(p, \ell, Z, q) \in T$, tal que:*

1. *$\ell \in L_c$, e $\beta = Z\alpha$;*
2. *$\ell \in L_r$, e ambos (i) $Z \neq \perp$ e $\alpha = Z\beta$, ou (ii) $Z = \alpha = \beta = \perp$;*
3. *$\ell \in L_i \cup \{\varsigma\}$ e $\alpha = \beta$.*

Assim, um movimento simples de $\mathscr{S}$ 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 simples, $(q, \beta) \in \mathscr{C}_\mathscr{S}$ também é uma configuração de $\mathscr{S}$.

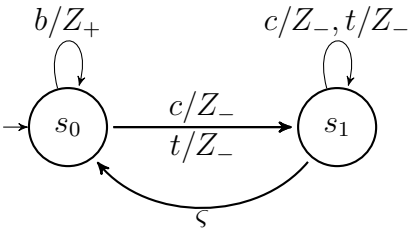

Figura 6 – A VPTS $\mathscr{S}_1$, with $L_c = \{b\}$, $L_r = \{c, t\}$, $L_i = \emptyset$.

Uma representação gráfica de um VPTS toma uma transição push (s, x, Z, q) com x/Z_+ próximo a aresta correspondente de s para q na figura. De forma semelhante, a transição pop (s, x, Z, q) terá um rótulo x/Z_- próximo a aresta de s à q. As transições simples e interna, $(s, x, \sharp, q)$, terão o rótulo x próximo a aresta correspondente.

Exemplo 1. *A Figura 1 representa um VPTS $\mathscr{S}$ com $S = \{s_0, s_1\}$, $S_{in} = \{s_0\}$. Os conjunto de rótulos são $L_c = \{b\}$, $L_r = \{c, t\}$, $L_i = \{\}$, e $\Gamma = \{Z\}$. Existe uma transição push (s_0, b, Z, s_0), as transição 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), e a transição interna $(s_1, \varsigma, \sharp, s_0)$. O comportamento de $\mathscr{S}$ 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 $\mathscr{S}$ voltando o controle para o estado s_0, através de um rótulo internal ς.*

Assim como nos IOLTSs, o conjunto de traces, ou comportamentos, definem a semântica de um VPTS.

Definição 2. *Seja $\mathscr{S} = \langle S, S_{in}, L, \Gamma, T\rangle$ um VPTS e $(p, \alpha), (q, \beta) \in \mathscr{C}_{\mathscr{S}}$.*

1. *Seja $\sigma = l_1, \ldots, l_n$ uma palavra em $L_\varsigma^\star$. σ é um* trace *de (p, α) até (q, β) se existem as configurações $(r_i, \alpha_i) \in \mathscr{C}_{\mathscr{S}}$, $0 \leq i \leq n$, tal que $(r_{i-1}, \alpha_{i-1}) \xrightarrow{l_i} (r_i, \alpha_i)$, $1 \leq i \leq n$, com $(r_0, \alpha_0) = (p, \alpha)$ e $(r_n, \alpha_n) = (q, \beta)$.*

2. *Seja $\sigma \in L^\star$. σ é um* trace observável *de (p, α) à (q, β) em $\mathscr{S}$ se existe um trace μ de (p, α) até (q, β) em $\mathscr{S}$ tal que $\sigma = h_\varsigma(\mu)$.*

Logo, 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). Quando (q, β) é* alcançável *em $\mathscr{S}$, então ela é alcançável de uma configuração inicial de $\mathscr{S}$.*

Como o símbolo ς pode ocorrer num trace, então quando esses símbolos são removidos, o trace é chamado de observável. Um trace σ de (p, α) to (q, β) é representado por $(p, \alpha) \xrightarrow{\sigma}$

(q, β). Quando σ é um trace observável de (p, α) para (q, β), então pode ser representado por $(p, \alpha) \stackrel{\sigma}{\Rightarrow} (q, \beta)$.

2.4.4 IOVPTS

O modelo IOVPTS (Input/Output Visibly Pushdown Transition System) é um sistema de transição com uma memória auxiliar, permitindo a modelagem e análise de sistemas com comportamentos assíncronos mais complexos [2]. Este modelo é uma variante do VPTS (Visibly Pushdown Labeled Transition System) [10], que por sua vez podem ser associados aos VPAs (Visibly Pushdown Automata) [11], uma classe de autômatos de pilha mais restrito que os tradicionais PDAs [12].

Com uma memória auxiliar, diferente dos IOLTS, os IOVPTSs podem modelar sistemas reativos mais expressivos, considerando tanto as interações com o ambiente externo quanto o estado interno do sistema. Este cenário é muito mais desafiador, pois o modelo base possui uma pilha para capturar comportamentos mais complexos, comumente encontrados em sistemas reativos complexos [2].

De forma geral o IOVPTS abrange uma classe de modelos mais complexa, usando uma memória auxiliar [2]. Para essa modelagem utiliza-se pilhas visíveis. Pilhas visíveis são estruturas de dados LIFO (Last-In-First-Out) onde o elemento inserido por último é o primeiro a ser retirado.

O IOVPTS permite modelar as transições de estado do sistema através de regras que especificam:

- Estado atual: O estado em que o sistema se encontra atualmente.
- Entrada: O evento externo que o sistema recebe do ambiente.
- Saída: O evento gerado pelo sistema como reação à entrada.
- Empilhamento: Os elementos que são adicionados à pilha visível durante a transição.
- Desempilhamento: Os elementos que são retirados da pilha visível durante a transição.
- Estado seguinte: O novo estado em que o sistema se encontra após a transição.

A pilha visível desempenha um papel importante no comportamento do sistema, pois os elementos da pilha podem influenciar as transições subsequentes. Isso permite modelar sistemas reativos que dependem do histórico de interações com o ambiente para determinar seu comportamento futuro [13]. A capacidade de modelar sistemas reativos complexos

com dinâmicas de pilha torna o IOVPTS um formalismo poderoso. No entanto, o uso de uma memória de pilha adiciona complexidade à relação de conformidade adotada para esses modelos, o que impacta na verificação de conformidade entre implementações e especificações.

O IOVPTS é definido por: $\mathcal{J} = \langle S, S_{in}, L_I, L_U, \Gamma, T \rangle$, onde:

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

Seja $t = (p, x, Z, q)$ uma transição de T. Uma transição *push*, $t \in T_c$, significa que uma entrada x está sendo lida quando o controle se move do estado p para q em $\mathscr{S}$, e empilha Z. Já uma transição *pop* onde $t \in T_r$, diz que uma mudança no controle de $\mathscr{S}$ de p para q, $x \in L_r$ é lida e desempilha o símbolo Z. Uma transição do tipo pop pode ser realizada com a pilha vazia, quando o símbolo é $\perp$. Por fim, uma transição simples $t \in T_i$ ocorre quando $x \in L_i$ ou por transição interna quando $t \in T_i$ com $x = \varsigma$. No primeiro caso, t lê um x se movendo de p para q sem modificar a pilha. De forma similar, no segundo caso a pilha também fica inalterada, porém nenhum símbolo de entrada é lido.

2.5 Verificação de Conformidade

2.5.1 Conformidade

Uma das abordagens usadas para verificação da corretude de sistemas é a verificação de conformidade [1]. O objetivo é garantir que o comportamento de um sistema esteja de acordo com o comportamento de sua respectiva especificação. Neste contexto, a figura **??** ilustra a relação entre a implementação, a especificação e o teste de conformidade.

Na figura 7, a implementação do sistema é representada por uma caixa preta. Isso significa que o seu funcionamento interno é opaco para o testador, ou seja, não há visibilidade dos detalhes da codificação ou dos mecanismos internos. O testador só pode interagir com a implementação através de suas interfaces, fornecendo inputs e observando os outputs.

Em contraste, a especificação do sistema é representada por uma caixa branca. Isso indica que o comportamento esperado do sistema é totalmente transparente para o 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.

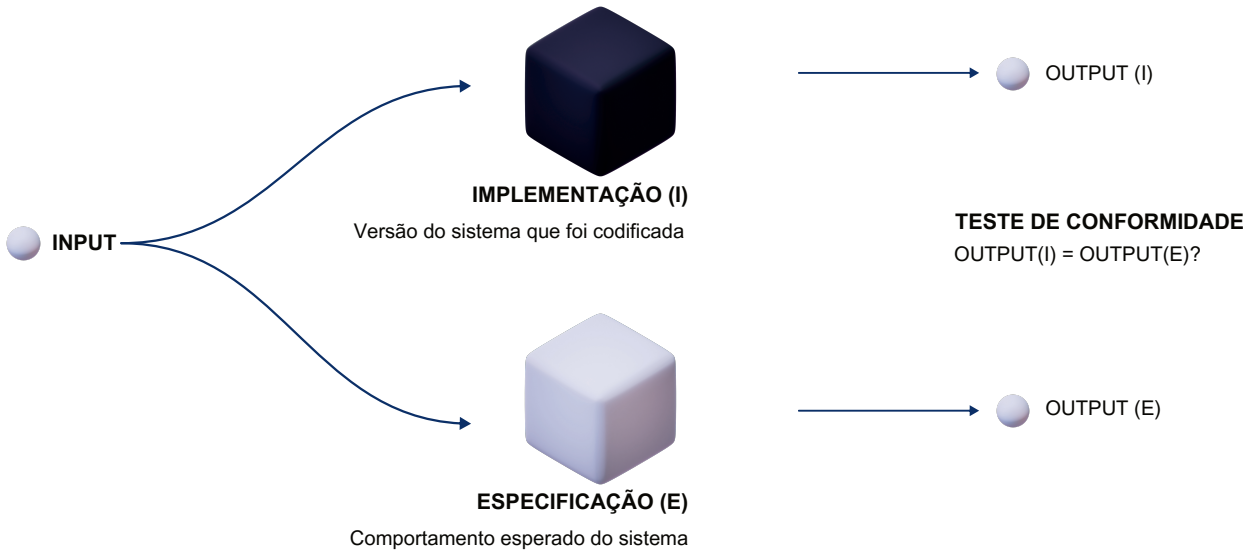

Figura 7 – Relação de conformidade

O teste de conformidade consiste em comparar o 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.

No exemplo fornecido abaixo, a entrada para ambos, implementação e especificação, é ”aababaaa”. A saída da implementação é ”aabxabaayay”, enquanto a saída da especificação é ”aabxabaayax”. Ao verificar então a conformidade entre as saídas da implementação e da especificação nota-se a diferença no último caractere, indicando que a implementação não está em total conformidade com a especificação.

Quiescência. Em conformidade a quiescência representa um estado especial de um sistema onde este não produz nenhuma saída. É simbolizado pelo caractere δ. Este conceito é importante para avaliar a conformidade de uma implementação em relação a uma especificação. A quiescência permite modelar situações onde o sistema não deve produzir nenhuma saída observável. Isso é relevante para especificar comportamentos esperados em determinados cenários.

Por exemplo, em uma especificação de um sensor de temperatura. Em situações onde não há variação de temperatura detectada, a especificação pode indicar que o sensor deve ficar em quiescência δ. A implementação do sensor deve respeitar esse comportamento, não produzindo nenhuma saída (valor de temperatura) até que haja uma mudança na temperatura detectada.

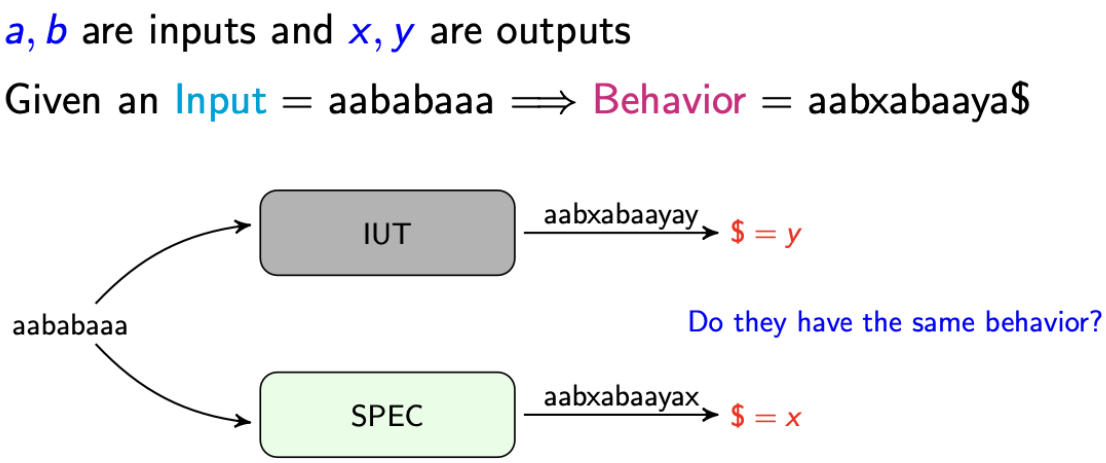

Figura 8 – Exemplo depois vou substituir

Implementação. A primeira coisa necessária para o teste é a Implementação Sob Teste (IUT). A IUT é o sistema que está sendo testado. Uma implementação pode ser um objeto físico real, como um hardware, um programa de computador com todas as suas bibliotecas rodando em um processador específico, um sistema embarcado contendo software embutido em algum dispositivo físico ou um sistema de controle de processo com sensores e atuadores. Como lidamos com testes de caixa preta, uma implementação é tratada como uma caixa preta que exibe comportamento e interage com seu ambiente, mas sem conhecimento sobre sua estrutura interna. A única maneira de um testador controlar e observar uma implementação é por meio de suas interfaces. O objetivo do teste é verificar a correção do comportamento da IUT em suas interfaces [6].

Especificação. A correção de uma IUT é expressa como conformidade a uma especificação. A especificação prescreve o que a implementação deve fazer e o que não deve fazer. Em testes formais, a especificação é expressa em alguma linguagem formal, ou seja, uma linguagem com sintaxe e semântica formais. Seja essa linguagem, o conjunto de todas as expressões válidas nessa linguagem, denotada por $SPEC$, então uma especificação s é um elemento dessa linguagem: $s \in SPEC$. Por meio de teste, queremos verificar se o comportamento da IUT está em conformidade com s [6].

Conformidade. Para verificar se uma implementação sob teste (IUT) está em conformidade com uma especificação (s), necessitamos de uma definição formal de conformidade. Essa definição deve relacionar implementações e especificações. No entanto, ao tentar definir

tal relação, surge um problema.

Enquanto uma especificação (s) é um objeto formal pertencente a um domínio formal ($SPEC$), uma implementação sob teste não é passível de raciocínio formal. Uma IUT não é um objeto formal: é uma entidade física real, composta por software, hardware, componentes físicos ou uma combinação destes, na qual somente experimentos e testes podem ser realizados.

Para raciocinar formalmente sobre implementações, adotamos um pequeno artifício: assumimos que qualquer implementação sob teste real (IUT) pode ser modelada por um objeto formal (i_{IUT}) em um conjunto de modelos (MOD). O domínio MOD é escolhido a priori e é chamado de universo de modelos de implementação. Essa suposição é comumente referida como hipótese de teste. É importante ressaltar que a hipótese de teste pressupõe um domínio específico de modelos (MOD) e que somente se assume que exista um modelo válido (i_{IUT}) da IUT nesse domínio, mas não que esse modelo (i_{IUT}) seja conhecido a priori [6].

Portanto, a hipótese de teste permite o raciocínio sobre implementações sob teste como se fossem implementações formais em MOD. É isso que faremos daqui para frente. Consequentemente, a conformidade pode ser expressa por uma relação formal entre modelos de implementações e especificações. Por isso, essa abordagem utiliza como base os modelos formais, para que cada relação de conformidade seja definida com base em um modelo específico.

2.5.2 IOCO

Uma abordagem para verificação de conformidade em sistemas reativos usa a relação IOCO (Input/Output Conformance), cujo objetivo é verificar se as saídas geradas pelo sistema são aquelas esperadas na especificação [6]. A relação *ioco* é definida sobre o modelo IOLTS e permite a comparação entre uma implementação e sua respectiva especificação IOLTS.

Informalmente, uma implementação $i \in \mathcal{IOS}\left(L_I, L_U\right)$ é ioco-conforming à especificação $s \in \mathcal{LTT}\left(L_I, L_U\right)$ se qualquer experimento derivado de s e executado em i leva a uma saída de i que é prevista por s [6].

De forma intuitiva, a verificação de conformidade baseada na relação *ioco* consiste basicamente na verificação do seguinte aspecto: a implementação sob teste pode produzir apenas saídas que também são produzidas pela especificação após uma sequência de estímulos.

Uma saída especial de i é a ausência de saídas, modelada pela quiescência δ. Isso significa que, se i estiver quiescente, então s também deve ter a possibilidade de estar quiescente

[6]. Ou seja, para uma implementação ser considerada conforme IOCO à especificação, ela deve respeitar o comportamento de quiescência definido na especificação (veja Seção 2.5.2): Se a especificação indica que o sistema deve ficar quiescente em uma determinada situação, a implementação também deve permanecer em quiescência nesse cenário.

Seja q um estado em um sistema de transição, e seja Q um conjunto de estados, então [6]:

1. $\text{out}(q) =_{\text{def}} \left\{x \in L_U \mid q \xrightarrow{x}\right\} \cup \{\delta \mid \delta(q)\}$
2. $\text{out}(Q) =_{\text{def}} \cup \{\text{out}(q) \mid q \in Q\}$

REFERÊNCIAS

[1] 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⟩.

[2] 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⟩.

[3] THOMPSON, M. *Reactive Systems: Principles and Patterns*. [S.l.: s.n.], 2020.

[4] ABBOTT, J. R. *Real-Time Systems Design and Analysis*. [S.l.: s.n.], 2017.

[5] TRETMANS, J. *Testing Techniques*. [S.l.: s.n.], 2004.

[6] 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⟩.

[7] 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⟩.

[8] KULL, A. Model-based testing of reactive systems.

[9] BONIFáCIO, A. L.; NASCIMENTO, C. Extração de propósitos de teste para modelos reativos. Disponível em: ⟨http://www.uel.br/cce/dc/wp-content/uploads/ProjetoTCC_CAROLINE_P_G_DONATO_NASCIMENTO.pdf⟩.

[10] 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⟩.

[11] ALUR, R.; MADHUSUDAN, P. Visibly pushdown languages. In: *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 2004. (STOC '04), p. 202–211. ISBN 1-58113-852-0. Disponível em: ⟨http://doi.acm.org/10.1145/1007352.1007390⟩.

[12] HOPCROFT, J. E.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Comutation*. [S.l.]: Addison Wesley, 1979.

[13] CARDELLI, D. S. e L. Input/output-visible pushdown transition systems. Journal of the ACM, 2002.