Universidade Estadual de Londrina

CAMILA SONODA GOMES

EVEREST:

UMA FERRAMENTA PARA VERIFICAÇÃO DE CONFORMIDADE E GERAÇÃO DE TESTES PARA MODELOS REATIVOS

Londrina
2020

CAMILA SONODA GOMES

EVEREST:

UMA FERRAMENTA PARA VERIFICAÇÃO DE CONFORMIDADE E GERAÇÃO DE TESTES PARA MODELOS REATIVOS

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Adilson Luiz Bonifácio

Londrina
2020

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

G633e Gomes, Camila Sonoda.
Everest : uma ferramenta para verificação de conformidade e geração de testes para modelos reativos / Camila Sonoda Gomes. - Londrina, 2020.
123 f. : il.

Orientador: Adilson Luiz Bonifácio.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2020.
Inclui bibliografia.

1. Computação - Tese. 2. Software - Desenvolvimento - Tese. 3. Sistemas reativos (Computação) - Tese. 4. Sistemas de transição rotulada de entrada e saída - Tese. I. Bonifácio, Adilson Luiz. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519

CAMILA SONODA GOMES

EVEREST:

UMA FERRAMENTA PARA VERIFICAÇÃO DE CONFORMIDADE E GERAÇÃO DE TESTES PARA MODELOS REATIVOS

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

_____

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

_____

Prof. Dr. André Takeshi Endo
Universidade Tecnológica Federal do Paraná – UTFPR

_____

Prof. Dr. Rodolfo Miranda de Barros
Universidade Estadual de Londrina – UEL

_____

Prof. Dra. Jandira Guenka Palma
Universidade Estadual de Londrina – UEL

Londrina, 16 de abril de 2020.

AGRADECIMENTOS

Primeiramente a Deus, por me permitir realizar mais um sonho, sempre a me abençoar e conduzir.

Ao meu orientador Adilson Luiz Bonifácio pela paciência, dedicação e suporte durante todo este percurso do mestrado.

A minha família que, não só neste momento, mas que em toda minha vida estiveram comigo, ao meu lado, fornecendo o apoio e compreensão. Ao meu noivo por me apoiar de maneira incondicional. E aos amigos que fiz durante o mestrado.

Também agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela concessão da bolsa durante o período de realização do mestrado.

GOMES, C. **EVEREST:** Uma ferramenta para verificação de conformidade e geração de testes para modelos reativos. 2020. 123 f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2020.

RESUMO

O processo de desenvolvimento de *software* é composto de várias etapas. Uma importante etapa neste processo é o teste de *software*, essencial para a entrega de um produto com qualidade. Nesta etapa, a aplicação de métodos formais tem se mostrado bastante promissora, em especial no desenvolvimento de sistemas reativos, caracterizados pela constante interação com o ambiente. Muitos sistemas reativos são também de natureza crítica onde a precisão é um fator importante e quando não atendida pode causar danos irreparáveis. O teste baseado em modelo tem sido amplamente empregado em sistemas reativos, tanto na geração de conjuntos de teste quanto na verificação de conformidade entre implementações candidatas e suas respectivas especificações, modeladas por *Input/Output Labeled Transition Systems* (IOLTSs). A verificação de conformidade permite detectar se o comportamento de uma implementação está de acordo com o comportamento modelado pela sua especificação. Já a geração de testes tem como objetivo construir conjuntos de teste para que as falhas, definidas de acordo com um domínio específico, sejam encontradas em implementações candidatas. Neste trabalho foi desenvolvida uma ferramenta de verificação de conformidade para modelos IOLTS, bem como a geração de conjuntos completos de teste para tais modelos reativos. A ferramenta suporta uma noção de conformidade mais geral baseada em linguagens regulares para especificar os comportamentos desejáveis e indesejáveis de uma implementação. Além de permitir uma verificação de conformidade mais geral, a ferramenta oferece também o *Input Output Conformance Testing* (**ioco**). O desenvolvimento da ferramenta compreendeu tanto o projeto conceitual dos algoritmos e das estruturas dados necessárias, quanto a análise e o projeto de desenvolvimento prático da ferramenta. Estudos de caso foram descritos a fim de apresentar as funcionalidades da ferramenta. Um estudo comparativo e experimentos práticos também foram contemplados no trabalho de forma a destacar algumas vantagens da ferramenta desenvolvida em relação às ferramentas similares existentes na literatura.

Palavras-chave: Teste baseado em modelos. Sistemas reativos. Verificação de conformidade. Geração de teste. IOLTS.

GOMES, C. **EVEREST:** A tool for conformance verification and test generation for reactive models. 2020. 123 p. Master's Thesis (Master in Science in Computer Science) – State University of Londrina, Londrina, 2020.

ABSTRACT

The software development process is comprised by several steps. An important step in this process is the software testing which guarantees more quality on the system products. Formal methods have been promising in testing activities, especially for reactive systems that are characterized by the interaction with the environment. Several reactive systems are also critical where accuracy is deemed important to avoid irreparable damage. Model-based testing has been widely applied to reactive systems, either in test suite generation or in conformance checking processes between candidate implementations and their respective specifications, modeled by Input Output Labeled Transition Systems (IOLTSs). Conformance checking allows for detecting whether the behavior of an implementation conforms to the behavior modeled by the specification. On the other hand, test generation aims to construct test suites for finding faults in implementations according to a specific domain. In this work we have developed a tool for checking conformance over IOLTS models and also for generating complete test suites for such models. This tool supports the classical Input Output Conformance Testing (**ioco**) and a more general notion of conformance based on regular languages, where desirable and undesirable behaviors can be specified by regular languages. The tool's development comprised of the algorithm design and data structures, as well as the software development process. Moreover, we described case studies to show the tool's functionalities. A comparative study and practical experiments are also offered in order to point out the advantages of our tool with respect to similar tools in the literature.

Keywords: Model-based testing. Reactive systems. Conformance testing. Test generation. IOLTS.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ATM	Automatic Teller Machine
CADP	Construction and Analysis of Distributed Processes
CS	Characterizing Sequence
DFA	Deterministic Finite Automaton
DS	Distinguishing Sequence
FSA	Finite State Automata
FSM	Finite State Machine
GB	Gigabyte
GNU	GNU's Not Unix!
IDE	Integrated Development Environment
IE	Internet Explorer
ioco	Input Output Conformance Testing
IOLTS	Input/Output Labeled Transition Systems
IUT	Implementation Under Test
JML	Java Modeling Language
LTS	Labeled Transition System
MBT	Model Based Testing
NFA	Nondeterministic Finite Automaton
OCL	Object Constraint Language
RAM	Random Access Memory
STG	Symbolic Test Generator
TCP	Transmission Control Protocol
TGV	Test Generation with Verification Technology
TP	Test Purpose
UDP	User Datagram Protocol

UIO	Unique Input–Output Sequence
UML	Unified Modeling Language

SUMÁRIO

1 INTRODUÇÃO

O processo de desenvolvimento de *software* é composto por várias etapas tais como, levantamento de requisitos, análise e projeto, codificação, validação e manutenção [82]. O teste de *software* é uma das atividades aplicadas frequentemente na etapa de validação de sistemas reativos [34].

Sistemas reativos, tanto de *software* quanto de *hardware*, são caracterizados pela interação contínua com o ambiente [27]. Muitos sistemas reativos também são críticos, falhas em sistemas desta categoria podem resultar em consequências drásticas tais como, prejuízos financeiros, de segurança e até perdas de vidas humanas. Por isso, sistemas dessa natureza precisam de um suporte rigoroso no processo de desenvolvimento, como a verificação e validação formal de especificações e testes baseados em formalismos.

A atividade de teste, em geral, demanda muito tempo, é custosa e ainda suscetível a erros. Com o crescimento da complexidade dos sistemas e, por consequência, das atividades de teste, a automatização dos testes se tornou alvo de inúmeros estudos [27, 34, 11, 93, 13, 15]. Neste contexto a abordagem de teste baseado em modelos tem se mostrada promissora, onde a geração de conjuntos de teste e a verificação de conformidade têm sido grandes desafios [27].

No teste baseado em modelos a especificação corresponde ao modelo detalhado que caracteriza e descreve os comportamentos dos sistemas. O conjunto de teste é gerado de acordo com esses comportamentos. A aplicação dos testes, gerados com base numa especificação, e a verificação de conformidade ocorre tanto em testes de caixa-branca quanto em testes de caixa-preta. Nos testes de caixa-branca a estrutura interna das implementações são conhecidas, permitindo que um modelo formal a represente, então os casos de teste podem ser aplicados para que os requisitos não cumpridos e comportamentos indesejáveis sejam encontrados [80]. Já no teste de caixa-preta não se tem acesso à estrutura interna da implementação. Neste caso, os testes são realizados através de entradas fornecidas à implementação e pela observação das saídas produzidas. Já a verificação de conformidade retorna um veredito positivo quando o comportamento de uma implementação corresponde ao comportamento modelado na especificação, de acordo com um modelo de falhas definido.

Algumas ferramentas da literatura têm sido desenvolvidas visando a geração de conjuntos de teste, tal como o *Test Generation with Verification Technology* (TGV) [5], ferramentas como Torx [89] e JTorx [4] também implementam a verificação de conformidade. Estas ferramentas lidam com o modelo IOLTS, comumente usado para especificar sistemas reativos assíncronos [90, 37, 103, 32, 92, 11, 93, 14]. O TGV implementa a geração de conjunto de teste com base na estratégia **ioco** [91], o Torx implementa a relação de conformidade **ioco** e o JTorx a relação uioco [95, 93], uma variação do **ioco** que aceita modelos menos restritivos. Ambas as ferramentas geram casos de teste completos para a relação **ioco**.

Este trabalho se desvia da noção clássica **ioco** para uma abordagem mais geral de conformidade baseada em linguagens regulares [27]. Essa abordagem permite que falhas não detectadas usando a conformidade **ioco** sejam encontradas através dessa noção mais ampla da qual não se tem conhecimento a cerca de ferramentas práticas. Uma ferramenta de teste, a EVEREST (*conformancE Verification on tEsting ReactivE SysTems*), foi então desenvolvida para suportar ambas as noções de conformidade, a **ioco** [93] tradicional e conformidade mais geral baseada em linguagens regulares, além da geração de conjuntos completos de teste para modelos IOLTS.

1.1 Objetivos Gerais e Específicos

As ferramentas que lidam com a verificação de conformidade e geração de conjuntos de teste, encontradas na literatura, via de regra são suportadas pela conformidade clássica **ioco** ou suas variações. Não se tem conhecimento de ferramentas práticas que suportam alguma abordagem mais ampla e distinta da conformidade **ioco**, tal como a abordagem de uma relação de conformidade baseada em linguagens.

Neste contexto este trabalho tem como objetivo a elaboração da ferramenta EVEREST para verificação de conformidade entre implementações e suas respectivas especificações usando modelos reativos e linguagens regulares. A verificação de conformidade entre os modelos é estabelecida com base na relação de conformidade clássica **ioco** e na abordagem mais geral baseada em linguagem, proposta por Bonifácio e Moura [27]. Este método de verificação de conformidade mais abrangente permite que a EVEREST seja capaz de detectar falhas que as ferramentas existentes não conseguem identificar usando a abordagem **ioco**. A especificação dos comportamentos desejáveis e indesejáveis através de linguagens regulares pode simplificar o processo teste, previamente definidos pelo *tester* conforme os objetivos específicos do teste. A EVEREST deve gerar conjuntos completos e finitos de teste para modelos IOLTS, tanto nos cenários de caixa-branca quanto de caixa-preta, além de permitir a execução dos testes.

Já os objetivos específicos do trabalho são:

- estudo do método de conformidade baseado em linguagens e da geração de conjuntos de teste;
- projeto dos algoritmos de conformidade e geração de testes, bem como a definição de estruturas de dados para lidar com as linguagens e os modelos formais;
- desenvolvimento da ferramenta de teste EVEREST ;
- aplicação da EVEREST em experimentos práticos;
- comparação da EVEREST com outras ferramentas similares da literatura.

1.2 Organização do Trabalho

O texto desta dissertação é organizado da seguinte estrutura. O Capítulo 2 apresenta a fundamentação sobre teste de *software*, teste baseado em modelos, modelos formais e propriedades sobre linguagens e autômatos. As abordagens de verificação de conformidade baseada em linguagem e **ioco**, bem como o método de geração de conjuntos de teste usando propósitos de teste, e seus respectivos algoritmos são descritos no Capítulo 3. Os trabalhos mais intimamente relacionados com a pesquisa desenvolvida nesta dissertação e uma análise preliminar de ferramentas da literatura são abordados no Capítulo 4. No Capítulo 5 estão presentes o projeto e desenvolvimento da EVEREST, além exemplos práticos de sua aplicação. Já o Capítulo 6 exibe uma análise comparativa entre a EVEREST e as ferramentas similares, além de estudos de caso e experimentos práticos para avaliação de desempenho, tanto na verificação de conformidade quanto na geração de conjuntos de teste. As considerações finais do trabalho desenvolvido estão contemplados no Capítulo 7.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos relacionados ao teste de sistemas, em especial os reativos, a verificação de conformidade, a geração e execução de conjuntos de teste, a cobertura e modelos de falha, bem como as características e aplicações do teste baseado em modelos.

2.1 Teste de Software

O teste de *software* é parte integrante do processo de Verificação e Validação (V&V) no desenvolvimento de sistemas. Enquanto a etapa de verificação avalia se um *software* implementa corretamente uma funcionalidade específica, a validação analisa se o *software* atende aos requisitos especificados durante a fase de desenvolvimento [82]. Através da fase de teste, espera-se que requisitos não cumpridos e comportamentos indesejáveis sejam encontrados, de modo a aumentar a qualidade final do produto [80]. Vale ressaltar a afirmação de Edsger W. Dijkstra de que os testes podem ser usados para encontrar falhas, mas nunca a sua ausência [102].

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 eficácia dos testes para lidar com sistemas cada vez mais complexos [102]. Um método eficiente procura otimizar o processo de teste aumentando a produtividade e diminuindo recursos. Já a eficácia é satisfeita de acordo com a finalidade dos testes, portanto, deve garantir que todos os requisitos do sistema sejam testados [65].

Alguns tipos de teste são apropriados a uma fase específica do desenvolvimento do *software* e/ou são mais apropriados a um determinado artefato do desenvolvimento, conforme os objetivos específicos do teste aplicado. O método V [30] propõe testes durante todo o ciclo de desenvolvimento do *software*. Desta forma os testes são trabalhados em diferentes níveis de acordo com a fase de desenvolvimento em que o *software* se encontra. 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 [94].

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 [94].

O teste pode ainda ser realizado de duas formas: (1) conhecendo a estrutura interna do sistema sob teste, como o código implementado e a estrutura lógica; (2) conhecendo a

especificação, mas sem o prévio conhecimento da estrutura interna da implementação. De acordo com estes aspectos os testes são classificados como [82]:

Teste de caixa preta: também conhecido como teste funcional, neste tipo de teste o sistema é tratado como uma caixa preta, sobre o qual não se tem conhecimento a respeito da estrutura e dos detalhes internos. Neste caso o sistema só pode ser acessado e observado por sua interface com o ambiente externo. A validação das funcionalidades acorre através da análise das reações do sistema em resposta aos estímulos externos. O teste funcional não trata da verificação de propriedades como performance, usabilidade, confiabilidade e manutenção [93].

Alguns exemplos de testes desta categoria são: teste de intervalo, teste de limite de valor, teste de integridade de banco de dados, teste de exceção, entre outros. Os testes de caixa preta têm como limitação a ausência do conhecimento da estrutura interna do sistema, por isso um teste completo e exaustivo, de forma geral, se torna impossível [65].

Teste de caixa branca: também conhecido como teste estrutural, este tipo de teste busca por falhas estruturais de baixo nível, seja no código fonte, na base de dados ou na interface [25]. Testes estruturais têm por vantagem a completude dos testes, uma vez que se tem conhecimento a cerca da estrutura interna do sistema, os erros de lógica têm maior probabilidade de serem detectados [65].

Teste de caixa cinza: engloba tanto o teste funcional quanto o estrutural. Neste tipo de teste é analisado a funcionalidade do sistema com base na especificação e sua estrutura lógica interna, o código fonte. Esta abordagem de teste é bastante útil em funcionalidades reutilizadas pelo sistema, pois o conhecimento da estrutura interna do sistema permite que o *tester* elimine testes desnecessários [65].

A vantagem do teste de caixa branca é que há garantia de que (1) toda a implementação será testada, (2) todas as condicionais com valores verdadeiro ou falso serão executadas, (3) todos os *loops* dentro do limite de intervalo serão executados, e (4) todas as estruturas de dados, para garantir sua validade, serão também executadas [81]. Entretanto, em sistemas cujo código fonte contém muitos *loops* e estruturas condicionais, em geral, existem grandes quantidades de caminhos lógicos, tornando o teste impraticável [81]. Nestes casos, os testes de caixa preta são a única alternativa viável ou a alternativa mais barata, mesmo que não se obtenha a cobertura completa de falhas. Os testes de caixa preta são um dos mais utilizados na prática, alguns *testers*, entretanto, aplicam os testes de caixa preta conjuntamente com testes de caixa branca para verificar partes não testadas de uma implementação, e então projetar testes específicos para estes casos [94]. Ambas as estratégias de teste são aplicáveis em testes de sistemas reativos, embora o teste de caixa preta seja mais difundido em sistemas desta natureza.

Sistemas reativos, tanto de *software* quanto de *hardware*, são caracterizados pela interação contínua com o ambiente externo ao sistema [59], nesta interação o sistema pode aceitar as

entradas do ambiente ou emitir saídas [86], conforme ilustrado na Figura 1. Os sistemas reativos, em geral, são também críticos onde uma falha pode causar sérias consequências. Por isso, tais sistemas exigem uma precisão maior no processo de desenvolvimento e uma das principais alternativas tem sido a abordagem de teste baseado em modelos.

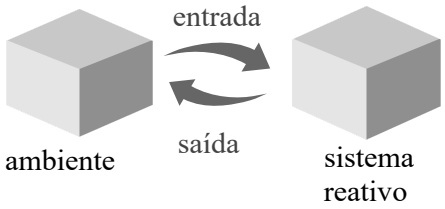


Figura 1 – Sistemas reativos

Esta categoria de sistemas foi introduzida em oposição aos convencionais sistemas transformacionais, onde as entradas são permitidas apenas no início da execução do sistema e as saídas retornadas somente ao término de sua execução [52, 46].

As principais características encontradas nos sistemas reativos são [51]:

Sincronismo: simultaneidade entre a execução do sistema e sua interação com o ambiente;

Confiabilidade: falhas em sistemas dessa natureza podem causar danos irreparáveis e/ou grandes prejuízos financeiros, por isso tais sistemas demandam maior rigor no processo de verificação e validação.

Alguns exemplos de sistemas reativos são: servidores *web*, protocolos de comunicação, sistemas operacionais, processadores, máquinas de auto-atendimento, sistemas embarcados, sistemas de controle de tráfego aéreo e reatores nucleares [29, 66, 53].

2.2 Teste Baseado em Modelos

Apesar da importância do processo de verificação e validação formal no desenvolvimento de *software*, durante o século XX com a valorização do desenvolvimento de sistemas a curto prazo e baixo custo, muitos projetos estavam abdicando da verificação e validação formal, essenciais para a garantia de um sistema com qualidade. Em pouco tempo as consequências começaram a manifestar-se quando falhas desastrosas e de alto custo foram constatadas em grandes projetos, como nos clássicos casos da Intel com o Pentium-II, onde uma falha foi ocasionada pela divisão de ponto flutuante (1995) e o acidente com o foguete Ariane-5, devido à conversão de um número real de 64 *bits* em um inteiro de 16 *bits* (1996) [10].

Testes manuais demandam muito tempo e recursos, e ainda assim são suscetíveis a erro. Segundo Tretmans [93] o teste de *software* demanda um custo elevado com relação ao esforço total de desenvolvimento. No teste manual os casos de teste são elaborados, com base nos requisitos do sistema, e executados, ambos de forma manual. Para cada caso de teste, o *tester*

interage diretamente com o sistema e compara a resposta obtida com a resposta esperada pelo caso de teste, e por fim o veredito do teste é registrado. Os testes manuais requerem que, a cada nova versão do sistema, o teste seja refeito, tornando a tarefa de teste trabalhosa e de alto custo. Neste contexto, para que seja mantido o orçamento de custo e tempo a cada versão do sistema, alguns casos de teste acabam sendo eliminados, levando o sistema a possíveis instabilidades e falta de robustez [29].

Com o aumento da complexidade dos sistemas e por consequência a complexidade de seus testes, a automatização dos testes tornou-se alvo de inúmeros estudos [93, 27, 21]. O Teste Baseado em Modelos (*Model Based Testing* - MBT) é uma das abordagens que proporciona a precisão e automatização dos testes através de formalismos. Como a especificação do sistema, em geral, é capturada por uma linguagem ou modelo formal adequado, em princípio, não é necessário a codificação da implementação para que os testes sejam gerados antes mesmo do início do desenvolvimento. A elaboração da especificação antes da implementação do sistema aumenta a qualidade do processo de desenvolvimento do *software* [80].

Uma implementação sob teste (*Implementation Under Test* - IUT) pode ser parte de um *hardware*, *software*, sistema embutido ou sistema com sensores e atuadores [93]. Já uma especificação corresponde ao modelo formal em alto nível de abstração, que caracteriza e descreve os comportamentos desejáveis/indesejáveis numa implementação, o que ela deve realizar e de que forma [62]. A especificação pode representar uma parte pequena de um sistema em relação ao todo, mas deve conter detalhes suficientes para descrever precisamente as características a serem testadas.

2.2.1 Um Framework Formal de Teste

O MBT pode ser definido como um *framework* formal para verificar se uma IUT esta conforme o comportamento caracterizado na especificação, dada por um modelo formal [27]. Com base no modelo formal da especificação os casos de teste são gerados, em geral, mais de um caso de teste é necessário para testar o sistema adequadamente [75]. Em comparação com o teste manual, o processo de geração e execução dos testes é otimizado, tendo como maior vantagem a geração de uma variedade de casos de teste com base em critérios de seleção estabelecidos [94].

A automatização dos testes, viabilizada pelo MBT, proporciona o aumento da eficiência e eficácia dos testes, e é composta por quatro principais atividades [102, 94]:

1. Modelagem formal do sistema: o sistema a ser testado deve ser mapeado em modelo formal apropriado;

2. Geração de testes: os testes são projetados a partir dos requisitos dados por um modelo formal;

3. Execução dos testes: os testes gerados são executados na IUT;

4. Avaliação dos resultados: após a execução, os resultados são analisados a fim de determinar a causa de uma falha encontrada;

Para adotar o MBT é necessário que uma especificação do sistema seja assumida como correta, usando um modelo (semi) formal válido, que caracterize o comportamento esperado e aceitável [93]. A vantagem de construir a especificação, validá-la, derivar os testes e executá-los, ao invés de validar diretamente o modelo completo da implementação, é a possibilidade de lidar com modelos menores, mais abstratos e mais simples do que IUT completa. A especificação ainda facilita o entendimento, a validação e a manutenção do sistema, além de simplificar a geração do conjunto de teste [29].

O processo de V&V, como mencionado anteriormente, faz parte do desenvolvimento de sistemas, cuja atividade de teste é uma de suas principais etapas. Com o embasamento formal, o MBT é uma das abordagens mais utilizadas no processo de validação dos sistemas [93]. Conforme cenários específicos e hipóteses bem definidas, o MBT oferece vantagens como a geração de conjunto de teste completo [27], de forma precisa, e automatizada, sendo também amplamente utilizado na verificação de conformidade entre IUTs e suas respectivas especificações.

No MBT a detecção de falhas e a verificação de conformidade entre uma IUT e sua especificação é estabelecida conforme ilustrada na Figura 2. Um conjunto de teste é extraído com base no modelo da especificação e no propósito de teste (*Test purposes - TPs)* ou no modelo de falha, que definem os comportamentos de falha do sistema. Cada elemento que constitui o conjunto de teste é denominado caso de teste. Cada caso de teste é aplicado à IUT e a saída gerada é observada e comparada com a saída esperada pela especificação ao aplicar o mesmo caso de teste. Se a saída obtida e a esperada forem distintas significa que uma falha foi detectada.

2.2.2 Modelagem Formal do Sistema

Um ponto importante do MBT é a escolha de um modelo formal apropriado para a descrição precisa do sistema a ser testado. Modelos formais adequados permitem que os sistemas sejam especificados, verificados e testados através de notação matemática [82]. O formalismo propicia uma análise mais precisa onde ambiguidades, incompletudes e inconsistências são mais facilmente descobertas e corrigidas [29].

A modelagem do sistema é essencial antes da aplicação do MBT e compreende as seguintes etapas [94]:

- Projetar o modelo com alto nível de abstração do sistema, omitindo detalhes desnecessários. O modelo deve incluir apenas estímulos de entrada e saída essenciais para modelar os comportamentos e operações a serem testados, pois a complexidade do processo de geração dos testes depende, entre outros fatores, do número de valores de entrada;
- Definir a notação formal para representar os sistemas, o que influencia diretamente na

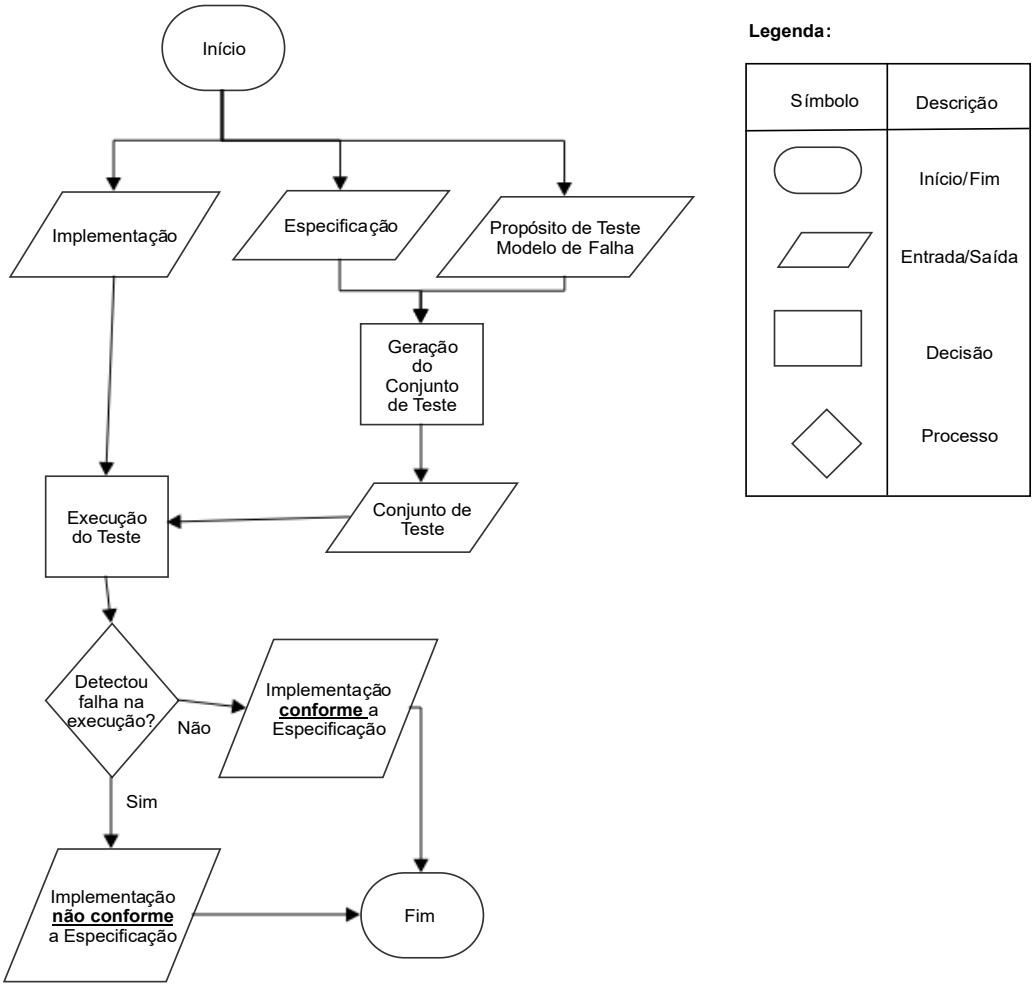


Figura 2 – Visão esquemática do MBT. (Fonte: Adaptado de Broy [29])

compatibilidade entre diferentes ferramentas. A notação adotada, em geral, está relacionada com as notações suportadas pelas ferramentas de MBT disponíveis. Notações que descrevem o sistema de modo mais natural tendem a ser mais difundidas;

- Validar o modelo para garantir que este especifique de forma precisa o comportamento do sistema a ser testado; e
- Verificar se o modelo está consistente e se foi corretamente escrito na notação definida;

Existem inúmeras notações que podem ser adotadas na modelagem da IUT e que são classificadas de acordo com os seguintes paradigmas [63]:

- Notações pré-pós (ou baseadas em estado): os sistemas são modelados como um conjunto de variáveis e operações que as modificam, onde cada operação é definida por uma pré e uma pós condição. Exemplos desta notação são B [9], *UML Object Constraint Language* (OCL) [99], *Java Modeling Language* (JML) [45], Spec# [84], VDM [18, 40] e Z [7, 58].
- Notações baseadas em transições: os sistemas de *software* ou *hardware* são descritos por um conjunto de estados acessíveis através de ações/interações com o ambiente. Desta

forma, o sistema pode ser visualizado como um grafo onde os vértices representam os estados e as arestas simbolizam as transições entre os estados. Os estados representam os possíveis cenários do sistema, portanto, retratam uma condição estável onde saídas podem ser produzidas e/ou novas entradas podem ser recebidas. Já uma transição indica o consumo de uma entrada, a emissão de uma saída e uma possível mudança de estado [29]. Exemplos desta notação são as máquinas de estados finito (*Finite State Machine* - FSMs) [26, 70, 56, 98, 48], os *Labeled Transition Systems* (LTSs) [90, 37, 103, 32], os diagramas de estado da *Unified Modeling Language* (UML) [19, 16, 12], os Autômatos Finitos (*Finite State Automata* - FSA) [74, 61], além de variações dos formalismos mencionados tais como os IOLTSs [92, 11, 93, 14], os modelos de tempo [38, 8, 41, 39] e de contexto [72, 85, 71, 97].

- Notações baseadas em história: modela o comportamento do sistema no decorrer do tempo usando várias notações de tempo (discreta, contínua, linear, ramificada, etc). O diagrama de sequência da UML é um tipo dessa notação e representa de forma gráfica e textualmente a ordem temporal de interações entre os componentes. Este tipo de notação porém não especifica em detalhes cada componente, por isso é mais usado para visualizar os testes gerados a partir do MBT e não para modelar a IUT;

- Notações funcionais: os sistemas são representados por um conjunto de funções matemáticas de primeira ordem, como especificações algébricas, ou de ordem superior como HOL (um ambiente interativo para prova de teoremas). Modelar o sistema a partir da notação funcional tende a ser mais complicado do que utilizar outras notações, por isso é pouco usada em MBT;

- Notações operacionais: o sistema é modelado como um conjunto de processos concorrentes, portanto são mais adequadas para descrever sistemas distribuídos e protocolos de comunicação. Entre outras temos a álgebra de processos e Redes de Petri;

- Notações estatísticas: descrevem o sistema como um modelo probabilístico de eventos e valores de entrada, em geral, são utilizados para modelar o ambiente e não a IUT. As cadeias de Markov [100] são exemplos desta notação.

- Notações de fluxo de dados: bastante utilizadas para modelar sistemas contínuos pois descrevem o fluxo de dados através da IUT. Exemplos desta notação são Lustre [67] e diagramas de blocos no Matlab [69].

Com o sistema, de software ou hardware, modelado por algum formalismo adequado, diferentes abordagens e métodos de teste baseado em modelos podem ser aplicados na fase de validação.

2.2.3 Geração e Execução de Testes

Uma das vantagens do MBT é a possibilidade da geração automática de grande quantidade de casos de teste a partir do modelo da especificação. Portanto, se o modelo de especificação é correto e representa precisamente as funcionalidades do sistema, os testes gerados também são comprovadamente válidos [93].

O MBT abrange quatro principais abordagens de geração de teste [94]:

1. Geração de entradas a partir do modelo de domínio: o modelo corresponde ao domínio dos valores de entrada e a geração de teste consiste na seleção da combinação dos subconjuntos dentro deste domínio.

2. Geração de entradas a partir do ambiente externo: a partir das interações com o ambiente, as possíveis sequências esperadas pela IUT são geradas com base nos intervalos de valores, operações, entre outros.

3. Geração de entradas e saídas esperadas a partir de modelos funcionais: o modelo deve descrever o comportamento esperado do sistema e sua relação com as entradas e saídas. Neste tipo de geração de teste o mais difícil é definir o veredito após a execução do teste, pois o gerador deve ser capaz de predizer os valores de saída do sistema. A vantagem desta abordagem é a automatização de todo o processo de teste desde a geração e execução dos testes até o veredito.

4. Geração de *script* de teste para testes abstratos: esta abordagem assume um modelo abstrato que representa a estrutura do sistema, tal como um diagrama de sequência da UML ou uma sequência de chamadas de métodos do sistema. Este modelo é então transformado em *script* de testes executáveis.

Algumas vantagens da geração automatizada de teste são: aumento da produtividade do *tester*; maior cobertura nos testes; geração de casos de teste mais eficazes; e redução de tempo e custo despendido na etapa de teste [75].

Para a geração dos casos de teste algumas informações importantes, como as descritas a seguir, podem ser utilizadas [75]:

1. Requisitos e especificação funcional;

2. Código fonte;

3. Domínio de valores de entrada e saídas; e

4. Modelo de falha.

Na abordagem do MBT os casos de teste gerados, com base num modelo formal, são sequências de símbolos (ou palavras) extraídas das execuções do modelo. Os símbolos pertencem a um alfabeto apropriado ao domínio do problema [55].

De forma geral os conjuntos de teste gerados são [94]:

- Finitos: os testes exaustivos não são praticáveis na maioria dos programas reais, pois, podem haver infinitas possibilidades de entradas para cada operação, além de entradas inválidas ou inesperadas.
- Selecionados: como os recursos para a geração/execução dos testes são limitados e a quantidade de casos de teste tendem a ser muito grande, um dos desafios no momento da geração do conjunto de teste é a seleção daqueles que serão capazes de expor falhas no sistema. Várias estratégias de cobertura podem auxiliar na seleção de conjuntos de teste efetivos.

Com o conjunto de teste gerado é possível realizar a verificação de conformidade entre uma IUT e uma especificação. A execução dos testes corresponde a fornecer estímulos as IUTs e observar se as respostas produzidas são àquelas esperadas [27]. A execução ocorre através da comunicação síncrona paralela entre os casos de teste e a IUT, e continua até que o último símbolo do caso de teste seja processado [29]. A execução dos testes só é considerada bem-sucedida, se para todo conjunto de casos teste a resposta observada na IUT corresponder a saída esperada na especificação. Desta forma, conclui-se que a IUT está conforme a especificação e nenhuma falha foi detectada [93].

O processo de teste pode ainda ser classificado como *online*, quando os testes são executados a medida que os casos de teste são produzidos, ou *offline*, quando todos os casos de teste são gerados e, somente depois dessa etapa completa, é que os testes são executados [94].

2.2.4 Cobertura de teste

Uma das dificuldades no processo de teste é decidir qual a condição de parada dos testes [57]. Para determinar se o sistema já foi o suficientemente testado pode-se utilizar métricas de cobertura de teste. Essas condições de cobertura também auxiliam na seleção dos testes, determinando os requisitos e comportamentos que devem estar contemplados no conjunto de teste, limitando assim a quantidade de casos de teste [29].

A descrição de um sistema real num modelo baseado em transições pode implicar em uma quantidade enorme de estados e transições. Consequentemente, para se testar completamente o sistema seria necessário uma grande quantidade de casos de teste, quando não infinitos, tornando a execução dos testes impraticáveis [29]. Portanto, torna-se imprescindível o uso de critérios de cobertura para a seleção dos testes.

Alguns aspectos devem ser observados no processo de seleção de casos de teste [75]:

1. As funcionalidades do sistema as quais pretende-se testar devem ser executadas ao menos uma vez;

2. O conjunto de teste não deve conter subcasos de testes a fim de evitar desperdício de recurso com casos de teste já executados;

3. Para definir o critério de cobertura e/ou modelo de falha mais adequado é necessário conhecer as funcionalidades especificadas.

A cobertura de teste define a capacidade de detecção de falha do conjunto de teste, pois, os casos de teste são selecionados com base em condições e critérios estabelecidos [29]. Dessa forma, a cobertura adotada influencia tanto no tamanho do conjunto de teste obtido, quanto no tempo necessário para gerar estes testes e nos aspectos das IUTs que serão testados [94]. Em sistemas reativos a cobertura pode ser definida com base no modelo de falhas. Neste caso o conjunto de teste é gerado para falhas específicas do modelo. Portanto, os testes são projetados para demonstrar a presença de um conjunto de falhas específicas com relação a uma especificação.

As principais métricas de cobertura estrutural para modelos baseados em transições, como o *Labeled Transition System* (LTS) e o *Input/Output Labeled Transition Systems* (IOLTS), são [75]:

1. Cobertura de estado: cada estado é visitado pelo menos uma vez. No entanto, este critério é relativamente ineficaz, pois, embora todos os estados estejam cobertos, podem haver transições que não foram alcançadas.

2. Cobertura de transição: cada transição é visitada pelo menos uma vez. Este critério é mais eficaz do que a cobertura de estado, embora apresente algumas limitações.

Em modelos reativos com entradas e saídas dissociadas, como é o caso dos modelos IOLTS, a cobertura é definida pelo modelo de falha adotado juntamente com a noção de conformidade desejada. Propósito de teste ou modelos de falha são usados para garantir que os testes gerados cubram um determinado domínio de falha especificado [79]. A noção de conformidade, definida de acordo com os objetivos de teste estabelecidos previamente, verifica se o comportamento de uma IUT está de acordo com sua especificação [27].

2.2.5 Completude de conjuntos de teste

Existem várias abordagens destinadas à geração de conjunto de teste completo para FSMs. Entre elas estão o método D [54], o método W [35, 96], o método Wp [44], o método U [87, 101] e o método T [76]. Esses métodos geram um conjunto de teste que garante uma cobertura completa de falhas. Porém, algumas restrições são impostas aos modelos para que a geração do conjunto de teste completo seja possível, tais como determinismo, minimalidade, completude e conectividade dos modelos. Em situações reais essas restrições impossibilitam a modelagem precisa da implementação principalmente quando se lida com teste de caixa preta [94] e sistemas reativos.

Neste contexto a geração de conjuntos completos de teste para sistemas reativos é um grande desafio no MBT [27]. Um conjunto de teste é dito completo se toda e qualquer falha do sistema é detectada por algum caso de teste do conjunto, levando-se em conta as restrições e o modelo de falhas assumidos. Na prática, este conceito é inviável devido aos seguintes aspectos [75]:

1. O domínio de possibilidades de entradas válidas e inválidas, sem restrições, fornecidas ao sistema é enorme, quando não infinito.
2. A criação de todos os cenários possíveis de execução do sistema pode não ser possível, em especial quando o comportamento do sistema depende de fatores do mundo real, tais como clima, temperatura, altitude e pressão.

Desta forma, premissas mais maleáveis podem ser adotadas para obtenção de conjuntos completos de teste. Neste caso todas as implementações corretas e algumas incorretas poderão passar pelo teste, mas nenhuma implementação correta resultará em falha (*soundness*). Já os testes exaustivos garantem que todas as implementações incorretas resultem em falha (*exhaustiveness*). A completude do conjunto de teste para modelos IOLTS pode ser obtida com base em algum modelo de falhas definido. Neste caso, a completude não significa necessariamente que qualquer falha será encontrada numa IUT, mas que toda e qualquer falha deve ser detectada no sistema levando-se em conta o domínio do modelo de falhas proposto, com suas condições e restrições assumidas [79].

2.2.6 Vantagens e Limitações

As principais vantagens do MBT podem ser classificadas em seis áreas [94]:

- Detecção de falhas: o principal objetivo do teste é a detecção de falhas na implementação. Como o poder de detecção de falhas, no MBT, depende do conhecimento de quem esta desenvolvendo o modelo e do critério de seleção utilizado na geração dos testes, não é possível afirmar que o MBT sempre irá detectar mais falhas do que os testes manuais, e vice-versa. Estudos mostram, porém, que os testes baseados em modelos são tão bons ou melhores do que os testes manuais no quesito detecção de falhas;
- Redução de custo e tempo: no MBT a maior parte do tempo é gasto na elaboração do modelo, na definição do critério de cobertura para a geração dos casos de teste, e no entendimento das ferramentas de MBT. Tem aumentado o número de publicações que relatam estudos de casos em que o custo do MBT é tipicamente menor do que os testes manuais. Embora em alguns casos o MBT seja mais custoso com relação ao tempo, a quantidade de testes gerados e a facilidade da avaliação dos requisitos é maior se comparado com os testes manuais.
- Maior qualidade do teste: a utilização de geradores de testes automatizados baseados em algoritmo e heurística torna o processo de geração sistemático e repetível. Com base na

cobertura do modelo é possível medir a qualidade do conjunto de teste gerado. Como os testes são gerados com base no modelo, a geração de mais casos de teste tem o custo associado apenas ao tempo da computação necessária para gerá-los, portanto, mais testes podem ser executados e mais falhas podem ser encontradas;

- Detecção de falhas de requisitos: a elaboração de um modelo preciso e abstrato que represente a IUT facilita a localização de falhas nos requisitos. Como a modelagem pode ser feita antes da codificação do sistema, falhas de requisitos podem ser detectadas e corrigidas em fase inicial, antes da implementação, o que torna o processo de correção menos custoso;

- Rastreabilidade: rastreabilidade é a capacidade de relacionar cada caso de teste ao requisito a ser testado. A rastreabilidade é útil a medida que o modelo evolui, pois, permite que a execução do teste seja realizada apenas no subconjunto de teste em que ocorreu alguma alteração. Por meio da rastreabilidade também é possível localizar as transições do modelo que não foram cobertas, visualizar o caso de teste como um conjunto de transições do modelo, encontrar os casos de teste que passam por uma determinada transição, identificar quais requisitos foram testados e reduzir o conjunto de teste. Além disso, a rastreabilidade facilita na localização e correção de falhas;

- Evolução dos requisitos: quando há alterações nos requisitos do sistema basta a correção do modelo, pois, os casos de teste são gerados automaticamente pela ferramenta de MBT. Como normalmente o modelo é menor que o conjunto de teste, a correção do modelo tende a ser menos custosa do que a correção manual de todos os casos de testes.

Já entre as limitações da aplicação de MBT está a necessidade expertise do *tester* para elaborar os modelos formais. Outra limitação está associada a automatização de outros tipos de teste que não seja o funcional. Requisitos desatualizados também podem levar a construção de modelos equivocados e a detecção de falhas inexistentes. Existem também cenários complexos onde a elaboração de testes manuais é mais vantajosa [94].

2.3 Modelos formais de teste

O MBT surgiu originariamente para o teste funcional de circuitos sequenciais, onde a modelagem usando FSMs é intuitiva. Outros formalismos baseados em transições surgiram ao longo do tempo, como os LTS, os diagramas de estado da UML, FSAs, e as mais diversas variações desses formalismos, como é o caso do IOLTS e os modelos de tempo e de contexto. A modelagem formal de um sistema reativo pode conter um conjunto infinito de estados e transições, considerando o aspecto prático, o modelo geralmente é a abstração de um sistema com menos detalhes podendo representar parte de um sistema, um comportamento específico ou ainda a interação do sistema com o ambiente [94]. As transições entre estados, nos modelos baseados em transições, são rotuladas com uma entrada e uma saída, que pertencem a algum alfabeto. A seguir são definidos os conceitos de símbolo e alfabeto de maneira precisa.

Definição 1. *([27]). Um alfabeto A é um conjunto não vazio de símbolos e uma palavra sobre este alfabeto é qualquer sequência finita $\sigma = x_0...x_n$ de símbolos de A, com $n \geq 0$ e $x_i \in A$, para todo $i = 0, 2, .., n$. Quando $n = 0$, σ é uma sequência vazia denotada por ϵ. O tamanho de uma palavra σ é dado por $|\sigma|$, e ϵ é uma palavra de tamanho 0. O conjunto de todas as palavras sobre A é denotado por $A^\star$. Uma linguagem L sobre A é qualquer conjunto $L \subseteq A^\star$. Se $L \subseteq A^\star$ é uma linguagem sobre A, então o seu complemento é a linguagem $\overline{L} = A^\star - L$.*

Algumas propriedades sobre modelos baseados em transições, necessárias na abordagem de teste estudada, são definidas a seguir [75, 29, 94]. Dado um modelo baseado em transições $\mathcal{M}$ com o conjunto de estados S e o alfabeto L, o modelo é:

- completamente especificado: se para cada estado $s \in S$ e cada rótulo $l \in L$ existe uma transição definida. Caso contrário $\mathcal{M}$ é parcial;
- fortemente conexo: se qualquer estado em S é alcançável por qualquer outro estado; e
- inicialmente conectado: se todo estado $s \in S$ é alcançável a partir do estado inicial.

As próximas subseções descrevem os principais modelos, LTS e IOLTS, tratados neste trabalho.

2.3.1 LTS

Os LTSs são modelos que capturam sistemas reativos e suas entradas e saídas ocorrem de forma independente. Estes rótulos de entrada e saída nos LTSs indicam, respectivamente, a execução de uma ação recebida, ou a saída produzida pelo sistema em resposta à algum estímulo [29].

Definição 2. *Um LTS sobre o alfabeto L é dado por $\mathcal{S} = (S, s_0, L, T)$ [27] onde:*

- *S é o conjunto de estados;*
- *s_0 é o estado inicial;*
- *L é o alfabeto de símbolos ou ações.*
- *$T \subseteq S \times (L \cup \{\tau\}) \times S$ é um conjunto finito de transições, onde a ação interna $\tau \notin L$. Uma transição (s, l, s') indica que a partir do estado de origem $s \in S$ através do rótulo $l \in (L \cup \{\tau\})$ o estado de destino $s' \in S$ é alcançado.*

O modelo pode se mover de um estado para o outro executando ações internas, expressas pelo símbolo $\tau \notin L$, que são independentes do ambiente. Logo, um LTS pode se mover com uma transição τ sem a necessidade de consumir uma entrada do ambiente. Um LTS é determinístico [17] se não possui ações internas τ e nenhum estado tem transições distintas com o mesmo rótulo.

O termo *caminho* denota a sequência de ações percorridas a partir de um determinado estado [11]. Um caminho é dito observável quando as ações internas τ foram removidas. Caminhos que iniciam do estado s são chamados de caminhos de s, e a semântica de um modelo LTS é dada pelos caminhos que partem do seu estado inicial. Estados que não são alcançáveis a partir do estado inicial podem ser eliminados.

Um caminho num LTS pode ser representado por $s \xrightarrow{\sigma} s'$, onde $s \in S$ é o estado de origem que, após percorrer o caminho $\sigma \in L_\tau^{\star 1}$, alcança o estado destino $s' \in S$. Um caminho observável σ, de s para s', é representado por $s \overset{\sigma}{\Rightarrow} s'$. Quando não há a necessidade de especificar o estado alcançado utiliza-se a notação $s \xrightarrow{\sigma}$ ou $s \overset{\sigma}{\Rightarrow}$ [27].

A semântica do modelo LTS é dada a seguir.

Definição 3. *([27]). Dado um LTS $\mathcal{S} = (S, s_0, L, T)$ e um estado $s \in S$:*

1. *O conjunto de caminhos de s é dado por $tr(s) = \{\sigma | s \xrightarrow{\sigma}\}$ e o conjunto de caminhos observáveis de s é $otr(s) = \{\sigma | s \overset{\sigma}{\Rightarrow}\}$.*

2. *A semântica de $\mathcal{S}$ é o conjunto $tr(s_0)$ ou $tr(\mathcal{S})$ e a semântica observável de $\mathcal{S}$ é o conjunto $otr(s_0)$ que também pode ser definida como $otr(\mathcal{S})$.*

Exemplo 1. *A Figura 3 representa um LTS $\mathcal{S} = (S, s_0, L, T)$. Neste exemplo $S = \{s_0, s_1, s_2, s_3\}$, s_0 é o estado inicial, o alfabeto é $L = \{a, b\}$ e o conjunto de transições é dado por $T = \{(s_0, a, s_1), (s_0, b, s_0), (s_1, a, s_3), (s_1, b, s_2), (s_2, \tau, s_0), (s_3, a, s_2)\}$, é possível observar a ação interna na transição de s_2 para s_0.*

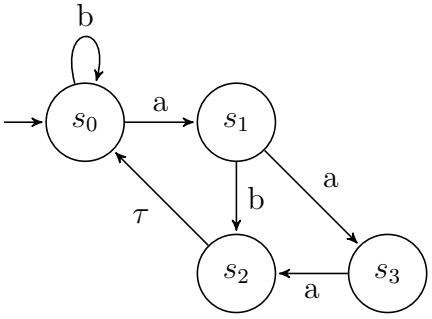


Figura 3 – Exemplo de LTS.

2.3.2 IOLTS

Os IOLTSs são uma variação dos LTSs. Os IOLTSs são apropriados em situações onde há a necessidade de separar as ações de entrada, como os estímulos recebidos do ambiente externo, das ações de saída, que são produzidas pelo sistema e enviados ao ambiente [27]. Por isso, os modelos IOLTS são mais adequados para a representação de sistemas reativos, onde os estímulos de entrada e saída são dissociados [24]. Um IOLTS é determinístico se o seu modelo LTS subjacente é determinístico.

[1] Para facilitar a notação usamos $L_\tau^\star$ para denotar $L^\star \cup \{\tau\}$.

A seguir o modelo IOLTS é formalmente definido em termos de um LTS subjacente.

Definição 4. *([93, 11]) Um IOLTS é dado por* $\mathcal{I} = (Q, q_0, L_I, L_U, T)$ *onde:*

- L_I *é o alfabeto de rótulos de entrada;*
- L_U *é o alfabeto de rótulos de saída;*
- $L_I \cap L_U = \emptyset$*;*
- $\mathcal{S} = (Q, q_0, L, T)$ *é o LTS subjacente associado a* $\mathcal{I}$*, sendo* $L = L_I \cup L_U$*.*

Além das ações internas, denotadas por τ, que não pertencem a L, os IOLTSs podem também conter ações quiescentes. A quiescência representa a ausência de ações de saída em um determinado estado e é denotada pelo símbolo δ, de modo que $\delta \notin L_\tau$. Em um cenário real a quiescência serve para indicar a lentidão para se produzir uma resposta, ou que o tempo foi excedido, ou ainda que a implementação não irá responder. Como a implementação não muda de estado nessa situação, a quiescência é modelada por um *self-loop* $s \xrightarrow{\delta} s$.

O alfabeto de entrada corresponde aos estímulos fornecidos pelo ambiente externo, enquanto o alfabeto de saída corresponde as respostas produzidas pelo sistema. Logo, o conjunto L_I do sistema deve corresponder ao conjunto L_U do ambiente, e vice-versa. Os marcadores "?" e "!" podem ser usados junto aos símbolos para facilitar a visualização das entradas em L_I e das saídas em L_U, respectivamente.

Um IOLTS é dito subespecificado (*underspecified*) caso não seja *input-enabled* [21]. A definição de *input-enabled* é dada a seguir.

Definição 5. *([27]) Um IOLTS é input-enabled se para cada estado* $s \in S$ *existem transições definidas para cada rótulo* $l \in L_I$*;*

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

Definição 6. *([27]) Dado o IOLTS* $\mathcal{I} = (S, s_0, L_I, L_U, T)$*, a semântica de* $\mathcal{I}$ *é o conjunto* $otr(\mathcal{I}) = otr(\mathcal{S}_\mathcal{I})$*, onde* $\mathcal{S}_\mathcal{I}$ *é o LTS subjacente associado ao IOLTS* $\mathcal{I}$*.*

Exemplo 2. *A Figura 4 representa um IOLTS* $\mathcal{S} = (Q, q_0, L_I, L_U, T)$*, com* $Q = \{q_0, q_1, q_2, q_3\}$*,* $L_I = \{a, b\}$ *e* $L_U = \{x\}$*. O conjunto de transições é dado por* $T = \{(q_0, a, q_1), (q_0, b, q_2), (q_1, b, q_1), (q_1, x, q_3)(q_2, b, q_2), (q_2, x, q_3), (q_3, \tau, q_0)\}$*.*

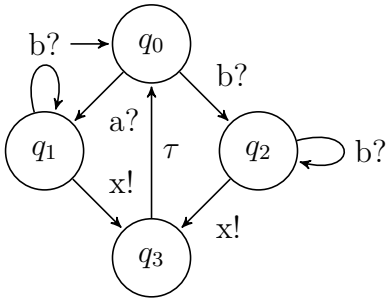


Figura 4 – Exemplo de IOLTS.

A seguir são definidas algumas funções importantes para a verificação de conformidade e geração de conjunto de teste.

Definição 7. *([27]) Dado um IOLTS* $\mathcal{S} = (S, s_0, L_I, L_U, T)$*, as funções out e inp representam, respectivamente, as saídas produzidas e as entradas aceitas a partir de um determinado estado. A definição das funções é dada por* $out(V) = \bigcup_{s \in V} \{l \in L_U | s \stackrel{l}{\Rightarrow}\}$ *e* $inp(V) = \bigcup_{s \in V} \{l \in L_I | s \stackrel{l}{\Rightarrow}\}$*. Um estado* $s \in S$ *é quiescente se* $out(s) = \emptyset$*.*

2.4 Propriedades de linguagens regulares

Os métodos de teste desenvolvidos neste trabalho se baseiam na teoria de linguagens formais, mais especificamente, nas linguagens regulares. Uma linguagem regular pode ser reconhecida por um autômato finito, que é um modelo formal baseado em transições, assim como os modelos LTS e IOLTS. Porém, os autômatos finitos possuem um conjunto especial para identificar os estados de aceitação (ou estados finais) do modelo. Após ler uma sequência de símbolos de entrada, se o autômato estiver em algum estado de aceitação diz-se que a cadeia de símbolos é aceita, caso contrário, a cadeia é rejeitada pelo autômato [68].

Definição 8. *([88]) Se L é o conjunto de todas as palavras aceitas pelo FSA* $\mathcal{M}$*, então* $L(\mathcal{M})$ *denota que* $\mathcal{M}$ *reconhece (ou aceita) L. Uma linguagem é dita regular se é reconhecida por um autômato finito. Um FSA que não aceita palavra alguma reconhece a linguagem* $\emptyset$*.*

As linguagens regulares também podem ser representadas por expressões regulares que são notações algébricas utilizadas para especificar as linguagens usando símbolos do alfabeto e operadores algébricos. As expressões regulares permitem representar de forma declarativa as palavras que são aceitas pelas linguagens regulares [55]. As expressões regulares são definidas, basicamente, por três operadores: união, concatenação e fecho Kleene [88].

Exemplo 3. *A seguir alguns exemplos de expressões regulares sobre o alfabeto* $\Sigma = \{a, b\}$*:*

1. $a.b^\star = \{a, ab, abb, abbb, ...\}$*, palavras iniciadas com 'a' seguido por zero ou mais* $b's$*;*
2. $(ab)^\star = \{\epsilon, ab, abab, abab, ababab, ..\}$*, palavras compostas por zero ou mais palavras 'ab';*
3. $(aaa)^\star = \{\epsilon, aaa, aaaaaa, ...\}$*, palavras constituídas de símbolos 'a', de tamanho divisível por 3;*
4. $(a|b)^\star = \{\epsilon, a, b, ab, ba, aa, bb, abab, ...\}$*, palavras compostas por qualquer número de* $a's$ *e* $b's$*.*

2.4.1 Automato Finitos

Os autômatos finitos (FSA) têm sido largamente empregados na modelagem de sistemas em diferentes áreas da computação [55]. Os FSAs são modelos compostos por estados e transições que são estimuladas de acordo com os símbolos que as rotulam [88].

O modelo FSA é formalmente definido a seguir.

Definição 9. *Um FSA $\mathcal{M}$ é dado por $\mathcal{M} = (S, s_0, L, T, F)$ onde:*

- *S é o conjunto de estados;*
- *s_0 é o estado inicial;*
- *L é o alfabeto;*
- *$T \subseteq S \times (L \cup \{\epsilon\}) \times S$ é o conjunto finito de transições e $\epsilon \notin L$ representa string vazia;*
- *$F \subseteq S$ é o conjunto de estados finais ou estados de aceitação.*

Um FSA é determinístico (*Deterministic Finite Automaton* - DFA) [94, 75] se (i) para cada rótulo $l \in L$ existe no máximo uma transição definida para cada $s \in S$ e (ii) não possui transições ϵ. Caso contrário, o FSA é não-determinístico (*Nondeterministic Finite Automaton* - NFA).

Exemplo 4. *A Figura 5 representa um FSA $\mathcal{M} = (S, s_0, L, T, F)$, com $S = \{s_0, s_1, s_2, s_3\}$, e $L = \{0, 1\}$. O conjunto de transições é dado por $T = \{(s_0, 0, s_1), (s_0, \epsilon, s_2), (s_1, 0, s_1), (s_1, 1, s_3), (s_2, 0, s_3)\}$, e $s_3 \in F$.*

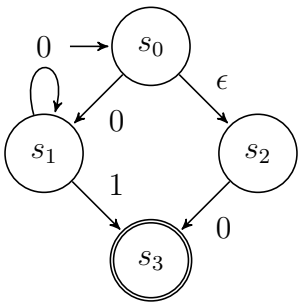


Figura 5 – Exemplo de FSA.

A semântica de um FSA é dada pela linguagem que o modelo aceita ou reconhece.

Definição 10. *([27]) Seja $\mathcal{M} = (Q, q_0, A, T, F)$ um FSA com $s, p, q \in Q$. A relação $\mapsto$ é definida como:*

- *$s \overset{\epsilon}{\mapsto} s$;*
- *$s \overset{a\alpha}{\mapsto} q$, se $(s, a, p) \in T$ e $p \overset{\alpha}{\mapsto} q$ com $a \in A \cup \{\epsilon\}$.*

A linguagem aceita por $\mathcal{M}$ é dada por $L(\mathcal{M}) = \{\alpha | q_0 \overset{\alpha}{\mapsto} q \text{ e } q \in F\}$. A linguagem $G \subseteq A^\star$ é regular *se existe um FSA $\mathcal{A}$ tal que $L(\mathcal{A}) = G$.*

Exemplo 5. *Seja o autômato finito $\mathcal{M} = (Q, q_0, A, T, F)$ da Figura 6 com $A = \{a, b\}$. A palavra $\sigma = baa$ é aceita por $\mathcal{M}$, parando no estado $q_4 \in F$, após seu processamento. A palavra $\sigma = aaa$ também é reconhecida pelo autômato. Já a palavra $\sigma = aab$ não é reconhecida, pois $(q_3, b, q) \notin T$, para qualquer $q \in F$.*

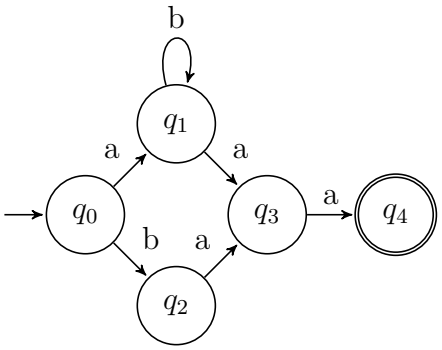


Figura 6 – FSA: processando palavras.

2.4.2 Fechamentos sobre linguagens regulares

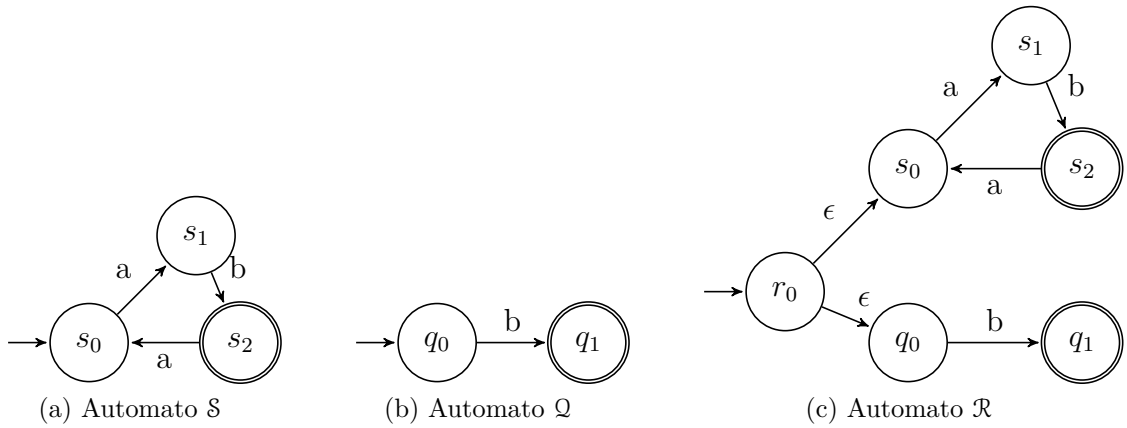
Os métodos de teste propostos neste trabalho lidam com os formalismos LTS e IOLTS para modelar as especificações e implementações. No entanto, envolvem também a utilização de autômatos e as respectivas propriedades sobre linguagens regulares, como união, interseção e complemento, que serão definidas na sequência. Além disso, transformações entre os modelos LTS/IOLTS e FSA, também necessárias, serão detalhadas mais adiante.

A primeira construção lida com a operação de união entre linguagens regulares. Dessa forma, dado dois autômatos finitos, que capturam suas respectivas linguagens regulares, um terceiro autômato finito, que captura a linguagem resultante da união das duas primeiras, é construído. Note que o autômato resultante é um NFA que reconhece as palavras que estão no conjunto união das duas linguagens de entrada.

Construção 1. *Sejam $\mathcal{S} = (S, s_0, L, \delta_s, F_s)$ e $\mathcal{Q} = (Q, q_0, L, \delta_q, F_q)$ dois FSAs que reconhecem as linguagens $L(\mathcal{S})$ e $L(\mathcal{Q})$, respectivamente. O FSA $\mathcal{R} = (R, r_0, L, \delta_r, F_r)$, tal que $L(\mathcal{R}) = L(\mathcal{S}) \cup L(\mathcal{Q})$, é construído da seguinte forma:*

- $R = \{r_0\} \cup S \cup Q$, *com* $r_0 \notin S \cup Q$
- $F_r = F_s \cup F_q$
- $\delta_r = \delta_s \cup \delta_q \cup \{(r_0, \epsilon, s_0), (r_0, \epsilon, q_0)\}$

Exemplo 6. *A Figura 7 mostra um exemplo da aplicação da Construção 1. O autômato $\mathcal{S}$ da Figura 7a reconhece a expressão regular $ab(aab)^\star$ e o autômato $\mathcal{Q}$ da Figura 7b reconhece b. A construção do autômato $\mathcal{R}$, que aceita a união das linguagens $L(\mathcal{S}) \cup L(\mathcal{Q})$, se dá a partir da criação de um novo estado r_0, que será o estado inicial de $\mathcal{R}$. Em seguida, são adicionadas as transições (r_0, ϵ, s_0) e (r_0, ϵ, q_0), e mantidas todas as transições e estados originais dos autômatos $\mathcal{S}$ e $\mathcal{Q}$. O autômato da Figura 7c então reconhece $ab(aab)^\star \vee b$.*



(a) Automato $\mathcal{S}$ (b) Automato $\mathcal{Q}$ (c) Automato $\mathcal{R}$

Figura 7 – União entre linguagens regulares.

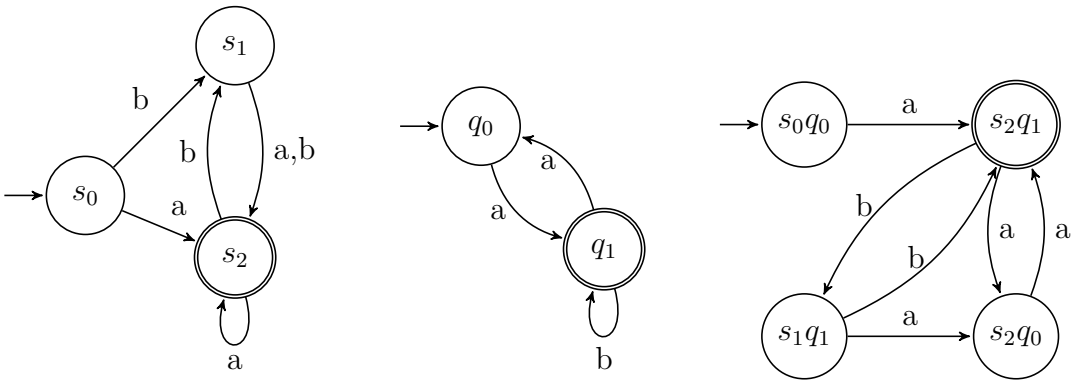
A próxima construção trata-se da operação de interseção entre linguagens regulares. Um autômato finito é construído para capturar a linguagem resultante da intersecção de outras duas linguagens regulares. O autômato resultante deve reconhecer apenas as palavras que são aceitas por ambos os autômatos finitos, que reconhecem as respectivas linguagens passadas por parâmetro.

Construção 2. *Sejam $\mathcal{S} = (S, s_0, L_s, \delta_s, F_s)$ e $\mathcal{Q} = (Q, q_0, L_q, \delta_q, F_q)$ dois DFAs que reconhecem as linguagens $L(\mathcal{S})$ e $L(\mathcal{Q})$, respectivamente. O FSA $\mathcal{R} = (R, r_0, L_q, \delta_r, F_r)$ é construído, tal que $L(\mathcal{R}) = L(\mathcal{S}) \cap L(\mathcal{Q})$, da seguinte forma:*

1. *$r_0 = (s_0, q_0)$ é rotulado pelo par de estados iniciais de $\mathcal{S}$ e $\mathcal{Q}$. Faça r_0 como estado inicial de $\mathcal{R}$.*
2. *Adicione o estado r_0 ao conjunto de estados R*
3. *$L_r = L_s \cup L_q$*
4. *Enquanto $R \neq \emptyset$*
   - *4.1 Remova $r = (s, q)$ de R*
   - *4.2 Para todo $l \in L_r$*
     - *Se $(s, l, s') \in \delta_s$ para algum $s' \in S$ e se $(q, l, q') \in \delta_q$ para algum $q' \in Q$*
       - *$r' = (s', q')$*
       - *$R = R \cup r'$ se $r' \notin R$*
       - *$\delta_r = \delta_r \cup (r, l, r')$ se $(r, l, r') \notin \delta_r$*
5. *$F_r = \{r = (s, q) | s \in F_s \text{ e } q \in F_q\}$*

Exemplo 7. *A Figura 8 ilustra a Construção 2. Os autômatos $\mathcal{S}$ e $\mathcal{Q}$ capturam, respectivamente, as linguagens $[b(a|b)(a|(b(a|b)))^\star]$ e $[a(b|(aa))^\star]$. O autômato $\mathcal{R}$ da Figura 8c é construído de tal forma que $L(\mathcal{R}) = a((bb)^\star|(baa)^\star|(aa)^\star)^\star$.*

Primeiramente, o estado inicial (s_0, q_0) é criado no autômato $\mathcal{R}$. Como existem transições definidas em $\mathcal{S}$ e $\mathcal{Q}$ sobre o mesmo símbolo a partir de s_0 e q_0, respectivamente, ocorre uma sincronização. A transição $((s_0, q_0), a, (s_2, q_1))$ é então adicionada em δ_r devido à sincronização das transições $(q_0, a, q_1) \in \delta_q$ e $(s_0, a, s_2) \in \delta_s$. Como $(s_0, b, s_1) \in \delta_s$, mas $(q_0, b, q') \notin \delta_q$ para algum $q' \in Q$, então não ocorre a sincronização, portanto, nenhuma outra transição do estado (s_0, q_0) é adicionada em δ_r. A partir do estado (s_2, q_1) de $\mathcal{R}$, ainda não explorado, é adicionada a transição $((s_2q_1), a, (s_2q_0))$, e a transição $((s_2q_1), b, (s_1q_1))$ rotulada com 'b' também é adicionada, pois $(s_2, b, s_1) \in \delta_s$ e $(q_1, b, q_1) \in \delta_q$. O processo se repete até que todos os estados em R sejam explorados, resultando no autômato $\mathcal{R}$ tal que $L(\mathcal{R}) = L(\mathcal{S}) \cap L(\mathcal{Q})$.



(a) Automato $\mathcal{S}$ (b) Automato $\mathcal{Q}$ (c) Automato $\mathcal{R}$

Figura 8 – Interseção entre linguagens regulares.

A próxima construção transforma um NFA $\mathcal{S}$ num DFA $\mathcal{R}$ de qual forma que $L(\mathcal{S}) = L(\mathcal{R})$.

Construção 3. *Seja $\mathcal{S} = (S, s_0, L, \delta_s, F_s)$ um NFA, construímos o DFA $\mathcal{R} = (R, r_0, L, \delta_r, F_r)$, tal que $L(\mathcal{S}) = L(\mathcal{R})$, da seguinte forma:*

1. *O estado inicial r_0 é rotulado com todos os estados alcançáveis por um caminho ϵ em $\mathcal{S}$ a partir de s_0 Assim, para todo $s_i \in S$ tal que $s_0 \xrightarrow{\epsilon} s_i$ temos que $s_i \in r_0$*
2. *Adicione r_0 ao conjunto de estados R*
3. *Enquanto $R \neq \emptyset$*

   3.1 Remova r de R

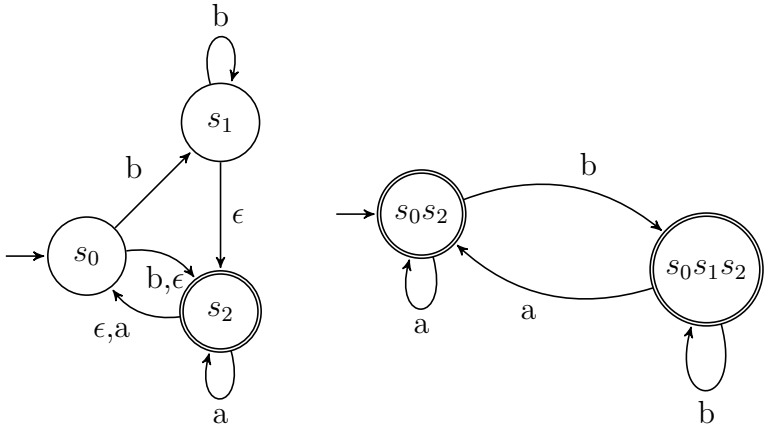
   3.2 Para todo $l \in L$

   - $r' = \emptyset$
   - *Para todo $s_i \in r$*
     - *Para toda transição $(s_i, l, s_j) \in \delta_s$ temos que $s_j \in r'$*
   - *Se $r' \neq \emptyset$*

- *Para cada $s_j \in r'$ se existe um caminho $s_j \xrightarrow{\epsilon} s_k$ em $\mathcal{S}$ para algum $s_k \in S$, crie r'' tal que $s_k \in r''$*
- *$R = R \cup r''$ se $r'' \notin R$*
- *$\delta_r = \delta_r \cup (r, l, r'')$ se $(r, l, r'') \notin \delta_r$*

4. *Para cada $r \in R$, se existe algum $s \in r$ tal que $s \in F_s$ então $F_r = F_r \cup \{r\}$.*

Exemplo 8. *Um exemplo da conversão de um NFA $\mathcal{S} = (S, s_0, L, \delta_s, F_s)$ em DFA $\mathcal{R} = (R, r_0, L, \delta_r, F_r)$ é apresentado na Figura 9.*



(a) NFA $\mathcal{S}$ (b) DFA $\mathcal{R}$

Figura 9 – Conversão de NFA em DFA.

O estado inicial de $\mathcal{R}$ é rotulado por todos os estados alcançáveis por caminhos ϵ partindo do estado inicial $s_0 \in S$, sendo assim o estado inicial s_0s_2 é criado e adicionado em R.

Retira-se de R o estado s_0s_2, o próximo passo é a criação de uma transição partindo de s_0s_2 com o rótulo 'a', como de s_0 não se tem transições rotuladas com 'a' e de s_2 tem-se a transição $(s_2, a, s_2) \in \delta_s$, partindo do estado alcançado s_2 com caminhos ϵ os estados s_0 e s_2 são obtidos, portanto, a transição (s_0s_2, a, s_0s_2) é adicionada em δ_r. Partindo de s_0s_2 com o próximo rótulo do alfabeto $b \in L$ tem-se que de s_2 não existem transições rotuladas com 'b' e de s_0 os estados s_1 e s_2 são alcançados pelas transições $(s_0, b, s_1), (s_0, b, s_2) \in \delta_r$. Partindo dos estados s_1 e s_2 com caminhos ϵ os estados s_0, s_1 e s_2 são alcançados, portanto, o estado $s_0s_1s_2$ é criado assim como a transição $(s_0s_2, b, s_0s_1s_2)$. O processo se repete até que todos os estados em R sejam explorados.

O estado de aceitação da NFA $\mathcal{S}$ é s_2, logo todos os estados da DFA $\mathcal{R}$ que tenha s_2 como rótulo é também estado de aceitação, neste caso todos os estados $\{s_0s_2, s_0s_1s_2\}$ são estados de aceitação.

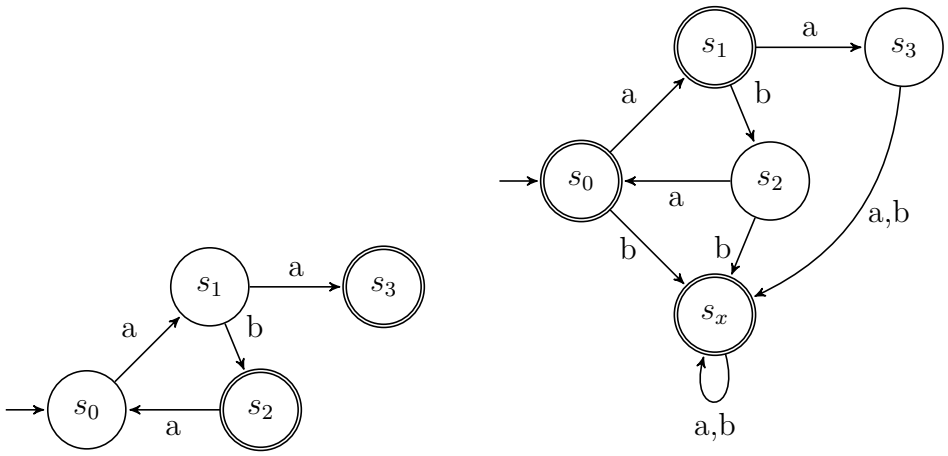
Outra importante construção é de um DFA que aceite o complemento de uma linguagem regular a partir do autômato que a reconhece.

Construção 4. *Seja* $\mathcal{S} = (S, s_0, L, \delta, F)$ *um DFA que aceita a linguagem* $L(\mathcal{S})$*, um DFA* $\overline{\mathcal{S}} = (S', s_0, L, \delta', F')$*, que reconhece a linguagem* $\overline{L}(\overline{\mathcal{S}})$*, onde* $\overline{L} = \Sigma^\star - L$*, é obtido conforme segue:*

- $S' = S \cup \{s_x\}$
- $\delta' = \delta \cup \{(s, l, s_x) | (s, l, s') \notin \delta \text{ com } l \in L \text{ e } s' \in S\}$
- $F' = \{S - F\} \cup \{s_x\}$

Exemplo 9. *A Figura 10b ilustra o autômato complemento* $\overline{\mathcal{S}}$ *dado o DFA* $\mathcal{S}$ *da Figura 10a que reconhece a expressão regular* $a(baa)^\star(a|b)$*. Note que os estados finais de* $\mathcal{S}$*,* s_2 *e* s_3*, não são estados finais em* $\overline{\mathcal{S}}$*, além disso, o estado* s_x *é utilizado para completar as transições não definidas em* $\mathcal{S}$*.*

O autômato complemento $\overline{\mathcal{S}}$ *reconhece as palavras que são rejeitadas pelo autômato* $\mathcal{S}$*. A palavra 'aa', por exemplo, é aceita pelo autômato* $\mathcal{S}$ *e é rejeitada por* $\overline{\mathcal{S}}$*, assim como a palavra'abb' é reconhecida por* $\overline{\mathcal{S}}$ *e é rejeitada por* $\mathcal{S}$*.*



(a) DFA $\mathcal{S}$ (b) Autômato $\overline{\mathcal{S}}$

Figura 10 – Complemento de uma DFA.

2.5 Considerações do capítulo

Este capitulo apresentou os conceitos relacionados ao teste de *software* e MBT, tais como cobertura, geração e execução de testes em sistemas reativos. Os modelos LTS e o IOLTS, adotados para representar sistemas reativos, foram introduzidos de forma detalhada, bem como as propriedades sobre linguagens regulares. Destaca-se como contribuição deste capítulo as construções das operações sobre linguagens regulares que foram elaboradas e implementadas como suporte ao método de verificação de conformidade e geração de testes desenvolvido neste trabalho. O próximo capitulo descreve os trabalhos mais intimamente relacionados com a pesquisa desenvolvida nesta dissertação.

3 UM MÉTODO DE TESTE PARA SISTEMAS REATIVOS

O método proposto por Bonifácio e Moura [27] abrange tanto a verificação de conformidade, entre IUTs e suas especificações, quanto a geração de conjuntos de teste, usando modelos IOLTS como formalismo base. Este capítulo define formalmente as noções de geração de conjunto de teste, verificação de conformidade baseada em linguagens e conformidade **ioco** [91]. Além disso, são descritos os algoritmos projetados para a verificação de conformidade e geração de conjuntos teste, bem como as construções dos modelos de falha e operações auxiliares sobre linguagens regulares, necessárias ao desenvolvimento da EVEREST.

3.1 Verificação de conformidade

Verificar se uma IUT está de acordo com o comportamento descrito numa dada especificação requer uma definição precisa do modelo de falhas e da relação de conformidade entre os modelos. A seguir serão apresentadas a relação de conformidade **ioco** e a relação de conformidade, mais geral, baseada em linguagens.

A relação **ioco**, definida para modelos IOLTS, estabelece a conformidade entre uma IUT e sua especificação. Essa relação de conformidade é verificada quando sequências de estímulos aplicadas a IUT produzem saídas previstas também na especificação [93].

Definição 11. *([93]). Seja* $\mathcal{S} = (S, s_0, L_I, L_U, T)$ *uma especificação e* $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ *uma IUT, ambas IOLTS.* $\mathcal{I}$ ***ioco*** $\mathcal{S}$ *se, e somente se,* $out(q_0\ after\ \sigma) \subseteq out(s_0\ after\ \sigma)$ *para todo* $\sigma \in otr(\mathcal{S})$*, onde* $s\ after\ \sigma = \{q | s \stackrel{\sigma}{\Rightarrow} q\}$ *para todo* $s \in S$*. Caso contrário,* $\mathcal{I}$ ~~***ioco***~~ $\mathcal{S}$*.*

Já a relação de conformidade baseada em linguagens [27] propicia uma cobertura de falha mais ampla tanto para modelos LTS quanto IOLTS. Essa abordagem permite testar comportamentos desejáveis e indesejáveis a uma implementação descritos por linguagens regulares e especificadas pelo *tester*. O método permite também a geração de conjuntos completos de teste para modelos LTS e IOLTS.

Seja a IUT $\mathcal{I}$, a especificação $\mathcal{S}$, a linguagem regular D, que define os comportamentos desejáveis, e a linguagem regular F, que especifica os comportamentos indesejáveis ao sistema. De acordo com a relação de conformidade baseada em linguagem, $\mathcal{I}$ está em conformidade a $\mathcal{S}$ de acordo com (D, F), ou seja, $\mathcal{I} conf_{D,F} \mathcal{S}$ se, e somente se, nenhum comportamento indesejável de F é observado em $\mathcal{I}$ e especificado em $\mathcal{S}$ e todos os comportamentos desejáveis de D são observados em $\mathcal{I}$ e especificado em $\mathcal{S}$.

Definição 12. *Dado um conjunto de símbolos* L*, e as linguagens* $\mathcal{D}, \mathcal{F} \subseteq L^*$ *sobre* L*. Seja* $\mathcal{S}$ *e* $\mathcal{I}$ *modelos LTS sobre os símbolos de* L *temos que* $\mathcal{I} conf_{D,F} \mathcal{S}$ *se, e somente se:*

1. $\sigma \in otr(\mathcal{I}) \cap F$*, então* $\sigma \notin otr(\mathcal{S})$*;*

2. $\sigma \in otr(\mathcal{I}) \cap D$*, então* $\sigma \in otr(\mathcal{S})$*.*

A noção de conformidade entre IUTs e especificações pode então ser expressa através de linguagens regulares e pelos seus comportamentos desejáveis ou indesejáveis.

Proposição 1. *([27]). Seja a especificação* $\mathcal{S}$ *e a IUT* $\mathcal{I}$*, modelos LTS sobre os símbolos de* L*, e as linguagens* $D, F \subseteq L^*$ *sobre* L*.* $\mathcal{I}$ $conf_{D,F}$ $\mathcal{S}$ *se, e somente se,* $otr(\mathcal{I}) \cap [(D \cap \overline{otr}(\mathcal{S})) \cap (F \cap otr(\mathcal{S}))] = \emptyset$*, onde* $\overline{otr}(\mathcal{S})$ *é o complemento de* $otr(\mathcal{S})$ *dado por* $\overline{otr}(\mathcal{S}) = L^* - otr(\mathcal{S})$*.*

Diferentes noções de conformidade podem ser derivadas de acordo com as linguagens especificadas em (D, F) [27]:

1. Quando apenas os comportamentos desejáveis são considerados temos que $D = L^*$ e $F = \emptyset$. Logo, $\mathcal{I} conf_{D,F} \mathcal{S}$ se, e somente se, $otr(\mathcal{I}) \subseteq otr(\mathcal{S})$;

2. Dada uma especificação $\mathcal{S}$, uma IUT $\mathcal{I}$ e $H \subseteq L$, um subconjunto de símbolos de L. Se os comportamentos desejáveis em $\mathcal{I}$ são os caminhos que terminam com um símbolo de H, então $D = L^*.H$ e $F = \emptyset$.

Vale ressaltar que para verificar se $\mathcal{I}$ é conforme a $\mathcal{S}$ de acordo com as linguagens regulares (D, F), assume-se que tais linguagens são finitas.

A relação de conformidade **ioco** é um caso especial da conformidade baseada em linguagem conforme mostra o Lema 1.

Lema 1. *([27]). Seja uma IUT* $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ *e uma especificação* $\mathcal{S} = (S, s_0, L_I, L_U, T)$*,* $\mathcal{I}$ ***ioco*** $\mathcal{S}$ *se, e somente se,* $\mathcal{I}$ $conf_{D,F}$ $\mathcal{S}$ *quando* $D = otr(\mathcal{S})L_U$ *e* $F = \emptyset$*.*

3.2 Conjuntos completos de teste

Esta seção descreve o método de geração de conjuntos de teste baseado nas linguagens D e F, usando os modelos LTS e IOLTS. A seguir são definidos os conceitos de conjunto de teste e casos de teste.

Definição 13. *([27]). Dado um conjunto de símbolos* L*, um conjunto de teste* T *sobre* L *é uma linguagem, onde* $T \subseteq L^*$*, de modo que cada* $\sigma \in T$ *é um caso de teste.*

A construção de um conjunto de teste tem o objetivo de detectar comportamentos indesejáveis a uma dada implementação $\mathcal{I}$, a partir de uma especificação $\mathcal{S}$ e usando o par de linguagens (D, F). O conjunto de teste obtido também é dado por uma linguagem regular, e por isso deve existir um autômato finito $\mathcal{A}$ que reconhece tal linguagem, onde os estados finais de $\mathcal{A}$ são os estados de falha. O conjunto de comportamentos indesejáveis, definidos por estes estados de falha, é denominado de modelo de falha de $\mathcal{S}$ [27]. Uma implementação $\mathcal{I}$ satisfaz o conjunto de teste $\mathcal{T}$ se nenhum comportamento observável em $\mathcal{I}$ é um comportamento de $\mathcal{T}$.

Definição 14. *([27]). Seja T um conjunto de teste e $\mathcal{I}$ uma IUT. Então $\mathcal{I}$ adere a T se, e somente se, para todo $\sigma \in otr(\mathcal{I})$ temos que $\sigma \notin T$.*

Dado a especificação $\mathcal{S}$ e o par de linguagens (D, F), um conjunto completo de teste pode ser extraído usando a Proposição 1. O conjunto de teste é dito completo se este é capaz de identificar a ausência de comportamentos desejáveis e a presença de comportamentos indesejáveis na especificação.

Lema 2. *([27]). Dado o conjunto de símbolos L, a especificação LTS $\mathcal{S}$ e as linguagens regulares $D, F \subseteq L^*$. O único conjunto completo de teste para $\mathcal{S}$ usando (D, F) é dado por:*

$$[(D \cap \overline{otr}(\mathcal{S})) \cup (F \cap otr(\mathcal{S}))].$$

3.3 Geração de propósitos de teste

Esta seção apresenta a técnica de geração de conjuntos de teste, baseada em propósitos de teste, onde a IUT é uma caixa-preta e os conjuntos de teste devem ser completos conforme o modelo de falhas [27].

O modelo de falhas, do método de geração abordado neste trabalho, é composto por um conjunto de TPs derivados de uma especificação. Um TP é formalmente definido por um IOLTS com dois estados especiais $\{pass, fail\}$, e representa um testador externo que interage com a IUT. O veredito do teste é obtido quando algum destes estados especiais é alcançado, através de um comportamentos de falha, quando o estado *fail* é alcançado, ou de comportamentos aceitáveis, quando o estado *pass* é alcançado [27].

Definição 15. *([27]) Dado o conjunto de símbolos de entrada L_I e de saída L_U, com $L = L_I \cup L_U$, onde $\mathcal{IO}(L_I, L_U)$ é a classe de todos os IOLTS sobre os alfabetos L_I e L_U. Um TP sobre L é qualquer IOLTS $\mathcal{T} \in \mathcal{IO}(L_U, L_I)$ tal que para todo $\sigma \in L^*$ não ocorre $fail \stackrel{\sigma}{\Rightarrow} pass$ e $pass \stackrel{\sigma}{\Rightarrow} fail$. Um modelo de falhas sobre L é um conjunto finito de TPs sobre L.*

A execução de um teste ocorre através do produto síncrono entre o TP $\mathcal{T}$ e a implementação $\mathcal{I}$, dado por $\mathcal{T} \times \mathcal{I}$. O TP interage com a implementação $\mathcal{I} = (S, s_0, L_I, L_U, T)$, produzindo saídas que são enviadas à $\mathcal{I}$, portanto o conjunto de símbolos de saída do TP corresponde ao conjunto L_I, de entradas da implementação, e o conjunto de entradas do TP corresponde ao conjunto L_U de $\mathcal{I}$.

A fim de garantir a propriedade de completude para os conjuntos gerados a partir da especificação $\mathcal{S}$, com relação a toda implementação, deve-se satisfazer a relação $\mathcal{I}$ **ioco** $\mathcal{S}$, ou então $\mathcal{I}$ ~~**ioco**~~ $\mathcal{S}$ para toda a implementação com falha. No entanto, para que seja possível um teste completo, de forma viável, é necessário que seja definido uma classe específica de implementações de interesse a serem cobertas.

Definição 16. *([27]) Seja a implementação $\mathcal{I} = (S_\mathcal{I}, q_0, L_I, L_U, T_\mathcal{I}) \in \mathcal{IO}(L_I, L_U)$ e o TP $\mathcal{T} = (S_\mathcal{T}, q_0, L_U, L_I, T_\mathcal{T})) \in \mathcal{IO}(L_U, L_I)$. Diz se que $\mathcal{I}$ passa por $\mathcal{T}$ se para todo $\sigma \in (L_I, L_U)^*$*

*e todo estado $q \in S_{\mathcal{I}}$, não ocorre $(t_0, q_0) \stackrel{\sigma}{\Rightarrow} (fail, q)$ em $\mathcal{I} \times \mathcal{T}$. Seja o modelo de falha dado por $\mathcal{M}$, diz-se que $\mathcal{I}$ passa por $\mathcal{M}$ se $\mathcal{I}$ passa por todos os TPs em $\mathcal{M}$. Dado então um IOLTS $\mathcal{S}$ e $\mathcal{IMP} \subseteq \mathcal{IO}(L_U, L_I)$ um conjunto de IOLTS, diz-se que $\mathcal{M}$ é **ioco** completo para $\mathcal{S}$ com relação à $\mathcal{IMP}$ se para toda implementação $\mathcal{I} \in \mathcal{IMP}$ tem-se que $\mathcal{I}$ **ioco** $\mathcal{S}$ se, e somente se, $\mathcal{I}$ passa por $\mathcal{M}$.*

Os casos de teste gerados pelo método proposto por Tretmans [93] possuem uma série de restrições sobre os modelos, tais como os TPs serem acíclicos, com execução finita, e *input-enabled*, pois o *tester* não pode prever a saída produzida pela IUT caixa-preta e portanto, qualquer estímulo produzido pela IUT deve estar habilitado. A execução completa do teste também é garantida devido a cada estado do TP estar habilitado para enviar ao menos um símbolo de entrada à IUT. Para evitar escolhas arbitrárias e não determinísticas o TP deve ser *output-deterministic*, ou seja, cada estado pode emitir apenas um dos símbolos de saída para a IUT. Por fim, os *self-loops* são permitidos apenas nos estados $\{pass, fail\}$. A seguir são definidas as propriedades *input-enabled* e *output-deterministic*.

Definição 17. *([27]) Dado $\mathcal{S} \in \mathcal{IO}(L_I, L_U)$. Dizemos que $\mathcal{S}$ é output-deterministic se $|out(s)| = 1$ e $\mathcal{S}$ é input-enabled se $inp(s) = L_I$ para todo $s \in S$.*

Um modelo de falhas **ioco** completo cujos os TPs são *input-enabled* pode então ser construído conforme a Proposição 2.

Proposição 2. *([27]) Seja $\mathcal{S}$ uma especificação IOLTS é possível construir um modelo de falhas $\mathcal{M}$ que seja **ioco** completo, determinístico, input-enabled, com um único estado fail e nenhum estado pass.*

Quando o TP é *input-enabled*, *output-deterministic* e acíclico exceto pelos estado *pass* e $fail$, todas as restrições impostas por Tretmans [93] são atendidas. Porém, para que os TP sejam acíclicos é preciso impor algum controle sobre o tamanho das implementações. A seguir uma verificação de conformidade é estabelecida quando é imposto um limite superior para o número de estados das IUTs.

Definição 18. *([27]) Dado $\mathcal{IMP} \subseteq \mathcal{IO}(L_I, L_U)$ e $m \geq 1$, denota-se por $\mathcal{IMP}[m]$ a subclasse de IOLTSs em $\mathcal{IMP}$ composta por todos os modelos com até m estados. Dado $\mathcal{S} \in \mathcal{IO}(L_I, L_U)$, o modelo de falha $\mathcal{M}$, sobre $L_I \cup L_U$, é m-**ioco**-completo para $\mathcal{S}$ com relação a $\mathcal{IMP}$ se, e somente se, $\mathcal{M}$ é **ioco** completo para $\mathcal{S}$ com relação a classe $\mathcal{IMP}[m]$. Neste caso todos os TPs são determinísticos e acíclicos.*

Como a construção de um modelo de falhas **ioco** completo composto por TPs acíclicos não é possível, neste trabalho modelos de falha m-**ioco** completos são construídos, onde m é um parâmetro fornecido.

Proposição 3. *([27]) Dado o IOLTS determinístico* $\mathcal{S} = (S, s_0, L_I, L_U, T) \in \mathcal{IO}(L_I, L_U)$ *e* $m \geq 1$*. Então existe um modelo de falhas* $\mathcal{M}$ *que é* m-***ioco*** *completo para* $\mathcal{S}$*, o qual todos os TPs de* $\mathcal{M}$ *são acíclicos e determinísticos.*

A construção do modelo de falha $\mathcal{M}$, m-**ioco** completo para $\mathcal{S}$, ocorre a partir da construção de um multigrafo com $mn + 1$ níveis, onde m é o número máximo de estados das IUT e n o número de estados em $\mathcal{S}$. Para cada nível do multigrafo, todos os estados e transições de $\mathcal{S}$ estão presentes, porém as transições que formam ciclos são direcionadas para os estados do próximo nível. O estado $fail$ é adicionado, bem como as transições para $fail$ com cada rótulo $l \in L_U$ não definido em cada estado do multigrafo. Dessa forma, se garante a construção de um multigrafo acíclico.

Cada possível caminho do multigrafo a partir do estado inicial até $fail$ compõe um TP determinístico que é adicionado a $\mathcal{M}$. Se uma IUT $\mathcal{I} = (Q, q_0, L_I, L_U, R)$ não passa pelo modelo de falhas $\mathcal{M}$, tem-se que $(t_0, q_0) \stackrel{\sigma}{\Rightarrow} (fail, q)$ em $\mathcal{M} \times \mathcal{I}$, para algum $\mathcal{TP} = (S_\mathcal{T}, t_0, L_U, L_I, T_\mathcal{T})$ em $\mathcal{M}$. Então $t_0 \stackrel{\sigma}{\Rightarrow} fail$ em $\mathcal{TP}$ e $q_0 \stackrel{\sigma}{\Rightarrow} q$ em $\mathcal{I}$.

Com os TPs m-**ioco** completos, determinísticos e acíclicos extraídos do multigrafo (Proposição 3), ainda é necessário que algumas propriedades sejam satisfeitas, tais como o *output-determinism* e *input-enabledness*.

Proposição 4. *Seja o IOLTS determinístico* $\mathcal{S} \in \mathcal{IO}(L_I, L_U)$ *e* $m \geq 1$*. Então existe um modelo de falha* $\mathcal{M}$ *que é* m-***ioco*** *completo para* $\mathcal{S}$ *com relação a* $\mathcal{IO}(L_I, L_U)[m]$*, cujos TPs são determinísticos, output-deterministic, input-enabled e acíclico exceto pelos self-loops nos estados pass e fail.*

Para assegurar a propriedade *input-enabledness*, o estado *pass* é adicionado ao TP e para cada $l \in L_U$ não definido em cada estado, uma transição é adicionada em direção ao estado *pass*. *Self-loops* rotulados por cada $l \in L_U$ também são adicionados aos estados *pass* e $fail$. Com estes ajustes o TP é determinístico, *input-enabled* e acíclico exceto pelos *self-loops* nos estados *pass* e *fail*.

Para garantir que o TP seja *output-deterministic*, para cada estado que não exista nenhuma entrada habilitada é adicionada uma transição a partir deste estado com qualquer rótulo $l \in L_I$ para o estado *pass*.

3.4 Algoritmos de verificação de conformidade

Esta seção apresenta os algoritmos de verificação de conformidade **ioco** e baseada em linguagem projetos para o desenvolvimento da EVEREST com base no método proposto por Bonifácio e Moura [27].

3.4.1 Conformidade ioco

O Algoritmo 1 realiza a verificação **ioco** dado os modelos IOLTS da especificação $\mathcal{S}$ e da IUT $\mathcal{I}$. A linha 2 do algoritmo indica a construção do modelo de falhas para a especificação $\mathcal{S}$ representado pelo autômato $\mathcal{A}_T$. Na linha 3 a IUT $\mathcal{I}$ é transformada no autômato $\mathcal{A}_I$, enquanto que as linhas 4 e 5 constroem o autômato $\mathcal{A}_B$ que reconhece a linguagem obtida da interseção das linguagens aceitas pelos autômatos $\mathcal{A}_T$ e $\mathcal{A}_I$. Note que $\mathcal{A}_T$ representa o modelo de falhas que captura os comportamentos que não devem ser encontrados na IUT. Quando a interseção não é vazia significa que algum comportamento está presente tanto no modelo de falhas quanto na IUT. Neste caso, o comportamento é uma palavra aceita pelo autômato $\mathcal{A}_B$ e o conjunto de estados de aceitação E de $\mathcal{A}_B$ não deve ser vazio. Logo, da linha 6 a linha 10 quando o conjunto E de estados finais é vazio a IUT $\mathcal{I}$ é declarada em conformidade com a especificação $\mathcal{S}$. Caso contrário, uma falha é detectada e $\mathcal{I}$ é declarada não conforme a especificação $\mathcal{S}$.

Algoritmo 1 Verify **ioco** Conformance

input: IOLTSs $\mathcal{S}, \mathcal{I}$
output: $\mathcal{I}$ **ioco** conforms to $\mathcal{S}$, or not

```
1:  function verifyIOCOConformance(S, I)
2:      A_T = faultModelIoco(S)
3:      A_I = ioltsToAutomaton(I)
4:      A_B = (S_B, s_0, L, δ, E)
5:      A_B = intersection(A_T, A_I)
6:      if E = ∅ then
7:          return I ioco S
8:      else
9:          return I ~~ioco~~ S
10:     end if
11: end function
```

Modelo de Falha ioco

O Algoritmo 2 constrói o autômato do modelo de falhas $\mathcal{A}_T$ com base na especificação $\mathcal{S}$. O modelo de falhas representa o conjunto de comportamentos que não devem ser encontrados numa implementação. A construção desse modelo é baseada no Lema 2, onde os conjuntos de comportamentos D e F são definidos como $D = otr(\mathcal{S})L_U$ e $F = \emptyset$.

A linha 2 do algoritmo constrói o autômato $\mathcal{A}_D$ que reconhece os comportamentos na especificação que terminam com a produção de uma saída. Na linha 3 é obtido o autômato $\mathcal{A}_S$ induzido pela especificação $\mathcal{S}$, enquanto que na linha 4 é construído o autômato $\mathcal{A}_{compS}$ que reconhece os comportamentos que não estão presentes na linguagem aceita pelo autômato da especificação. O modelo de falhas é então obtido na linha 5, onde o autômato $\mathcal{A}_T$ é construído de forma a capturar os comportamentos comuns as linguagens aceitas pelos autômatos $\mathcal{A}_D$ e

$\mathcal{A}_{compS}$. Dessa forma, o modelo de falhas representa os comportamentos considerados falhas em implementações candidatas.

Algoritmo 2 Fault Model **ioco**

input: IOLTS $\mathcal{S}$
output: The fault model automaton

```
1: function faultModelIoco(S)
2:     A_D = modelD(S)
3:     A_S = ioltsToAutomaton(S)
4:     A_compS = complement(A_S)
5:     A_T = intersection(A_D, A_compS)
6:     return A_T
7: end function
```

A construção do autômato que reconhece os comportamentos da especificação que terminam com uma saída é obtida pelo Algoritmo 3. O autômato $\mathcal{A}_D$ é construído de forma a capturar os comportamentos definidos pelo conjunto $D = otr(\mathcal{S})L_U$.

O conjunto de estados de $\mathcal{A}_D$ é definido, na linha 3, pelo conjunto de estados de $\mathcal{S}$ e mais um estado especial d. O estado d deve então ser o único estado de aceitação (linha 6) de $\mathcal{A}_D$. As transições de δ são criadas da seguinte forma:

1. As linhas de 7 a 13 transformam as transições τ do IOLTS $\mathcal{S}$ em transições ϵ do autômato $\mathcal{A}_D$; ou em transições de δ com o mesmo rótulo das transições de T.

2. As linhas de 14 a 18 constroem novas transições em δ com rótulos de saída de L_U a partir de todo estado de $\mathcal{S}$ para o estado especial d.

Como o autômato resultante $\mathcal{A}_D$ pode ser não-determinístico, a linha 19 retorna um autômato determinístico através da função *convertToDeterministicAutomaton*.

Algoritmo 3 Model D

input: IOLTS $\mathcal{S} = (S, s_0, L_I, L_U, T)$
output: $\mathcal{A}_D$ automaton

```
1: function modelD(S)
2:     A_D = (S, s_0, L, δ, F)
3:     S = S ∪ d
4:     L = L_I ∪ L_U
5:     δ = ∅
6:     F = d
7:     for (p, x, r) ∈ T do
```

```
8:          if x = τ then
9:              δ = δ ∪ (p, ϵ, r)
10:         else
11:             δ = δ ∪ (p, x, r)
12:         end if
13:     end for
14:     for s ∈ S do
15:         for l ∈ L_U do
16:             δ = δ ∪ (s, l, d)
17:         end for
18:     end for
19:     return convertToDeterministicAutomaton(A_D)
20: end function
```

3.4.2 Conformidade Baseada em Linguagem

A verificação de conformidade baseada em linguagens é realizada pelo Algoritmo 4. Dadas uma IUT $\mathcal{I}$, uma especificação $\mathcal{S}$ e o par de expressões regulares (D, F) através da função $regex$ (*Regular Expressions*), o algoritmo verifica se $\mathcal{I}$ está em conformidade com $\mathcal{S}$ com base nos comportamentos desejáveis e indesejáveis definidos pelas expressões regulares D e F, respectivamente, conforme a Proposição 1.

O modelo de falhas é representado pelo autômato $\mathcal{A}_T$ (linha 2) construído com base na especificação $\mathcal{S}$ e nas linguagens (D, F). Na linha 3 é obtido o autômato $\mathcal{A}_I$ induzido por $\mathcal{I}$ usando o Algoritmo 8. Já as linhas 4 e 5 constroem o autômato $\mathcal{A}_B$ que reconhece os comportamentos comuns das linguagens regulares aceitas pelos autômatos $\mathcal{A}_T$ e $\mathcal{A}_I$. Nas linhas de 6 a 10, se o conjunto de estados de aceitação E da intersecção é vazio então não existe comportamento comum entre o modelo de falhas e a implementação. Logo, nenhuma falha é detectada na IUT e o algoritmo declara que $\mathcal{I}$ está em conformidade a $\mathcal{S}$ baseado em (D, F). Caso contrário, ao menos uma falha é detectada em $\mathcal{I}$ e o algoritmo retorna que $\mathcal{I}$ e $\mathcal{S}$ não estão em conformidade, usando (D, F).

Algoritmo 4 Verify Language-Based Conformance

input: LTS $\mathcal{S}, \mathcal{I}$ and regex (D, F)
output: $\mathcal{I}$ conforms to $\mathcal{S}$, or not

```
1: function verifyLanguageConformance(S, I, D, F)
2:     A_T = faultModelLanguage(S, D, F)
3:     A_I = ltsToAutomaton(I)
4:     A_B = (S_B, s_0, L, δ, E)
5:     A_B = intersection(A_T, A_I)
6:     if E = ∅ then
```

```
7:      return ℐ conforms to 𝒮
8:    else
9:      return ℐ does not conform to 𝒮
10:   end if
11: end function
```

Modelo de Falha Baseado em Linguagem

O modelo de falhas baseado em linguagens é obtido através do algoritmo 5 com base no Lema 2. A construção do autômato que representa o modelo de falhas inicia com a construção do autômato $\mathcal{A}_S$ induzido pelo LTS $\mathcal{S}$ (linha 2). Na linha 3 é construído o autômato $\mathcal{A}_{compS}$ que reconhece o complemento (Algoritmo 13) da linguagem aceita por $\mathcal{A}_S$. Caso a expressão regular D tenha sido informada, D é transformada no autômato $\mathcal{A}_D$ que aceita a respectiva linguagem (linha 5). Na linha 6 é construído o autômato $\mathcal{A}_{faultD}$ que representa os comportamentos comuns entre $\mathcal{A}_D$ e $\mathcal{A}_{compS}$. A interseção então contém os comportamentos desejáveis que não estão presentes na especificação $\mathcal{A}_S$. Na linha 8 é verificado se a expressão regular F foi informada e então a função $regexToAutomaton$ toma a expressão regular F e constrói o autômato $\mathcal{A}_F$ (linha 9). Já na linha 10 é construído o autômato $\mathcal{A}_{faultF}$ que representa os comportamentos comuns aceitos por $\mathcal{A}_F$ e $\mathcal{A}_S$. A interseção então deve conter os comportamentos de falha presentes na especificação.

Caso as expressões regulares D e F tenham sido informadas, na linha 13 é construído o autômato $\mathcal{A}_T$ que corresponde a união de todos os comportamentos de falha, seja pela ausência de algum comportamento desejável que não está em $\mathcal{S}$ ou pela presença de algum comportamento indesejável que está presente em $\mathcal{S}$, então na linha 14 o modelo de falha é retornado. Caso apenas a expressão F ou D tenha sido informada o modelo de falhas retornado pelo algoritmo será respectivamente $\mathcal{A}_{faultF}$ (linha 17) ou $\mathcal{A}_{faultD}$ (linha 20).

Algoritmo 5 Language-Based Fault Model

input: LTS $\mathcal{S}$ and the regex (D, F)
output: The fault model automaton

```
1: function faultModelLanguage(𝒮, D, F)
2:   A_S = ltsToAutomaton(𝒮)
3:   A_compS = complement(A_S)
4:   if D ≠ ∅ then
5:     A_D = regexToAutomaton(D)
6:     A_faultD = intersection(A_D, A_compS)
7:   end if
8:   if F ≠ ∅ then
9:     A_F = regexToAutomaton(F)
10:    A_faultF = intersection(A_F, A_S)
```

```
11:     end if
12:     if F ≠ ∅ ∧ D ≠ ∅ then
13:         A_T = union(A_faultD, A_faultF)
14:         return A_T
15:     end if
16:     if F ≠ ∅ ∧ D = ∅ then
17:         return A_faultF
18:     end if
19:     if D ≠ ∅ ∧ F = ∅ then
20:         return A_faultD
21:     end if
22: end function
```

3.5 Algoritmos de propósitos de teste

Esta seção apresenta os algoritmos, projetados e construídos neste trabalho, relacionados a geração de casos de teste para implementações caixa-preta. Primeiramente é descrito o algoritmo que constrói o multigrafo conforme a Proposição 3. Com base na especificação $\mathcal{S}$ e um parâmetro m, que indica a classe de implementações a serem testadas, o Algoritmo 6 constrói um autômato que representa o modelo de falhas **ioco** completo.

O multigrafo, inicialmente, é definido como um IOLTS e sua construção se dá a partir do nível 0. O estado inicial do multigrafo é dado por $(s_0, 0)$, ou seja, pelo estado inicial de $\mathcal{S}$ seguido do nível inicial. O número de níveis do multigrafo é dado pelo número de estados em S e pelo parâmetro m, onde cada nível deve contar o mesmo número de estados da especificação.

Entre as linhas 9 até 26 o primeiro nível do multigrafo $\mathcal{D}$ é construído de acordo com a especificação $\mathcal{S}$. Enquanto houver estados em S não visitados (linha 9) remove-se da lista, de estados a serem visitados, o novo estado a ser explorado e para cada transição partindo deste estado é verificado se o estado alcançado já foi explorado. Caso contrário, este estado é adicionado à lista de estados a serem visitados (linhas 12 à 14). Um novo estado e seu respectivo nível é definido entre as linhas 15 e 22, onde a propriedade acíclica é garantida (linha 15), verificando se o novo estado resultará em um ciclo ou um *self-loop*, para então determinar a que nível o novo estado deve pertencer. Nas linhas 23 e 24 o novo estado e uma nova transição são adicionados ao multigrafo. O restante dos dos estados e transições são adicionadas a cada nível, entre as linhas 27 e 36, até que se alcance o total de níveis calculado inicialmente (linha 7).

O estado $fail$ é adicionado ao conjunto de estados do multigrafo. Da linha 38 até 44 para cada estado do multigrafo que não tenha algum dos rótulos de saída definido, novas transições são adicionadas partindo destes estados para o estado $fail$. Por fim a linha 46 define o automato subjacente ao multigrafo $\mathcal{D}$ contendo o estado $fail$ como estado de aceitação.

Algoritmo 6 Multi-graph D

input: IOLTS $\mathcal{S} = (S, s_0, L_I, L_U, T)$ and max state m of implementations
output: Automaton

```
1: function multiGraphD(𝒮, m)
2:     level = 0
3:     𝒟 = (D, d_0, L_I, L_U, Z)
4:     d_0 = (s_0, level)
5:     D = D ∪ d_0
6:     toVisit = s_0
7:     totalLevels = |S| * m + 1
8:     L = (L_I ∪ L_U)
9:     while toVisit ≠ ∅ do
10:        current = Removes a state of the beginning of the queue toVisit
11:        for (current, l, s) ∈ T to any l ∈ L and s ∈ S do
12:            if s ∉ toVisit then
13:                toVisit = toVisit ∪ s
14:            end if
15:            if current = s OR (current, s) ∈ nextLevel then
16:                d = (s, level + 1)
17:            else
18:                if (s, l', current) ∈ T to any l' ∈ L then
19:                    toNextLevel = toNextLevel ∪ (s, current)
20:                end if
21:                d = (s, level)
22:            end if
23:            D = D ∪ d
24:            Z = Z ∪ ((current, level), l, d)
25:        end for
26:    end while
27:    for (d, l, d') ∈ Z, to d = (name_d, level_d) and d' = (name'_d, level'_d) do
28:        while level'_d + 1 < totalLevels do
29:            level_d = level_d + 1
30:            level'_d = level'_d + 1
31:            ini = (name_d, level_d)
32:            end = (name'_d, level'_d)
33:            Z = Z ∪ (ini, l, end)
34:            D = D ∪ ini ∪ end
35:        end while
36:    end for
```

```
37:     D = D ∪ fail
38:     for l ∈ L_U do
39:         for d ∈ D do
40:             if (d, l, d') ∉ Z to any d' ∈ D then
41:                 Z = Z ∪ (d, l, fail)
42:             end if
43:         end for
44:     end for
45:     F = fail
46:     automatonD = (D, d_0, L, Z, F)
47:     return automatonD
48: end function
```

Após a construção do multigrafo, sequências que vão do estado inicial até os estados de falha são extraídas através de um algoritmo de busca em largura. Estas sequências são a base para a construção dos TPs, conforme descrito na Proposição 4. O Algoritmo 7 toma então cada sequência extraída e constrói um TP determinístico, *input-enabled*, *output-deterministic* e acíclico, exceto pelos *self-loops* nos estados *pass* e *fail*. Das linhas 6 a 12 do algoritmo, são adicionados ao IOLTS $\mathcal{TP}$ os estados e as transições induzidas pela sequência extraída do multigrafo. As propriedades *input-enabled* (linhas 14-18) e *output-deterministic* (linhas 19-21) são garantidas através das modificações no $\mathcal{TP}$ induzido pela sequência extraída. Por fim, as transições de *self-loops* são adicionadas nos estados *pass* e *fail*, das linha 23 a 26, rotuladas com os símbolos de L_I.

Algoritmo 7 TP

input: Automaton $\mathcal{D} = (D, d_0, L, Q, F)$,a $word = \{w_0, w_1, \ldots, w_n\}$, input set L_I and output set L_U

output: A IOLTS TP deterministic, input-enabled, output-deterministic, and acyclic except for self-loops at fail and pass states.

```
1: function TP(D, word, L_I, L_U)
2:     TP = (T, t_0, L_U, L_I, Z)
3:     t_0 = d_0
4:     T = pass ∪ t_0
5:     current = d_0
6:     for w ∈ word do
7:         if (current, w, d') ∈ Q to any d' ∈ D then
8:             T = T ∪ d'
9:             Z = Z ∪ (current, w, d')
10:            current = d'
11:        end if
```

```
12:     end for
13:     for t ∈ T − {fail, pass} do
14:         for l ∈ L_I do
15:             if (t, l, t') ∉ Z to any t' ∈ T then
16:                 Z = Z ∪ (t, l, pass)
17:             end if
18:         end for
19:         if (t, l, t') ∉ Z to any t' ∈ T and any l ∈ L_U then
20:             Z = Z ∪ (t, l, pass)
21:         end if
22:     end for
23:     for l ∈ L_I do
24:         Z = Z ∪ (pass, l, pass)
25:         Z = Z ∪ (fail, l, fail)
26:     end for
27: end function
```

3.6 Algoritmos de operações sobre autômatos

Nesta seção são detalhados os algoritmos clássicos de operações sobre linguagens. Apesar de serem conhecidos da teoria clássica de linguagens formais e autômatos, os algoritmos foram detalhadamente projetados neste trabalho com base na literatura [88, 68, 55].

Autômato induzido por LTS

O Algoritmo 8 é responsável por construir o autômato $\mathcal{A}_{\mathcal{S}}$ induzido pelo LTS $\mathcal{S}$. O FSA $\mathcal{A}_{\mathcal{S}}$ é obtido, basicamente pela transformação das transições τ do LTS em transições ϵ, que são adicionadas em δ assim como as demais transições. Além disso, todos os estados S de $\mathcal{A}_{\mathcal{S}}$ são definidos como finais. A função *convertToDeterministicAutomaton* garante que o autômato resultante seja determinístico.

Algoritmo 8 LTS to Automaton

input: LTS $\mathcal{S} = (S, s_0, L, T)$
output: FSA $\mathcal{A}_S = (S, s_0, L, \delta, S)$

```
1: function ltsToAutomaton(S)
2:     A_S = (S, s_0, L, δ, S)
3:     for (p, x, r) ∈ T do
4:         if x = τ then
5:             δ = δ ∪ (p, ϵ, r)
6:         else
7:             δ = δ ∪ (p, x, r)
```

```
8:      end if
9:    end for
10:   return convertToDeterministicAutomaton(A_S)
11: end function
```

Autômato induzido por IOLTS

De forma similar a transformação de modelos LTS para autômatos, o Algoritmo 9 obtém o autômato $\mathcal{A}_S$ induzido pelo IOLTS $\mathcal{S}$. A única diferença é que o conjunto de símbolos L de $\mathcal{A}_S$ é obtido pela união dos símbolos de entrada (L_I) e dos símbolos de saída (L_U) do IOLTS.

Algoritmo 9 IOLTS to Automaton

input: IOLTS $\mathcal{S} = (S, s_0, L_I, L_U, T)$
output: FSA $\mathcal{A}_S = (S, s_0, L, \delta, S)$

```
1: function IOLTSTOAUTOMATON(S)
2:   A_S = (S, s_0, L, δ, S)
3:   L = L_I ∪ L_U
4:   for (p, x, r) ∈ T do
5:     if x = τ then
6:       δ = δ ∪ (p, ε, r)
7:     else
8:       δ = δ ∪ (p, x, r)
9:     end if
10:  end for
11:  return convertToDeterministicAutomaton(A_S)
12: end function
```

Autômato determinístico

O Algoritmo 10 mostra o processo de determinização de um autômato $\mathcal{S}$ através da Construção 3. A linha 3 representa um *state* a ser adicionado no modelo resultante. As linhas de 4 a 6 localizam os estados alcançados por um caminho ϵ a partir do estado inicial s_0. Estes estados são então adicionados no estado inicial r_0 conforme mostra a linha 7. Na linha 8 o estado inicial é adicionado a lista de estados R. Da linha 9 a 33 são adicionados os estados e as transições do autômato $\mathcal{A}_R$. Para cada rótulo do alfabeto (linha 11) e cada estado em *currentState*[ ] (linha 13), na linha 14 é verificado se existem transições em α partindo de s com o rótulo l para algum estado. Vale destacar que as variáveis contendo [ ] correspondem a implementação da estrutura de listas. Caso exista, os estados alcançados são armazenados em s_2*Reached*[ ] e para cada um desses estados alcançados é verificado quais estados são alcançáveis com caminhos ϵ (linha 18 até 22). Os estados alcançados são então adicionados ao estado corrente *state*[ ]. Nas linhas 24 e 25 são adicionados ao autômato $\mathcal{A}_R$ o novo estado e a nova transição. Os estados de aceitação são definidos nas linhas 26 a 30. O estado de $\mathcal{A}_R$ que

contém algum estado em X é definido como estado de aceitação em W. Por fim na linha 34 o autômato determinístico é retornado.

Algoritmo 10 Convert To Deterministic Automaton

input: A automaton $\mathcal{S} = (S, s_0, L, \alpha, X)$
output: The deterministic automaton

```
1:  function convertToDeterministicAutomaton(Ω)
2:      A_R = (R, r_0, L, π, W)
3:      state[ ] = ∅
4:      for (s_0, ε, s_2) ∈ α, to any s_2 ∈ S do
5:          state[ ] = state[ ] ∪ s2
6:      end for
7:      r_0 = state[ ]
8:      R = R ∪ r_0
9:      while R ≠ ∅ do
10:         currentState[ ] = Removes the beginning of the queue R
11:         for l ∈ L do
12:             state[ ] = ∅
13:             for s ∈ currentState[ ] do
14:                 if (s, l, s_2) ∈ α, to any s_2 ∈ S then
15:                     s_2Reached[ ] = s_2Reached[ ] ∪ s_2
16:                 end if
17:             end for
18:             for s_reached ∈ s_2Reached[ ] do
19:                 if (s_reached, ε, s_3) ∈ α, for any s_3 ∈ S then
20:                     state[ ] = state[ ] ∪ s_3
21:                 end if
22:             end for
23:             if state[ ] ≠ ∅ then
24:                 R = R ∪ state[ ]
25:                 π = π ∪ (currentState[ ], l, state[ ])
26:                 for r' ∈ state[ ] do
27:                     if r' ∈ X then
28:                         W = W ∪ state[ ]
29:                     end if
30:                 end for
31:             end if
32:         end for
33:     end while
34:     return A_R
```

35: **end function**

União de Linguagens Regulares

A união de linguagens regulares também é uma operação necessária para aplicar os métodos desenvolvidos neste trabalho. O Algoritmo 11, que implementa a Construção 1, retorna o autômato que reconhece a união das linguagens aceitas por dois autômatos. A linha 2 define o autômato $\mathcal{A}_Y$ que captura a união das linguagens aceitas pelos autômatos $\mathcal{Q}$ e $\mathcal{S}$. Das linhas de 3 a 5, o conjunto de estados e de transições de $\mathcal{A}_Y$ é definido pela união dos estados e transições dos autômatos participantes, bem como o conjunto de estados finais. Também é adicionado um novo estado t_0 ao conjunto de estados, além de transições ϵ do estado inicial t_0 para os estados iniciais s_0 e q_0, respectivamente. Como o autômato gerado é não-determinístico a função *convertToDeterministicAutomaton*, na linha 6, o transforma num autômato determinístico.

Algoritmo 11 Union

input: $\mathcal{Q} = (Q, q_0, L, \mu, E)$ and $\mathcal{S} = (S, s_0, L, \alpha, X)$
output: $\mathcal{A}_Y = (T, t_0, L, \delta, F)$

1: **function** UNION($\mathcal{Q}, \mathcal{S}$)
2: $\quad \mathcal{A}_Y = (T, t_0, L, \delta, F)$
3: $\quad T = Q \cup S \cup t_0$
4: $\quad \delta = \mu \cup \alpha \cup \{(t_0, \epsilon, q_0), (t_0, \epsilon, s_0)\}$
5: $\quad F = E \cup X$
6: $\quad$ **return** convertToDeterministicAutomaton($\mathcal{A}_Y$)
7: **end function**

Intersecção de Linguagens Regulares

A operação de interseção sobre linguagens regulares é parte importante no processo de verificação de conformidade. O Algoritmo 12 constrói o autômato que reconhece a interseção das linguagens aceitas pelos autômatos recebidos por parâmetro. O resultado da interseção (Construção 2) é obtido pela sincronização paralela dos DFA participantes $\mathcal{Q}$ e $\mathcal{S}$. As linhas de 2 a 6 definem o autômato resultante $\mathcal{A}_R$ com o conjunto de estados $R \subseteq S \times Q$, o estado inicial dado pelo par de estados iniciais (s_0, q_0), e o conjunto de estados finais W vazio. As linhas de 7 a 16 adicionam os estados à R e suas respectivas transições sincronizadas em π. Nas linhas de 17 a 21 os estados de aceitação de $\mathcal{A}_R$ são definidos quando ambos os estados sincronizados de $\mathcal{S}$ e $\mathcal{Q}$ também são estados finais.

Algoritmo 12 Intersection

input: $\mathcal{Q} = (Q, q_0, L, \mu, E)$ and $\mathcal{S} = (S, s_0, L, \alpha, X)$
output: $\mathcal{A}_R = (R, r_0, L, \pi, W)$

1: **function** INTERSECTION($\mathcal{Q}, \mathcal{S}$)

```
2:    A_R = (R, r_0, L, π, W)
3:    R ⊆ Q × S
4:    r_0 = (s_0, q_0)
5:    R = {r_0}
6:    W = ∅
7:    while R ≠ ∅ do
8:        Removes r = (s, q) from queue R
9:        for l ∈ L do
10:           if (s, l, s_2) ∈ α for some s_2 ∈ S and (q, l, q_2) ∈ μ for some q_2 ∈ Q then
11:               r' = (s_2, q_2)
12:               π = π ∪ {(r, l, r')}
13:               R = R ∪ {r'}
14:           end if
15:       end for
16:   end while
17:   for r = (s, q) ∈ R do
18:       if q ∈ E ∧ s ∈ X then
19:           W = W ∪ r
20:       end if
21:   end for
22:   return A_R
23: end function
```

Complemento de Linguagens Regulares

Uma outra operação, sobre linguagens regulares, usada na verificação de conformidade é a de complemento, descrita na Construção 4, e obtida pelo Algoritmo 13 que constrói um autômato que reconhece o complemento da linguagem aceita por um dado autômato. Nas linhas de 2 a 5 é definido o autômato $\mathcal{A}_{comp}$, com o conjunto de estados de $\mathcal{A}_{\mathbb{Q}}$ mais um estado especial s_{comp}, e o conjunto de estados finais dado pela complementação do conjunto de estados finais de $\mathcal{A}_{\mathbb{Q}}$. Para cada rótulo de transição não definido em cada estado, uma nova transição é criada a partir desse estado e com este rótulo para o estado especial s_{comp}, como mostra as linhas de 6 a 10.

Algoritmo 13 Complement

input: $\mathcal{A}_{\mathbb{Q}} = (Q, q_0, L, \mu, E)$
output: $\mathcal{A}_{comp} = (C, q_0, L, \delta, F)$

```
1: function COMPLEMENT(A_Q)
2:     A_comp = (C, q_0, L, δ, F)
3:     C = Q ∪ s_comp
4:     F = (C − E)
```

```
    δ = μ
    for q ∈ Q and l ∈ L do
        if (q, l, q2) ∉ μ for some q2 ∈ Q then
            δ = δ ∪ (q, l, s_comp)
        end if
    end for
    return A_comp
end function
```

3.7 Considerações do capítulo

Este capítulo apresentou o método teórico proposto por Bonifácio e Moura [27] que abrange tanto a verificação de conformidade quanto a geração de testes. As principais contribuições deste capítulo estão relacionadas aos algoritmos projetados para o desenvolvimento da EVEREST. Entre os algoritmos construídos estão o de verificação de conformidade **ioco** e baseado em linguagem, de geração de testes, e as operações sobre linguagens. No próximo capítulo são apresentados os trabalhos relacionados à proposta desta pesquisa.

4 TRABALHOS RELACIONADOS

Os principais trabalhos mais intimamente relacionados com essa pesquisa são abordados neste capítulo. Primeiramente, é apresentada uma comparação entre as características dos métodos de geração de testes, verificação de conformidade **ioco** [93] e o método baseado em linguagens [27]. Em seguida, ferramentas similares que implementam a verificação de conformidade e a geração de teste são descritas e comparadas.

4.1 Verificação de conformidade e conjuntos de teste

A abordagem **ioco**, proposta por Tretmans [92], é o método mais conhecido de verificação de conformidade e geração de conjuntos de teste para modelos IOLTS. Tretmans [93] propõe também o método uioco [93] que lida com especificações subespecificadas. Nestas abordagens uma implementação está correta em relação a sua respectiva especificação se os testes derivados e executados na IUT resultam na produção de saídas previstas na especificação.

Uma abordagem mais recente proposta por Bonifácio e Moura [27] abrange a teoria **ioco** e também proporciona uma verificação de conformidade usando linguagens regulares. A Tabela 1 mostra as principais características entre os métodos mencionados.

Tabela 1 – Análise comparativa de métodos.

Aspectos sobre	Tretmans [92, 93]	Bonifácio e Moura [27]
Modelos	IOLTS	IOLTS/LTS
Especificações	subespecificadas / *input-enabled*	subespecificadas / *input-enabled*
IUT	caixa-preta, caixa-branca	caixa-preta, caixa-branca
	input-enabled	subespecificada / *input-enabled*
IUT e especificação	quiescência	quiescência
	não determinismo	não determinismo
Verificação de conformidade	**ioco** /uioco	**ioco** baseado em linguagem
Conjuntos de teste	*Sound* para **ioco** /uioco	m-**ioco** completo
Estratégia de geração	online	online
Estratégia de verificação	offline	offline

A verificação de conformidade e a geração de conjuntos de teste, tanto na teoria **ioco** quanto na uioco, requerem que a especificação e a implementação sejam modelos IOLTSs. Já a verificação de conformidade baseada em linguagem é capaz de lidar com modelos LTSs e IOLTSs, neste caso, não há o interesse explícito em diferenciar os estímulos entre entrada e saída, pois a verificação se dá sobre comportamentos desejáveis e indesejáveis.

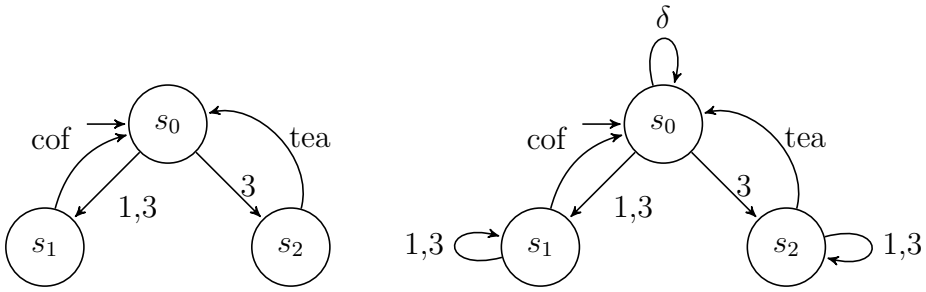
Ambos os métodos aceitam especificações subespecificadas, porém na abordagem de Tretmans [93] a IUT precisa ser *input-enabled*. Além disso, os métodos lidam com modelos contendo estados quiescentes e não determinísticos.

Exemplo 10. *A Figura 11a representa uma máquina de bebida $\mathcal{N}$ modelada por um IOLTS. O alfabeto de entrada é $L_I = \{1, 3\}$ que corresponde as moedas fornecidas pelo usuário de acordo com a bebida desejada, $L_U = \{cof, tea\}$.*

O modelo $\mathcal{N}$ é não determinístico, pois a partir do estado inicial s_0 ao inserir a moeda 3 pode-se alcançar tanto o estado s_1 quanto o estado s_2. O modelo também é subespecificado, pois as entradas $\{1, 3\}$ não estão habilitadas em todos os estados. O modelo contém também um estado quiescente, o estado inicial s_0, que não produz nenhuma saída.

A Figura 11b exibe a IUT $\mathcal{P}$ adaptada para que seja possível a execução dos métodos ***ioco*** *e uioco proposto por Tretmans [93]. Uma transição de self-loop rotulada por δ é adicionada ao estado quiescente s_0, e nos estados subespecificados s_1 e s_2 são adicionados self-loops rotulados com as entradas não habilitadas.*

A abordagem proposta por Bonifácio e Moura [27] lida com estados quiescentes da mesma forma que o método de Tretmans, porém trataria a IUT $\mathcal{P}$ subespecificada sem transforma-la em input-enabled preservando seu comportamento original.



(a) Máquina de bebidas $\mathcal{N}$ (b) Máquina de bebidas $\mathcal{P}$

Figura 11 – Máquina de bebidas

Ambos os métodos suportam a verificação de conformidade para IUTs caixa-branca, quando a estrutura interna é conhecida, e a geração de conjuntos de teste para as IUTS caixa-preta, quando o código fonte e a lógica do sistema não está disponível.

O conjunto de teste gerado a partir do método de Tretmans [93] é *sound* para **ioco**, ou seja, nenhuma implementação correta resulta em falha. No entanto, algumas implementações incorretas, considerando a relação **ioco**, podem passar pelo teste. O método proposto por Bonifácio e Moura [27] obtém conjuntos de teste m-**ioco** completos, sendo capazes de detectar a não conformidade em IUT com até m estados. A verificação de conformidade **ioco** em ambas as abordagens são completas.

A verificação de conformidade, em ambos os métodos, ocorre de forma *offline*, pois os testes são gerados completamente para depois serem executados. Já a geração de conjuntos de teste é *online*, ou seja, cada caso de teste gerado é executado na implementação logo em sequência.

4.2 Ferramentas similares

Existem algumas ferramentas na literatura que lidam com a geração de casos de teste e a verificação de conformidade para modelos formais. Cada ferramenta apresenta suas particularidades quanto ao modelo formal adotado, a relação de conformidade definida e cobertura de falhas. Algumas ferramentas adotam variações da relação de conformidade **ioco** (rioco e sioco), tais como o STG (Symbolic Test Generator) [36], o TorXAkis [73] e o UPPAAL-TRON [64] que lidam com modelos simbólicos e/ou temporizados. Já as ferramentas Torx [23, 47, 20] e TGV [21, 94, 24, 83, 33, 59] implementam a relação de conformidade **ioco** [91] clássica, enquanto o JTorx [21, 20] aborda a teoria *uioco* [95, 93]. As ferramentas Torx, TGV e JTorx lidam com modelos IOLTSs e serão detalhadas a seguir.

Assim como os métodos, as ferramentas de teste descritas nesta seção também podem ser classificadas em *online* ou *offline*. No modo *online* as etapas de geração e execução dos testes ocorrem em conjunto, onde os casos de teste são gerados sob demanda e já executados em sequência. Já o teste em modo *offline* ocorre em duas etapas distintas, primeiramente os casos de teste são gerados a partir da especificação para, em seguida, serem executados no modelo da implementação.

4.2.1 Torx

O Torx é uma ferramenta desenvolvida em C que implementa a relação **ioco** para verificação de conformidade, derivação e execução de testes em modelos IOLTS. A ferramenta oferece tanto a abordagem *online* quanto a *offline*. A vantagem da abordagem *online* é a ausência da necessidade de explorar todo o espaço de estados durante a geração dos testes [23]. A geração do conjunto de teste pode ser guiado por propósitos de teste, exaustivo para **ioco** ou de modo aleatório. Neste último nenhum critério de cobertura é definido e a escolha do caminho percorrido na especificação para a geração dos casos de teste ocorre de forma aleatória. Além disso, o usuário deve definir o limite de profundidade das transições estimuladas na especificação para a geração dos testes [21].

O modo *offline* é mais adequado para os casos de teste elaborados manualmente, já que os humanos são bons na seleção de testes, mas não são tão ágeis em sua execução. Além disso, nos testes *online* toda a computação é realizada em tempo de execução, já no teste *offline* parte do processamento pode ser realizado em tempo de compilação, por isso o método *offline* facilmente satisfaz os requisitos das implementações em tempo real, embora o tamanho do conjunto de teste gerado seja maior e por consequência, seja necessário mais recursos para o armazenamento [22].

No Torx os modelos das IUTs devem ser *input-enabled*, para que sejam capazes de consumir todos os estímulos enviados pelo *tester*, assim como o *tester* deve ser capaz de consumir todas as saídas produzidas pela IUT [23].

No teste *online*, as transições percorridas são estabelecidas de forma aleatória ou através

dos propósitos de teste. Se o propósito de teste ou a abordagem aleatória opta por observar uma ação de saída, as saídas produzidas pelos modelos são observadas a fim de se verificar a conformidade entre os modelos. Se a escolha é feita por uma ação de estímulo, uma entrada é então derivada da especificação e enviada a implementação. Esses passos são realizados até que alguma inconsistência seja capturada, ou seja, quando as saídas da especificação e da implementação não são equivalentes levando a um veredito de falha, ou até que um número pré-estabelecido de etapas de teste seja cumprido, atribuindo-se o veredito de *pass* [23].

Se a interseção entre os conjuntos de ações permitidas do propósito de teste e de ações permitidas pela especificação é vazia, no modo guiado, a ferramenta produz um veredito *missed* que corresponde a um veredito inconclusivo [29]. Caso algum dos caminhos de um propósito de teste seja executado até o fim, então o comportamento é dito bem-sucedido (*hit*). Neste caso o propósito de teste guia a especificação sem ser interrompido como, por exemplo, num cenário em que a especificação não consegue processar a ação realizada pelo propósito de teste. Logo, quando um propósito de teste é aplicado, o veredito é uma tupla $\{pass,\ fail\} \times \{hit,\ miss\}$, onde $\{pass,\ fail\}$ indica se uma falha foi ou não encontrada e $\{hit,\ miss\}$ sinaliza se os caminhos do propósito de teste foram completamente analisados ou não. Note que no modo aleatório o veredito é dado apenas por $\{pass,\ fail,\ inconclusive\}$.

4.2.2 JTorX

O JTorx [4] é uma reimplementação em Java da ferramenta TorX, ambas desenvolvidas pela Universidade de *Twente*. A ferramenta TorX foi desenvolvida em C, o que dificultava a instalação nas diferentes plataformas. O desenvolvimento em Java proporcionou um processo de instalação e configuração mais prático para o usuário, sem a necessidade da instalação de pacotes adicionais e compilação de código fonte.

Uma das características do JTorx é a verificação de conformidade *uioco*, uma noção mais fraca do que a conformidade **ioco**, que lida com especificações subespecificadas. Para se adequar a teoria **ioco** o JTorx, primeiramente, modifica automaticamente o modelo de implementação tornando-o *input-enabled*. A modificação é realizada através da adição de transições de *self-loop* com as entradas não definidas em cada estado. Para modelos da IUT contendo estados quiescentes a ferramenta adiciona *self-loops* rotulados com δ e as ações internas τ também devem ser substituídas por *self-loops* rotulados com δ, pois, a ferramenta processa as ações internas τ como um período de tempo em que o sistema ficou inativo, indicando a ausência de saída [21]. Após a verificação de conformidade, em caso de não conformidade entre os modelos, o JTorx exibe os casos de teste que levam a este veredito além das saídas produzidas pela implementação e especificação.

A ferramenta dispõe de um visualizador gráfico dos modelos de implementação, especificação e propósito de teste. Há também um simulador iterativo que exibe visualmente como a derivação do conjunto de teste é realizada, a partir da escolha entre as ações estimular o modelo ou observar as saídas produzidas. O *tester* pode ainda adotar a abordagem automá-

tica (aleatória), iterativa (o usuário escolhe as ações) ou guiada (propósito de teste). O JTorx oferece também a funcionalidade de verificar se os *Straces*, caminhos subespecificados e com quiescência δ, coincidem com os *UTraces* que não contém caminhos subespecificados.

O modo de teste automático e guiado usando propósito de teste funciona da mesma forma que no Torx. Já o modo manual permite que o usuário escolha a ação a ser executada, seja observar a saída do modelo ou estimular o modelo aplicando uma entrada. O conjunto de casos de teste derivado no JTorx é completo conforme o propósito de teste e a noção de conformidade **ioco**. O veredito apresentado após a verificação de conformidade é similar ao apresentado pelo Torx.

Os modelos IOLTS no Jtorx podem ser especificados em formatos como: Aldebaran (.aut) [31], GRAPHML usando a ferramenta yEd [6], e Jararaca [2].

4.2.3 TGV

O CADP[1] é um conjunto de ferramentas para modelagem de sistemas assíncronos e verificação de modelos. O TGV é uma das ferramentas do CADP para a geração de conjuntos de testes baseada na relação de conformidade **ioco**. Esta ferramenta não apresenta a abordagem aleatória.

O TGV recebe como entrada uma especificação e um propósito de teste *input-enabled*, ambos IOLTS. O propósito de teste deve conter dois estados especiais, o estado *accept*, usado para a seleção de comportamentos que serão testados, e o estado *reject*, que limita os caminhos que não precisam ser testados. Caso o propósito de teste seja subespecificado a ferramenta o torna *input-enabled* adicionando *self-loops* com rótulo '*' em cada estado que não contém alguma entrada especificada. A utilização do propósito de teste garante que os casos de teste gerados sejam *sound*, visto que nenhuma implementação conforme será rejeitada por algum caso de teste.

A ferramenta gera um conjunto de casos de teste através do produto síncrono entre a especificação e o propósito de teste, marcando os estados da especificação com o rótulo *accept* ou *reject*, que resulta num novo modelo IOLTS que então é determinizado. No TGV os vereditos podem ser [83, 59, 60]:

- *pass*: quando a IUT esta em conformidade com a especificação de acordo com o propósito de teste;
- *inconclusive*: quando os comportamentos da IUT estão em conformidade com especificação, porém, não é possível validá-los usando propósito de teste, já que o teste não é exaustivo;
- *fail*: quando a IUT não está em conformidade com a especificação de acordo com o propósito de teste e a relação de conformidade **ioco** ;

O TGV permite que os modelos sejam especificados em LOTOS (via CADP), SDL (via ObjectGéode SDL ou TestComposer), UML (via UMLAUT) e IF (via simulador IF compiler).

4.2.4 Análise das ferramentas

Esta subseção apresenta uma análise comparativa das ferramentas TGV, Torx e JTorx. A Tabela 2 apresenta uma síntese das características dessas ferramentas, classificadas pelos tópicos: verificação de conformidade; geração de conjuntos de teste; estratégia de teste; implementação; hipóteses sobre o modelo; vereditos e tipo de teste.

Tabela 2 – Análise comparativa de ferramentas.

	TGV	Torx	JTorx
Verificação de conformidade			
ioco	-	X	X
Baseada em linguagem	-	-	-
Geração de conjuntos de teste			
Geração de conjunto de teste	X	X	X
Estratégia de teste			
Teste online	X	X	X
Teste offline	X	X	X
Propósito de teste	X	X	X
Abordagem aleatória	-	X	X
Implementação			
Linguagem	C	C	Java
Open source	X	X	X
Hipóteses sobre o modelo			
Lida com modelos subespecificados	X	-	X
Requer modelos *input-enabled*	X	-	X
Lida com quiescência	-	-	X
Vereditos			
Execução de teste	A[1]	A e B[2]	A e B
Conformidade	-	X	X
Tipo de teste			
Teste de caixa branca	-	X	X
Teste de caixa preta	X	X	X

[1] $\{pass, fail, inconclusive\}$.
[2] $\{pass, fail\} X \{hit, miss\}$.

As ferramentas TGV e Torx foram desenvolvidas em C, enquanto o JTorx, é uma reimplementação do Torx, desenvolvido em Java. Apesar de serem *open source*, diversos problemas foram encontrados na obtenção, compilação e uso da das ferramentas. O TGV apresentou problemas na compilação, resultando em vários erros no código fonte e da ausência de bibliotecas. Já o JTorx não dispunha do código fonte completo, com algumas classes ausentes, o que impossibilitou sua utilizando com código baixado.

O Torx e o TGV implementam a noção de conformidade clássica **ioco** [91], enquanto o JTorx possui a relação **ioco** mais recente [93] que lida com modelos subespecificados e quiescentes. No (J)TorX e no TGV a geração de conjuntos de teste é exaustiva em relação à conformidade **ioco** e/ou propósito de teste.

A estratégia de teste *online* e *offline* estão disponíveis nas ferramentas (J)Torx e TGV, que também aceitam propósitos de teste para a geração dos conjuntos de teste. A ferramenta (J)Torx dispõe também da abordagem de exploração aleatória das transições da especificação para a geração do conjunto de teste.

A implementação da relação de conformidade **ioco** nas ferramentas impõe algumas restrições e propriedades sobre os modelos. A ferramenta Torx não aceita modelos subespecificados, já o Jtorx e TGV apesar de lidarem com modelos subespecificados modificam artificialmente estes modelos adicionando *self-loops* para os tornar *input-enabled*. Os estados quiescentes são tratados no Jtorx com a adição de *self-loops* rotulados por δ tanto no modelo da implementação quanto na especificação.

Os vereditos fornecidos pelas ferramentas após a verificação de conformidade indicam a conformidade ou não entre os modelos. Já o veredito produzido pelas ferramenta TGV e (J)Torx, na execução dos testes, podem ser de três tipos: (i) se o teste passou; (ii) não passou; ou (iii) se foi inconclusivo. No (J)Torx, quando o propósito de teste representa o modelo de falhas, o veredito corresponde ao produto síncrono entre $\{pass, fail\}X\{hit, miss\}$, onde $\{pass, fail\}$ indica que teste foi bem sucedido e $\{hit, miss\}$ indica que o propósito de teste foi satisfeito ou não.

O (J)Torx suporta tanto o teste de caixa-banca quanto o teste de caixa-preta. No teste de caixa-preta o JTorx usa o módulo *Adapter* para realizar a comunicação entre a IUT e a ferramenta, através do protocolo TCP ou *User Datagram Protocol* (UDP). A verificação de conformidade é realizada pela comparação das saídas esperadas pela especificação e as produzidas pela implementação real. Já o TGV realiza o teste de caixa-preta através da geração dos casos de teste com base na especificação e no propósito de teste.

4.3 Considerações do capítulo

Este capítulo apresentou os trabalhos relacionados com a pesquisa desenvolvida, que abrange a comparação entre os métodos implementados pelas ferramentas da literatura e pela Everest. A verificação de conformidade e geração **ioco** proposta por Tretmans [92, 93] estão presentes nas ferramentas da literatura, enquanto que a verificação baseada em linguagem e a geração de testes propostas por Bonifácio e Moura [27] foram implementadas na Everest. Uma análise comparativa entre as ferramentas da literatura também foi realizada, mostrando que apenas a teoria clássica **ioco** está presente nessas ferramentas, tanto para verificação de conformidade quanto para a geração de testes. O desenvolvimento da Everest vem preencher esta lacuna com a implementação de métodos mais gerais, que sejam capazes de cobrir e detectar

falhas não cobertas pela abordagem **ioco**. O próximo capitulo introduz a ferramenta EVEREST, descreve o seu processo de desenvolvimento e apresenta sua aplicação na prática.

5 EVEREST : UMA FERRAMENTA DE TESTE

Este capítulo apresenta o projeto e o desenvolvimento da ferramenta EVEREST, que inclui a arquitetura da ferramenta, a análise e projeto UML, além das funcionalidades gráficas e exemplos de aplicação prática com a ferramenta.

5.1 Aspectos do desenvolvimento

A justificativa para a implementação de uma nova ferramenta ao invés de estender alguma já existente, se deve, primeiramente, a exigência de propriedades sobre os modelos, o que não ocorre no método proposto por Bonifácio e Moura [27]. O método engloba tanto a verificação de conformidade quanto a geração de conjuntos de teste para modelos IOLTS. Além disso, o código fonte das ferramentas candidatas, como o JTorx, não está completamente disponível e atualizado, dificultando a implementação de novos módulos.

A ferramenta foi desenvolvida em Java [77] usando o paradigma orientado a objetos. A linguagem Java foi escolhida por ser amplamente conhecida e disseminada tanto no meio acadêmico quanto nas empresas, além de ser multiplataforma, robusta e orientada a objetos. O desenvolvimento da ferramenta foi realizado através da *Integrated Development Environment* (IDE) *Eclipse* [42]. O *Eclipse* foi adotado como IDE por ser um ambiente de desenvolvimento com recursos tanto para a codificação, quanto para depuração e documentação, além de ser uma ferramenta de licença livre. As bibliotecas de código aberto utilizadas neste projeto foram: *brics* [28] versão 1.12-1 para a transformação de expressões regulares em autômatos finitos e *JGraphX* [3] versão 4.0.0 para a geração de imagens a partir dos modelos IOLTS/LTS. A ferramenta está disponível em <https://everest-tool.github.io/everest-site/> sob licença GNU GPL [43].

A análise e o projeto da ferramenta foram realizados com base nos diagramas da UML para garantir maior qualidade ao processo [49]. A arquitetura da ferramenta é apresentada na Figura 12, onde os retângulos representam os módulos da ferramenta, as elipses são as entradas/saídas produzidas pelo sistema e os rótulos nas setas denotam o fluxo de dados entre os módulos.

O módulo *módulo View* foi implementado usando a biblioteca gráfica *Swing* [78] do Java que possui componentes de interface para a elaboração de telas. A interface se divide em quatro visões distintas: configuração; conformidade **ioco** ; conformidade baseada em linguagem; e geração de TPs/execução de teste. Na tela de configuração são informados os modelos que representam a especificação e implementação, em formato Aldebaran, o tipo de modelo (LTS ou IOLTS) a ser tratado, e o particionamento entre rótulos de entrada e saída no caso de modelos IOLTS.

Nas demais visões é possível visualizar graficamente os modelos, além disso, as principais informações de configuração permanecem visíveis. Na tela de conformidade **ioco** e baseada

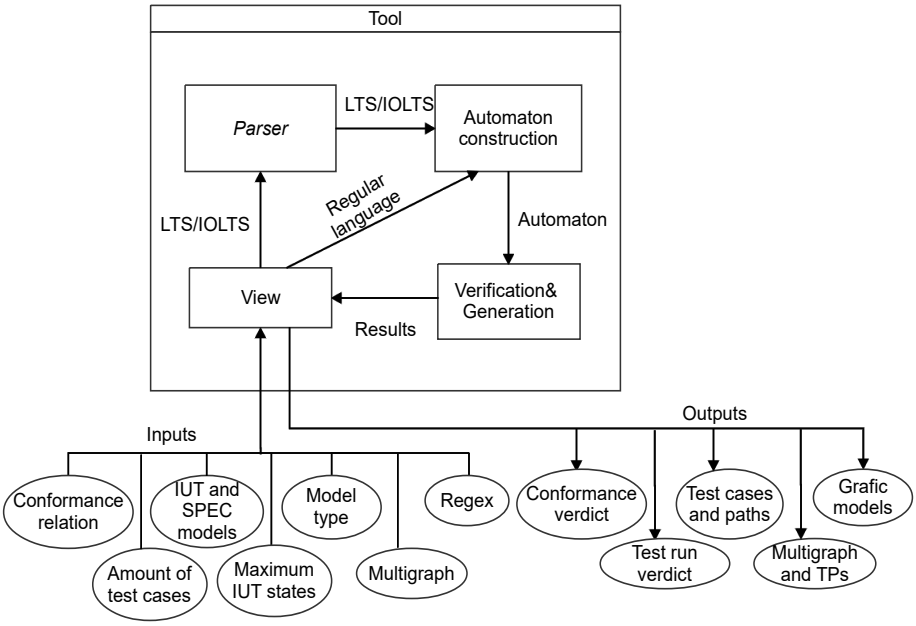


Figura 12 – Arquitetura da ferramenta.

em linguagem, após o processo de verificação, o veredito de conformidade é exibido quando nenhuma falha é encontrada. Em caso de não conformidade a ferramenta apresenta um conjunto de casos de teste, de tal forma que se obtenha a cobertura de transições na especificação, são exibidos também os caminhos induzidos por cada caso de teste no modelo da especificação e implementação. Na conformidade baseada em linguagem, em particular, podem ser fornecidas expressões regulares que representarão os comportamentos desejáveis e as propriedades de falhas. Quando as expressões não são fornecidas, assume-se o fecho Kleene [88] sobre o alfabeto, de modo a identificar falhas quando os modelos não são isomorfos.

Na tela de geração de TPs e execução dos testes a construção do multigrafo só é possível caso sejam fornecidos uma especificação e um limite máximo no número de estados das IUTs a serem cobertas. Num segundo passo, os TPs são construídos a partir de um multigrafo previamente gerado ou através dos parâmetros fornecidos, além da definição do número de TPs a serem gerados. Vale ressaltar que a exploração completa do multigrafo retorna um conjunto de testes completo, porém esbarra nas limitações dos recursos computacionais de processamento e memória devido a sua característica combinatória. No último passo, os TPs gerados são aplicados às IUTs e os vereditos das execuções são salvos em arquivo *csv*.

O *módulo Parser* provê a validação dos modelos selecionados na configuração e a construção dos objetos que representam os modelos na ferramenta. No *módulo Automaton Construction*, os modelos LTS/IOLTS são transformados em seus respectivos autômatos finitos, são construídos também os autômatos que aceitam as linguagens regulares, descritas pelas expressões regulares. Já o *módulo Verification & Generation* implementa o algoritmo de construção do autômato que representa o conjunto de teste, dos autômatos auxiliares que realizam as operações de união, interseção e complemento das linguagens regulares [88], bem como a implementação do processo de verificação de conformidade **ioco** e baseado em linguagem, além dos

algoritmos de construção do multigrafo, geração dos TPs e execução dos testes.

5.2 Análise e projeto de desenvolvimento

A análise e o projeto da ferramenta foi realizado com o suporte da UML, uma notação para especificar e modelar sistemas orientados a objetos. Essa notação é bastante difundida por sua capacidade de especificar requisitos do sistema, comportamentos, estrutura lógica, e dinâmica dos processos, inclusive as necessidades físicas de equipamentos [50]. A UML é composta por vários diagramas, cada um capaz de fornecer uma visão diferente do sistema e proporcionar a cobertura de diferentes aspectos relacionados ao seu desenvolvimento. A análise e o projeto da ferramenta usando os diagramas da UML auxilia no processo de desenvolvimento, proporcionando a descoberta de possíveis falhas além de facilitar a extensão das funcionalidades da ferramenta.

5.2.1 Diagrama de Classe

O diagrama de classes define a estrutura das classes do sistema, especificando seus atributos, métodos e a forma como estas classes estão relacionadas [50]. A Figura 13 representa o diagrama de classes da ferramenta contendo os respectivos métodos a serem implementados.

As principais classes do diagrama são detalhadas a seguir:

State e Transition

Os modelos baseados em transições são compostos por transições e estados. A classe State possui os atributos *id*, *name*, *visited* e *info*. Já a classe Trantisition define a relação entre dois estados, onde *iniState* é o estado de origem da transição, *endState* determina o estado de destino e o atributo *label* identifica a ação executada.

LTS

A classe LTS armazena as informações de um modelo LTS: o conjunto de estados (*states*), o estado inicial (*initialState*), o conjunto de transições (*transitions*) e um alfabeto (*alphabet*). A classe LTS possui duas especializações, as classes *Automaton* e $IOLTS$.

Automaton

Essa classe, derivada da classe LTS, armazena as informações de um autômato. O atributo extra $finalStates$ representa o conjunto de estados finais ou estados de aceitação. O método *isDeterministic* verifica se o autômato é determinístico, o método *getStatesWithEpsilonTransition* retorna uma lista de estados que possuem transições rotuladas com ϵ e o método $reachableStatesWithEpsilonTransition$ retorna os estados alcançados através de transições ϵ.

IOLTS

A classe $IOLTS$, também derivada da classe LTS, armazena as informações de um modelo IOLTS e tem como atributos extras o alfabeto de entrada (*inputs*) e o alfabeto

de saída (*outputs*). Entre os métodos da classe IOLTS estão o *ioltsToAutomaton* que constrói o autômato subjacente ao IOLTS, o método *outputsOfState* que retorna as saídas produzidas a partir de um estado, o método *toLts* que transforma um IOLTS em um LTS subjacente, o método *addQuiescentTransitions* que adiciona as transições de $self-loop$ rotuladas com δ aos estados quiescentes, o método *isInputEnabled* que verifica se o IOLTS tem todas as entradas habilitadas em cada estado, e o método *isQuiescent* que verifica se um estado é quiescente.

Operations

A classe *Operations* implementa as operações sobre linguagens, conforme as construções apresentadas na Seção 2.4. Através dos métodos *union* e *intersection* são construídos os autômatos que reconhecem a união/interseção das linguagens aceitas por dois autômatos. O método *complement* retorna o autômato que reconhece o complemento da linguagem aceita por um autômato, e o método *regexToAutomaton*, utiliza a biblioteca *dk.brics* para retornar o automato que aceita a expressão regular passada por parâmetro. Também é implementado o método *convertToDeterministicAutomaton* que transforma um autômato não-determinístico em determinístico e o método *path*, que retorna os caminhos percorridos nos modelos para todo caso de teste aplicado.

A biblioteca *dk.brics*, denotada pela anotação $<< external >>$, transforma uma expressão regular em um autômato subjacente que aceita a mesma linguagem regular, através da chamada ao método *toAutomaton*.

ioco Conformance

Esta classe implementa a verificação de conformidade **ioco**. O método *modelD* retorna um autômato contendo os comportamentos desejáveis na implementação, enquanto o método $faultModelIoco$ constrói o modelo de falhas com base na especificação. O método $verifyIOCOConformance$, através dos métodos *modelD* e $faultModelIoco$, verifica se uma implementação esta em conformidade com a especificação.

LanguageBasedConformance

Esta classe implementa a verificação baseada em linguagens usando expressões regulares para especificar os comportamentos desejáveis e indesejáveis à implementação. O método $faultModelLanguage$ constrói o autômato do modelo de falhas baseado na especificação e nas expressões regulares D e F que correspondem, respectivamente, aos comportamentos desejáveis e indesejáveis. O método $verifyLanguageConformance$, através do modelo de falhas construído, verifica se a implementação está em conformidade com a especificação.

TestGeneration

Esta classe implementa a construção do multigrafo, a geração dos TPs e a execução dos testes. O método *multiGraphD* constrói o multigrafo com base no modelo da especificação e no parâmetro m, que define um número máximo de estados das IUTs. Já o método $testPurpose$ constrói um IOLTS que representa um TP, dado um multigrafo e uma palavra

extraída deste. Os métodos *run*, *runAllIutTp* e *runIutTp* são responsáveis por executar os TPs extraídos do multigrafo em IUTs, e o método *saveOnCSVFile* gera um arquivo csv contendo o veredito da execução dos testes.

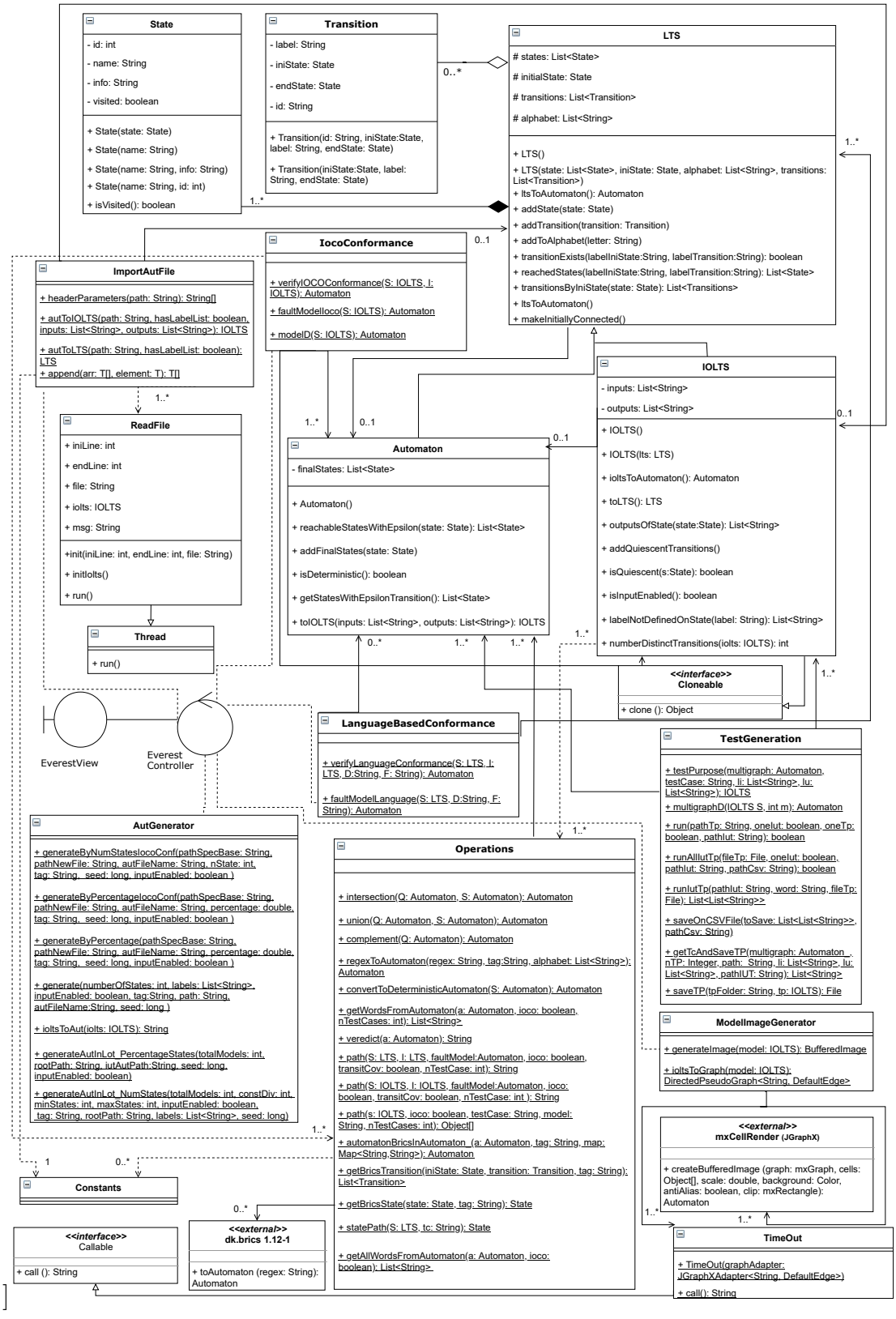


Figura 13 – Diagrama de classe.

5.2.2 Diagrama de Pacote

O diagrama de pacotes descreve a organização das classes em pacotes. A Figura 14 exibe como os pacotes da ferramenta EVEREST estão organizados.

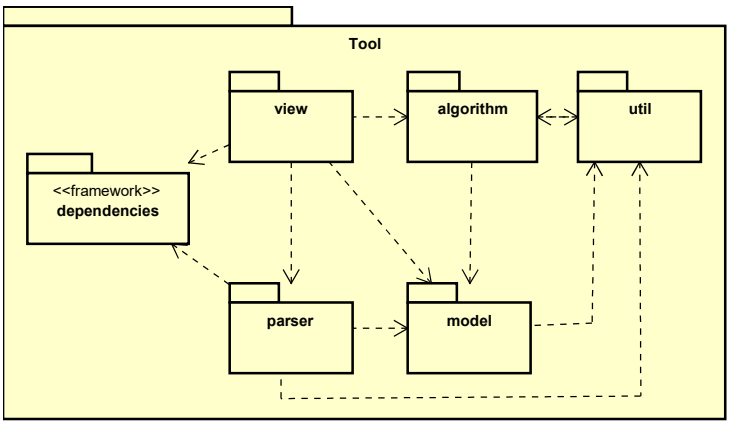


Figura 14 – Diagrama de pacotes.

As classes que constam em cada pacote são detalhadas a seguir, de acordo com a arquitetura da ferramenta (Figura 12).

View

O módulo *View* é composto essencialmente pela interface do sistema e pela classe *EverestController*. Através da interface é possível visualizar graficamente os modelos Aldebaran (Classe *ModelImageGenerator*). Para a verificação de conformidade devem ser informados, através da interface, o tipo de conformidade (**ioco** ou baseada em linguagem), o tipo de modelo (IOLTS ou LTS), o particionamento dos rótulos de entrada e saída nos modelos IOLTS, as expressões regulares no caso da conformidade baseada em linguagem, além dos modelos de implementação e especificação em arquivo Aldebaran. Para a geração dos TPs é necessário informar, a partir da interface, a especificação e o número de estados das IUTs a serem cobertas. Para a execução dos testes devem ser informados os TPs e as IUTs em formato Aldebaran. A Figura 15 apresenta o diagrama de pacote correspondente ao módulo *View*.

Parser

O módulo *Parser* é responsável pela conversão dos modelos que constam nos arquivos Aldebaran em objetos LTS ou IOLTS. Na Figura 16 é possível observar que a classe $ImportAutFile$, em caso de modelos IOLTS, utiliza os atributos $INPUT\_TAG$ e $OUTPUT_TAG$ da classe *Constants* para distinguir as transições de entrada e saída ('!' denota uma saída e '?' denota uma entrada), ou uma lista de rótulos de entradas e outra de saídas devem ser informadas. Os métodos $autToLTS$ e $autToIOLTS$, da classe $ImportAutFile$, são os responsáveis por ler o arquivo recebido pela interface e retornar o respectivo modelo LTS ou IOLTS, conforme as classes do pacote *Model*. A classe *ReadFile* é responsável pela leitura do arquivo Aldebaran.

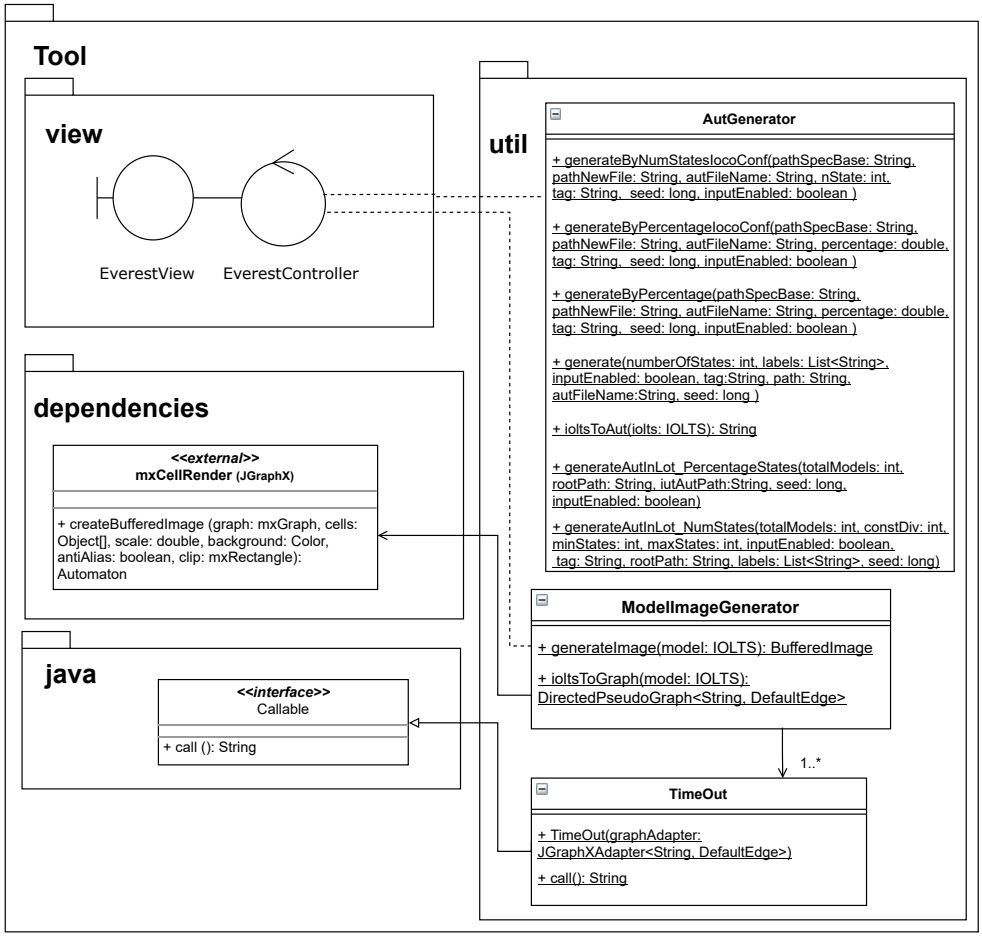


Figura 15 – Diagrama de pacote: módulo *View*.

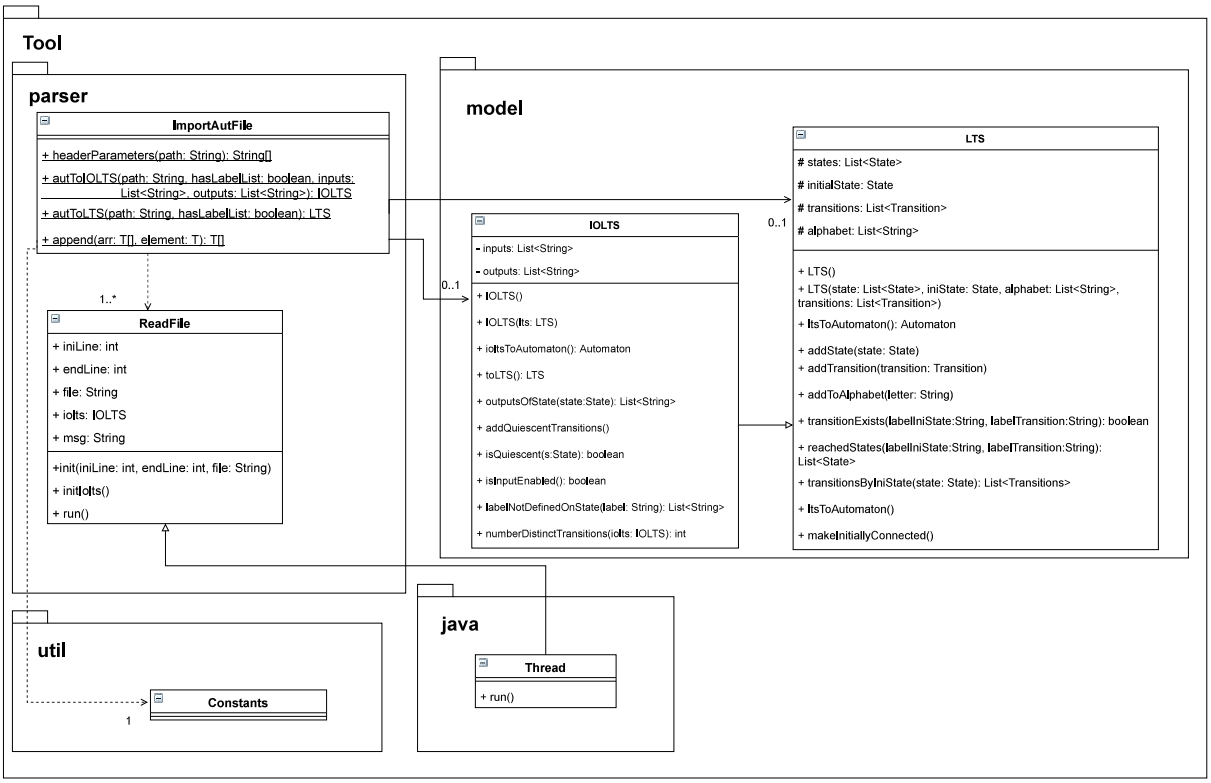


Figura 16 – Diagrama de pacote: módulo *Parser*.

Automaton Construction

A Figura 17 ilustra o módulo *Automaton Construction* responsável por transformar os modelos LTS/IOLTS em autômatos subjacentes e também pela conversão das expressões regulares em autômatos que aceitam as respectivas linguagens. Os métodos responsáveis pela transformação dos LTS/IOLTS em autômatos subjacentes são: $ltsToAutomaton$ (Algoritmo 8) da classe LTS e o $ioltsToAutomaton$ (Algoritmo 9) da classe $IOLTS$. A conversão das expressões regulares é realizada pelo método $regexToAutomaton$ da classe *Operations* com o auxílio da biblioteca *dk.brics*, que consta no pacote *dependencies*.

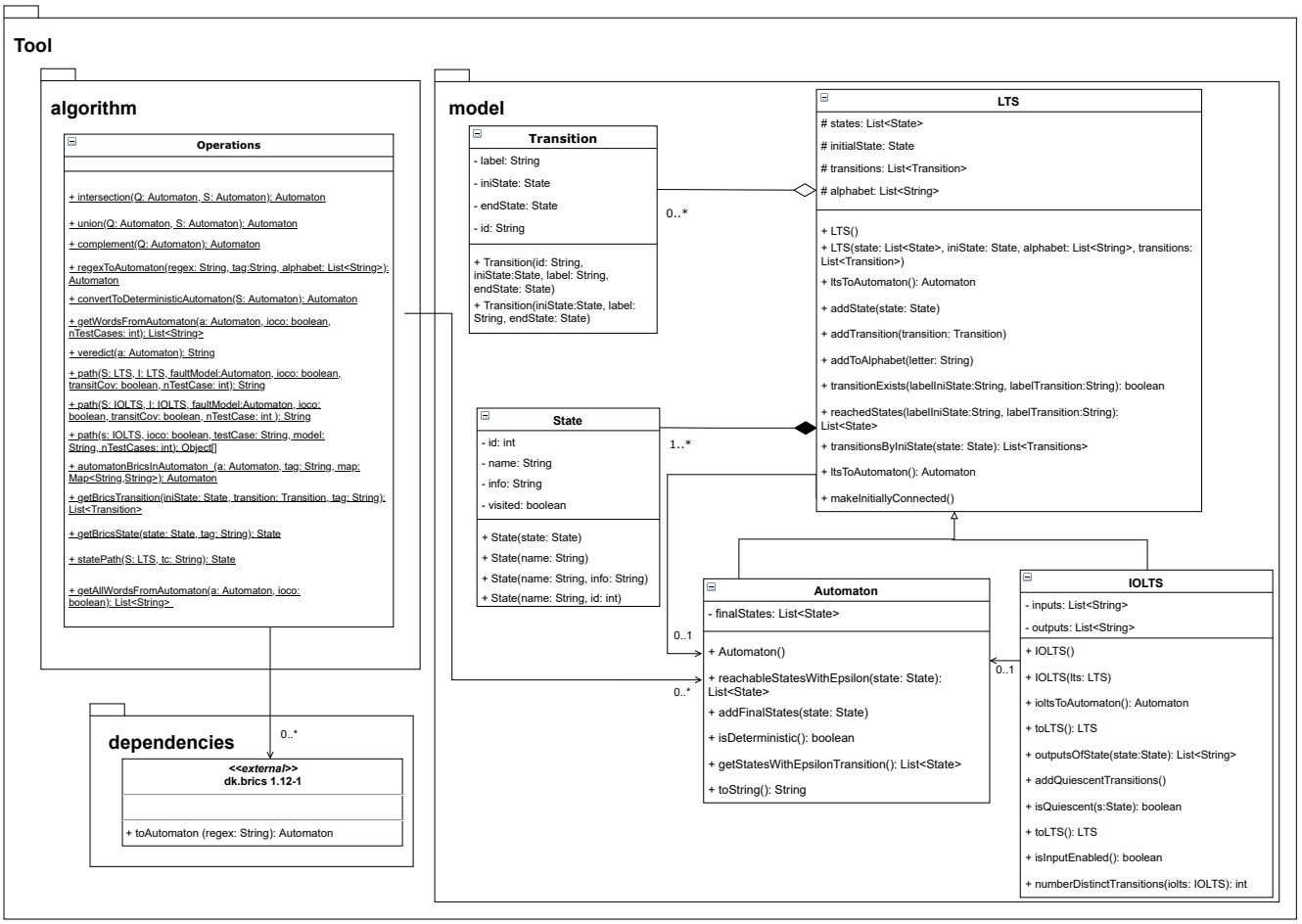


Figura 17 – Diagrama de pacote: módulo *Automaton Construction*.

Verification & Generation

O módulo *Verification & Generation* verifica a conformidade, tanto **ioco** quanto baseada em linguagem, entre os modelos de implementação e especificação, além de ser responsável pela geração de TPs e execução dos testes. O diagrama da Figura 18 mostra as principais classes, organizadas em pacotes, utilizadas na verificação de conformidade, geração de TPs e execução dos testes. Além dos métodos das classes $LanguageBasedConformance$ e $IocoConformance$, os métodos *intersection* (Algoritmo 12), *union* (Algoritmo 11), *complement* (Algoritmo 13), $regexToAutomaton$ e $convertToDeterministicAutomaton$ (Algoritmo 10) da classe *Operations* também são usados no processo de verificação de conformidade. A classe *TestGeneration* é

utilizada na geração dos TPs a partir dos métodos *multigraphD* e *TestPurpose*. A execução dos testes ocorre por meio dos métodos *Run*, *RunIutTP* e *RunAllIutTP*.

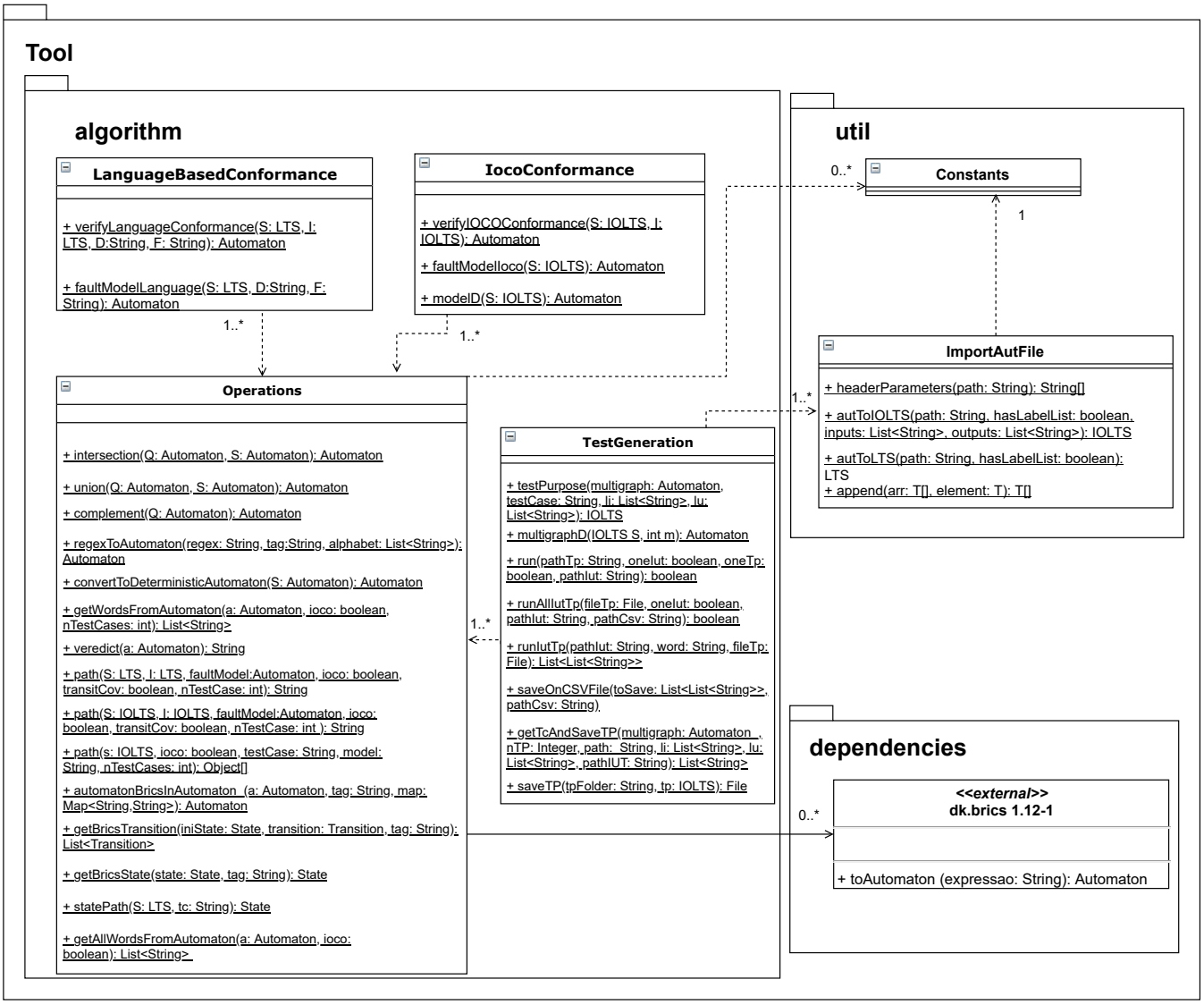


Figura 18 – Diagrama de pacote: módulo *Verification & Generation*.

5.2.3 Diagrama de Sequência

O diagrama de sequência exibe as interações entre os objetos, de acordo com uma ordem cronológica, chamadas de métodos e mensagens de retorno [50]. No diagrama de sequência o ator, *Tester*, representa entidades externas que interagem com o sistema. A seguir são exibidos os diagramas de sequência que descrevem o processo de verificação de conformidade **ioco** e da conformidade baseada em linguagem, além da geração e execução dos TPs.

Conformidade ioco

Na Figura 19 o *Tester* inicia o processo de verificação de conformidade **ioco**, informando

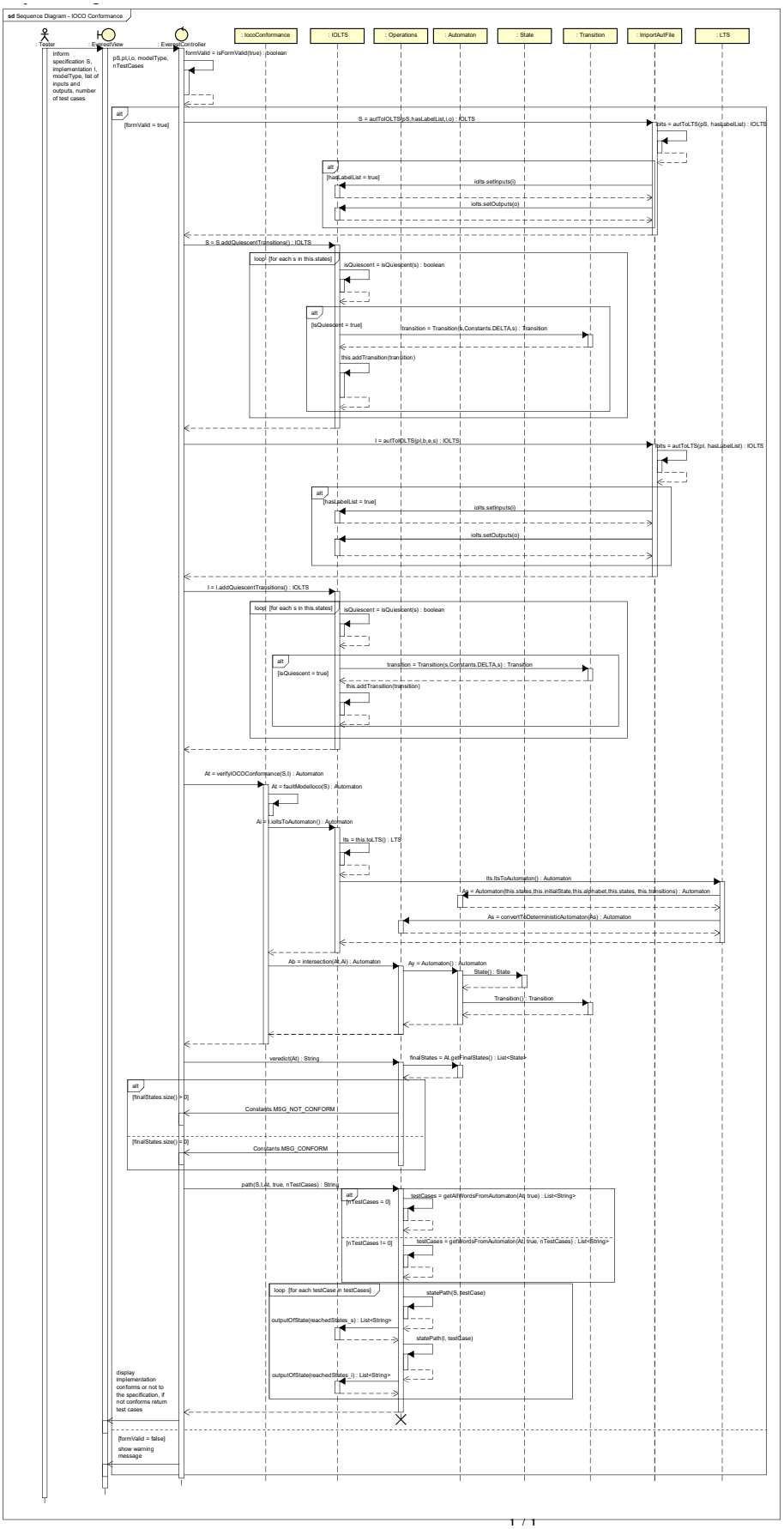


Figura 19 – Diagrama de sequência: ***ioco*** *Conformance*.

à interface o diretório em que estão os modelos da especificação (pS) e da implementa-

ção (pI), o tipo de modelo $modelType$ (LTS/IOLTS), as listas de rótulos de entrada e saída (i,o) quando necessário, além do número de casos de teste a serem gerados em caso de não conformidade ($nTestCases$).

A $EverestView$ então repassa os parâmetros informados pelo *Tester* para o controlador que verifica se os dados estão corretos. Quando os dados estão incorretos a $EverestView$ envia um alerta. Caso contrário, o evento $autToIOLTS$ é disparado para converter os modelos Aldebaran em IOLTSs. Após a construção dos modelos $IOLTSs$ o método $addQuiescentTransitions$ verifica a existência de estados quiescentes (método *isQuiescent*) e adiciona as transições δ.

O evento $verifyIOCOConformance$, da classe $IocoConformance$, verifica a conformidade **ioco** a partir da construção do modelo de falha baseada na especificação (método *faultModelIoco*). O evento *ioltsToAutomaton* é então disparado para construir o autômato subjacente a implementação IOLTS, transformando o modelo IOLTS em LTS pelo método *toLTS*, que em seguida é transformado num autômato determinístico (método *ltsToAutomaton*). Por fim o evento $intersection$ é disparado pela classe $IocoConformance$, retornando um autômato que captura a interseção das linguagens aceitas pelos autômatos do modelo de falhas e da implementação.

O autômato *At*, retornado pelo evento $verifyIOCOConformance$, é passado por parâmetro na chamada do método *veredict* que retorna à interface se a implementação é **ioco** conforme ou não conforme a especificação. Em caso de não conformidade, os casos de teste são exibidos por meio do método *path*, que retorna para o controlador as saídas e os caminho percorridos pela implementação e pela especificação para cada caso de teste.

A Figura 20 ilustra o diagrama de sequência da construção do autômato que representa o modelo de falhas, instanciado no diagrama da Figura 19.

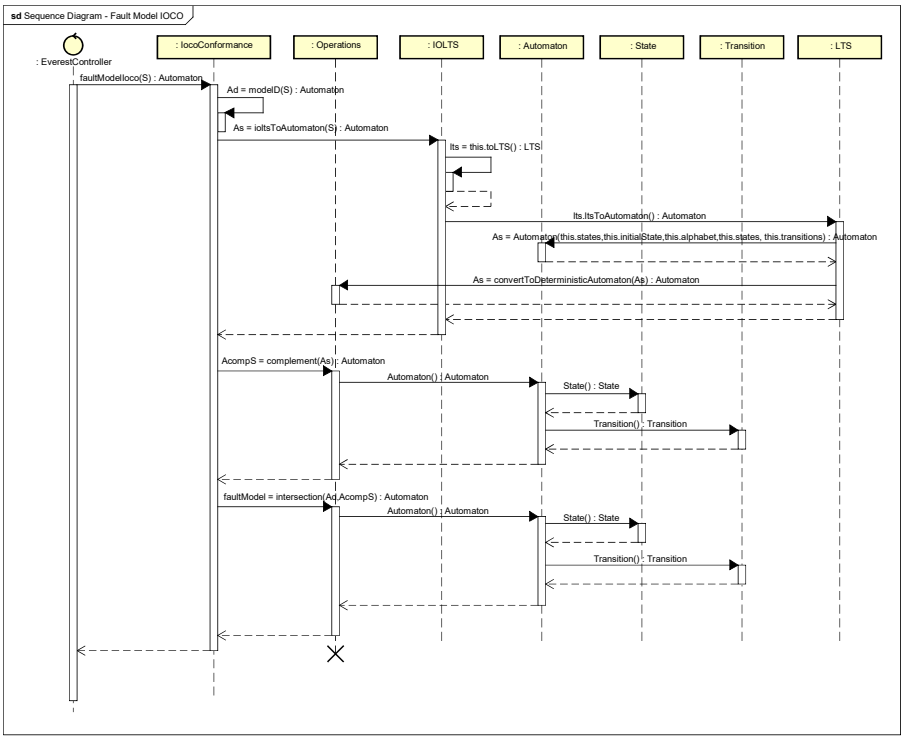


Figura 20 – Diagrama de sequência: *Fault Model* ***ioco***.

O evento $faultModelIoco$ passa por parâmetro a especificação S e constrói o modelo de falhas através do autômato Ad (evento $modelD$) contendo os comportamentos desejáveis de acordo com a noção de conformidade **ioco**. O autômato subjacente da especificação também é construído pela chamada do evento $ioltsToAutomaton$ e o método *complement* retorna o autômato complemento da especificação. O último evento disparado pela classe $IocoConformance$ é o $intersection$, para a interseção entre os autômatos Ad e complemento da especificação, que resulta no modelo de falhas **ioco**.

Conformidade Baseada em Linguagem

A Figura 21 apresenta o diagrama de sequência referente a funcionalidade de verificação de conformidade baseada em linguagem. O *Tester* fornece à $EverestView$, por meio da interface, o diretório em que estão os modelos da especificação (pS), da implementação (pI), o tipo de modelo (IOLTS ou LTS – $modelType$), as expressões regulares D e F, as listas de rótulos de entrada e saída (i,o) quando necessário e o número de casos de teste a serem gerados em caso de não conformidade ($nTestCases$). A $EverestView$ repassa os elementos fornecidos pelo *Tester* para o controlador ($EverestController$) que verifica se os dados inseridos na interface são válidos através do método $isFormValid$. Se os dados estiverem incorretos um alerta é retornado. Caso contrário, é verificado o tipo de modelo e disparado o método $autToLTS$ ou $autToIOLTS$, responsável por converter a especificação e a implementação em modelos $LTS/IOLTS$.

Quando os modelos são IOLTS é verificado, através da variável $hasLabelList$, se o *Tester* informou as listas de rótulos de entrada e saída ou se no arquivo Aldebaran já está especificado as entradas e saídas. O método $addQuiescentTransitions$ é executado, quando o $modelType$ é IOLTS, para encontrar os estados quiescentes (método $isQuiescent$) e adicionar transições δ.

O evento $verifyLanguageConformance$, da classe $LanguageBasedConformance$, verifica se os modelos estão em conformidade dado as expressões regulares D e F. O processo de verificação é similar à verificação de conformidade **ioco**, o modelo de falhas é construído (método $faultModelLanguage$), a implementação LTS é transformada em autômato (método $ltsToAutomaton$) e então o autômato At é construído de modo a capturar a interseção das linguagens aceitas pelos autômatos da implementação e do modelo de falhas. O método $veredict$ retorna o veredito de conformidade de acordo com a quantidade de estados finais (método $getFinalStates$) do autômato At. O método $path$, em caso de não conformidade, retorna os caminhos percorridos pela implementação e pela especificação para todo caso de teste, através do métodos $getWordFromAutomaton$ ou $getAllWordFromAutomaton$ que recuperam as palavras aceitas pelo autômato At.

A Figura 22 ilustra o processo de construção do autômato que representa o modelo de falhas, instanciado no diagrama da Figura 21. O processo se inicia com o controlador disparando

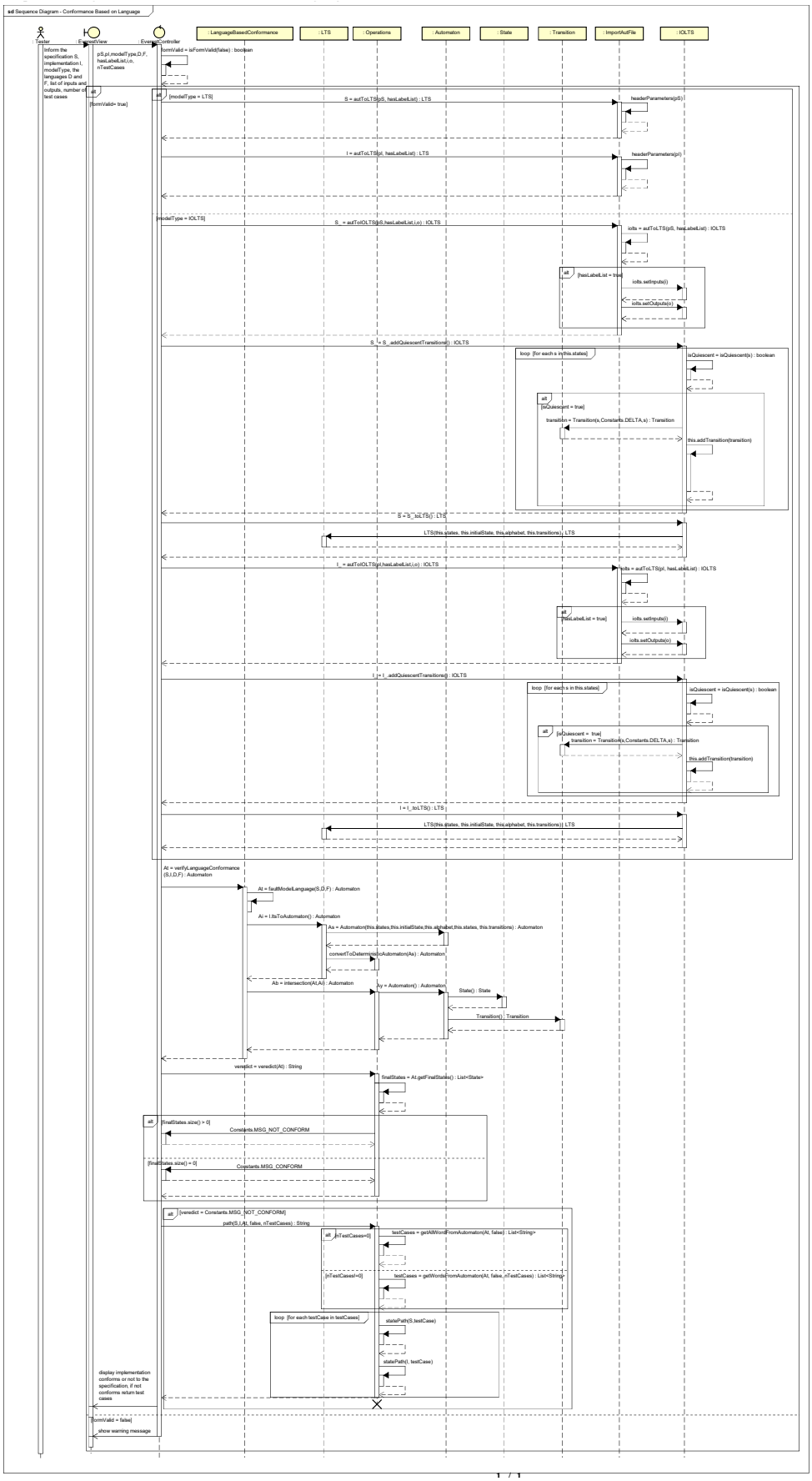


Figura 21 – Diagrama de sequência: *Conformance Based-Language*.

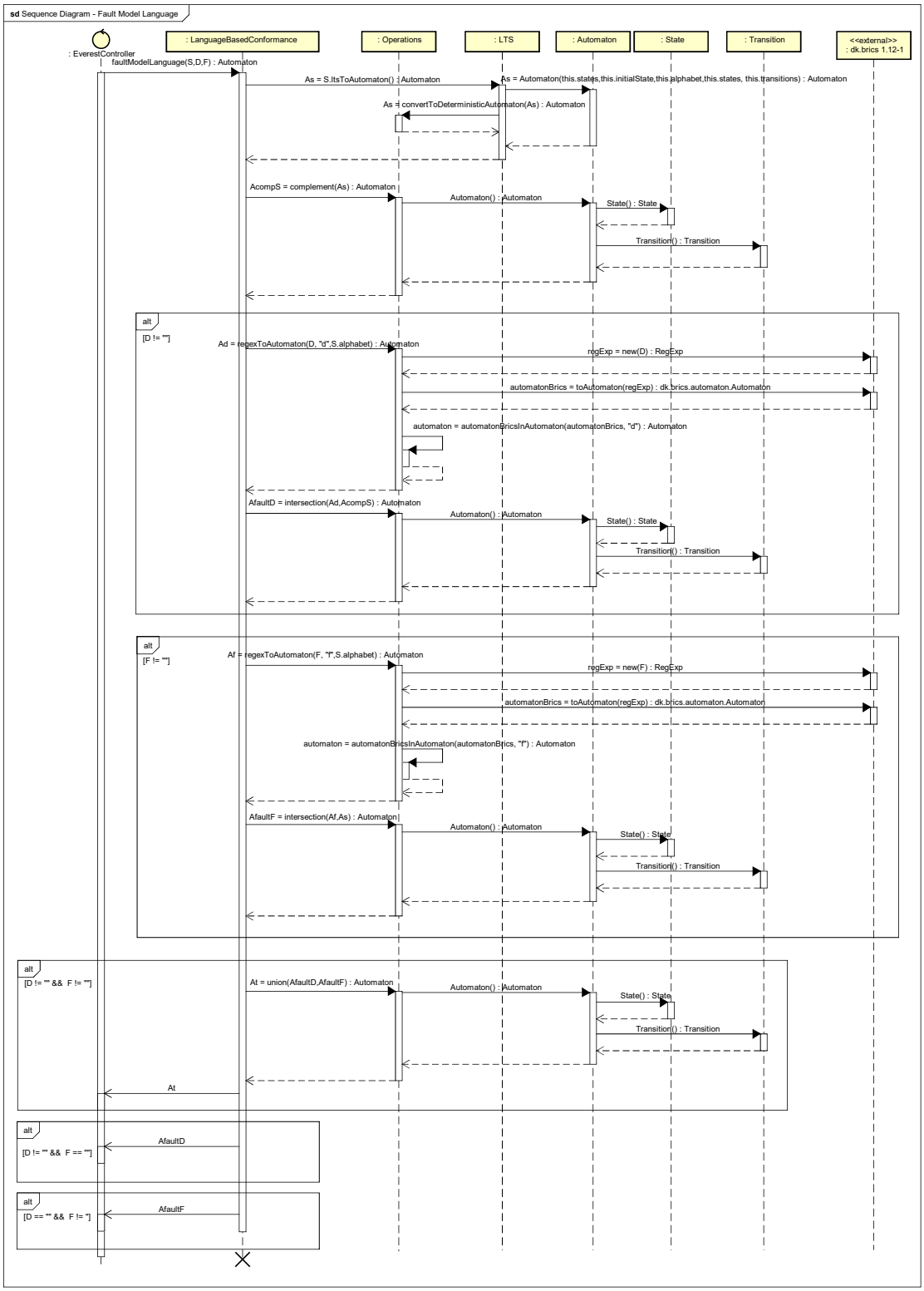


Figura 22 – Diagrama de sequência: *Fault Model Language*.

o método *faultModelLanguage* que chama o evento *ltsToAutomaton* para construir o autômato *As* subjacente a especificação *S*. Em seguida, o evento *complement* retorna o autômato *AcompS* que reconhece o complemento da linguagem aceita pelo autômato recebido por parâmetro.

No passo seguinte as expressões regulares, informadas pelo *Tester*, são transformadas

em autômatos que aceitam as respectivas linguagens através do evento *regexToAutomaton* da classe *Operation*. Este evento cria uma instância *RegExp*, passando por parâmetro uma expressão regular, e o método *toAutomaton* da biblioteca *dk.brics* é acionado de modo a retornar um objeto *dk.brics.automaton.Automaton*, o método *automatonBricsInAutomaton* por sua vez converte o objeto retornado em *Automaton*. Logo as expressões regulares D e F são transformadas respectivamente nos autômatos Ad e Af .

O evento *intersection* da classe *Operations* é chamado duas vezes, primeiro para a interseção entre os autômatos Ad e o complemento da especificação, que resulta no autômato $AfaultD$, e depois para a interseção entre os autômatos Af e a especificação, resultando no autômato $AfaultF$. Na sequência o evento *union* retorna a união das linguagens aceitas pelos autômatos que reconhecem as intersecções anteriormente obtidas. O autômato resultante da união contem os comportamentos desejáveis que não constam na especificação e os comportamentos indesejáveis contidos na especificação. Quando apenas D é informado o modelo de falha retornado é $AfaultD$ e quando apenas F é informado o modelo de falha retornado é $AfaultF$.

Geração e Execução de Teste

A Figura 23 apresenta o diagrama de sequência da geração de TPs e da execução dos testes em IUTs. O *Tester* informa através da interface do sistema (*EverestView*) os parâmetros: número máximo de estados nas IUTs a serem cobertas (m); diretório da especificação (pS); diretório da IUT (pI); diretório do multigrafo (*pMultigraph*); a quantidade de TPs que serão gerados (nTP); rótulos de entrada e saída dos modelos (i, o); e diretório dos TPs (pTP).

Os dados preenchidos na interface são repassados ao controlador (*EverestController*) e validados através do método *isFormValidGeneration*. Caso haja alguma informação inválida uma mensagem é exibida na interface. Após a validação das informações, o *Tester* deve selecionar um diretório para salvar (método *chooseFolderToSave*) as saídas a serem produzidas pela Everest, e o método *autToIOLTS* é executado para converter o modelo da especificação Aldebaran (pS) em um objeto IOLTS. Existem quatro formas de se executar as funcionalidades de geração e execução dos testes: (i) dados a especificação e o máximo de estados das IUTs a serem cobertas, o multigrafo é gerado através do método *multigraph*; (ii) dados um multigrafo e o número de TPs desejados, os TPs são gerados pelo método *getTcAndSaveTP*; (iii) dados o multigrafo, o número de Tps desejados e a IUT, os TPs são gerados e aplicados a IUT usando a estratégia online; e (iv) dados um conjunto de TPs e uma IUT, os testes são aplicados a IUT através do método *run*.

5.3 Aplicação da ferramenta teste

Nas próximas subseções são detalhados os aspectos relacionados ao desenvolvimento da ferramenta, tais como o formato dos modelos, a interface do sistema e sua usabilidade em aplicação prática.

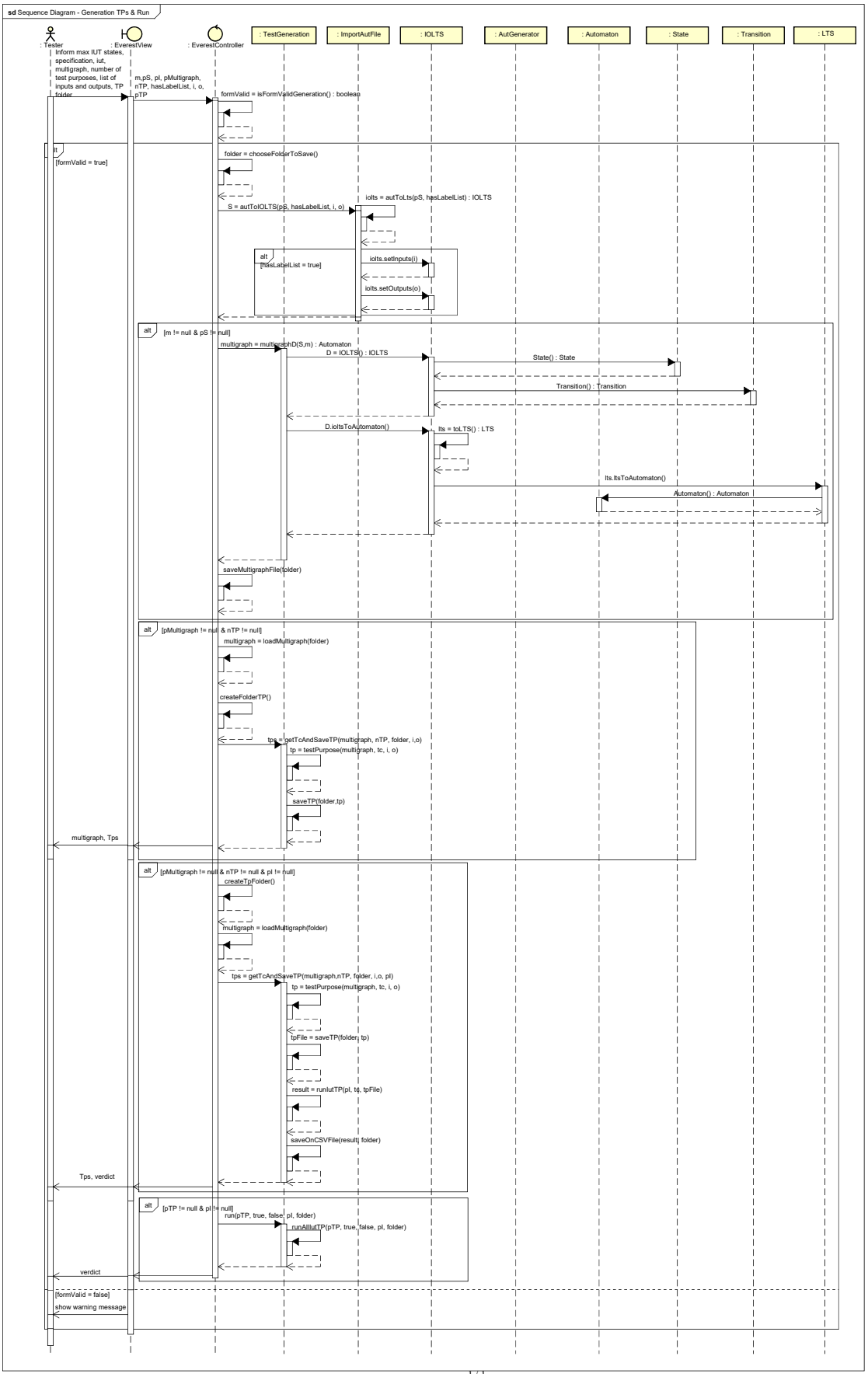
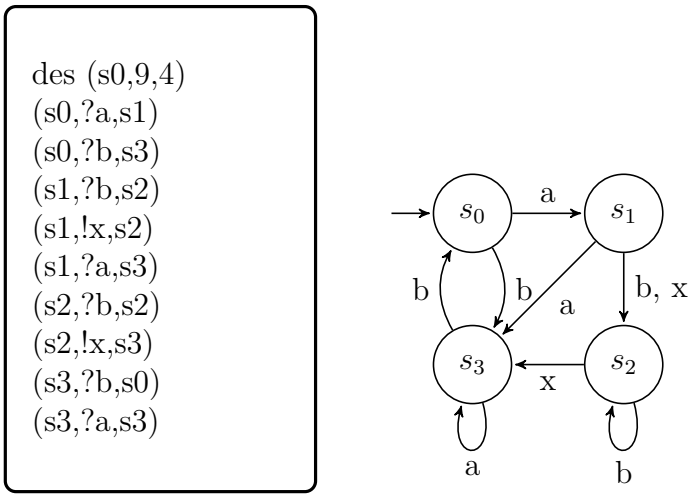


Figura 23 – Diagrama de sequência: *Generation TPs & Run*.

5.3.1 Especificação em Aldebaran

O processo de verificação de conformidade, geração e execução dos testes requer uma representação padrão dos modelos para que a ferramenta possa os reconhecer. Um modelo LTS ou IOLTS pode então ser modelado no formato Aldebaran[31] como um conjunto de transições.

A Figura 24a apresenta um exemplo de arquivo Aldebaran e a Figura 24b exibe o IOLTS subjacente ao modelo Aldebaran.



(a) Exemplo de IOLTS em formato *.aut*.

(b) Especificação do Aldebaran.

Figura 24 – Exemplo de arquivo Aldebaran e modelo subjacente.

Definição 19. *[83] Os modelos em formato Aldebaran são determinados por:*

des(<estado-inicial>, <numero-transicoes>, <numero-estados>)

(<estado-ini>, <rotulo>, <estado-fim>)

O estado inicial é definido no cabeçalho do arquivo através do campo <estado-inicial>. No cabeçalho também são estabelecidos valores numéricos que correspondem a quantidade de transições (<numero-transicoes>) e quantidade de estados (<numero-estados>). Em seguida, o conjunto de transições do modelo é definido pelo nome do estado de origem (<estado-ini>), pelo rótulo da transição (<rotulo>) e o nome do estado de destino (<estado-fim>).

5.3.2 Interface da Ferramenta

A interface da ferramenta EVEREST é composta por quatro visões: configuração; conformidade **ioco**; conformidade baseada em linguagem; geração e execução dos testes.

A Figura 25 apresenta a interface de configuração, onde são informados os modelos que representam a especificação e a implementação, em formato Aldebaran, respectivamente nos campos *Model* e *Implementation*. O tipo de modelo é definido no item *Model type*. Quando se

trata de um LTS, os campos *Label*, *Input labels* e *Output labels* ficam ocultos. Quando se trata de um IOLTS é necessário informar como os rótulos de entrada/saída serão distinguidos (item *Label*), seja informando nos campos *Input labels* e *Output labels* os rótulos de entrada e saída, ou no próprio arquivo Aldebaran da especificação/implementação por meio dos marcadores '!' e '?'.

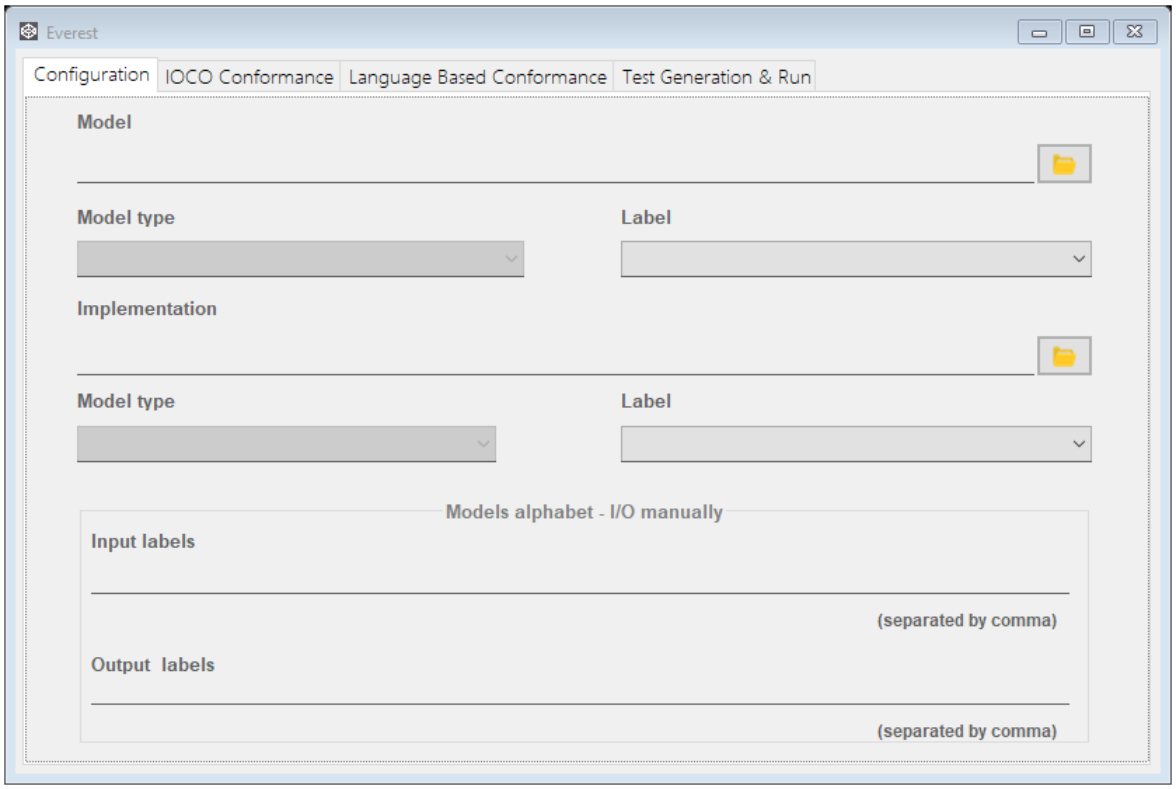


Figura 25 – Interface: configuração.

A Figura 26 apresenta a interface para a verificação de conformidade **ioco**. No cabeçalho da interface são mantidos os campos informados na parte de configuração, como os itens *Model* e *Implementation* que exibem os nomes dos arquivos utilizados como especificação e implementação, além dos campos *Input Label* e *Output Label* que apresentam os rótulos de entradas e saídas dos modelos. Os botões *view model* e *view IUT* permitem a visualização gráfica dos modelos da especificação e implementação. A Figura 27 apresenta um exemplo de visualização para o modelo de especificação da Figura 24.

Na verificação de conformidade **ioco** pode ser informado, no campo *#Test cases*, o número de casos de teste a serem gerados em caso de não conformidade. Ao clicar no botão *Verify*, o veredito é apresentado em forma de texto. Em caso de não conformidade a ferramenta apresenta (no campo de texto abaixo do botão) um conjunto de casos de teste, com base na cobertura de transições da especificação, além dos respectivos caminhos induzidos nos modelos. O item *Warnings* informa os campos preenchidos de forma incorreta na interface de configuração, que são essenciais para que a verificação seja realizada.

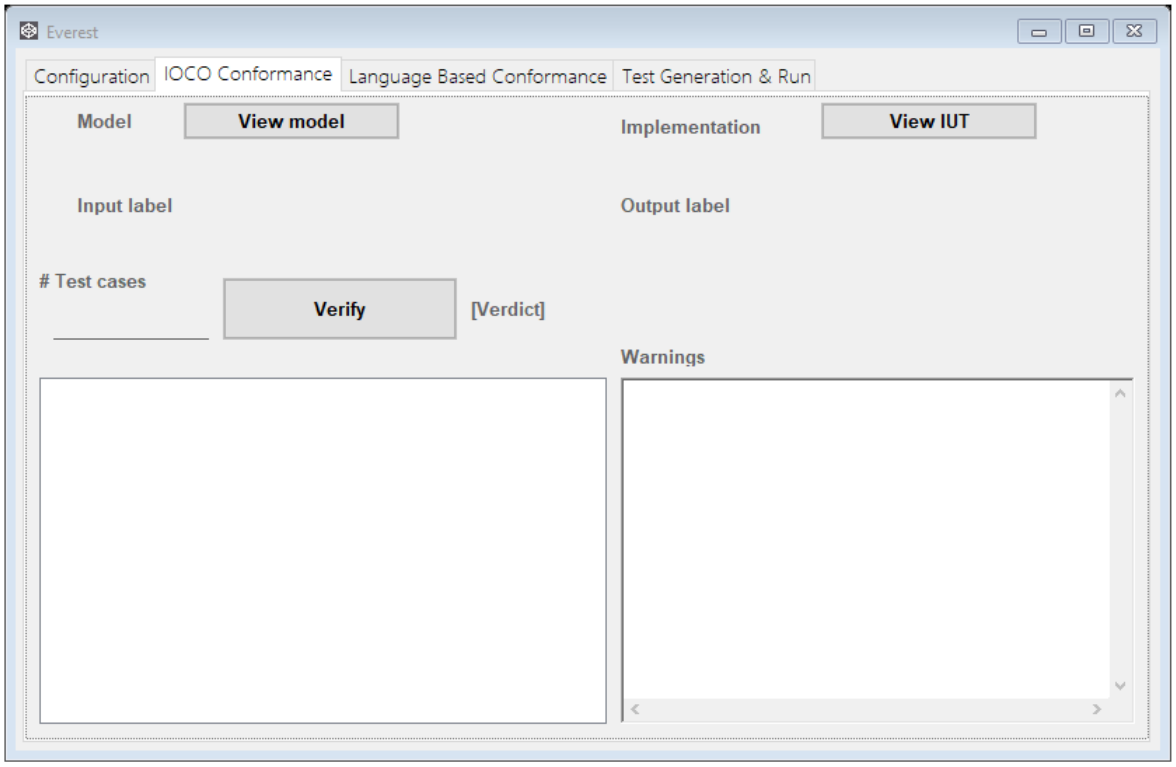


Figura 26 – Interface: verificação **ioco**.

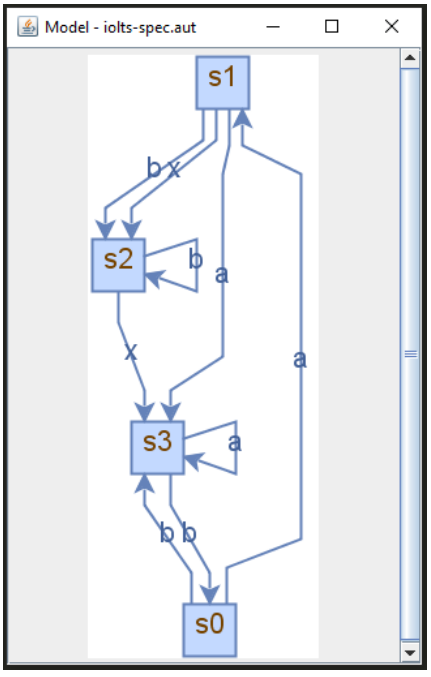


Figura 27 – Interface: visualização do modelo.

A Figura 28 exibe a interface para a verificação de conformidade baseada em linguagem. As expressões regulares devem ser informadas nos campos *Desirable behavior* e *Undesirable behavior*, e especificam respectivamente os comportamentos desejáveis e indesejáveis. Quando nenhuma expressão regular é fornecida, assume-se o fecho Kleene sobre o alfabeto, deste modo as falhas são identificadas quando os modelos não são isomorfos. Assim como na verificação de

conformidade **ioco**, é possível informar o número de casos de teste a serem gerados em caso de não conformidade (campo # *Test cases*). Após clicar no botão $Verify$ o veredito é exibido de modo similar a verificação de conformidade **ioco**.

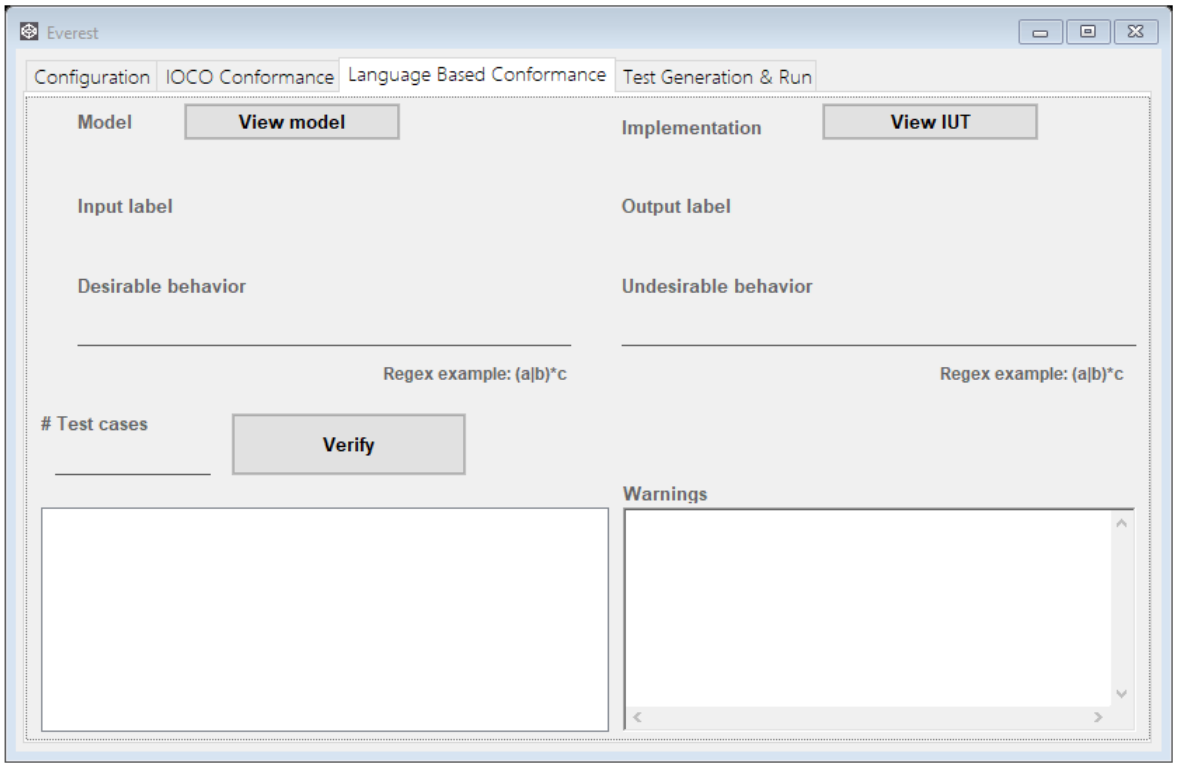


Figura 28 – Interface: verificação baseada em linguagem.

A Figura 29 exibe a interface de geração do multigrafo, dos TPs e de execução dos testes. Na geração do multigrafo o *tester* deve informar a especificação na tela de configuração e o limite máximo de estados que uma IUT caixa-preta pode ter para que as falhas estejam cobertas (campo # *max IUT states*). Já na geração dos TPs pode-se informar o multigrafo (campo *Multigraph*), ou inserir os mesmos parâmetros fornecidos para a construção de multigrafos, além do número de propósitos de teste desejado (campo # *Test purposes*). Os TPs gerados são exibidos ao lado do campo *Warning* e salvos em formato Aldebaran. Por fim, para a execução dos TPs basta informar a IUT na interface de configuração. No campo *Test purpose folder* é possível indicar o diretório contendo os propósitos de teste a serem executados. Na interface de geração e execução dos testes são mantidas as informações preenchidas na tela de configuração. O campo *Warning* informa o usuário caso alguma informação necessária esteja preenchida de forma incorreta, além de exibir os parâmetros necessários para gerar o multigrafo, os TPs e executar os testes.

A Tabela 3 resume as possibilidades de execução da ferramenta conforme as funcionalidades desejadas pelo *tester*. Os botões da interface de geração e execução dos testes ficam, em principio, ocultos para facilitar a interação com o usuário. A medida que os campos são preenchidos tais botões e funcionalidades ficam habilitados, guiando assim o usuário de forma mais intuitiva.

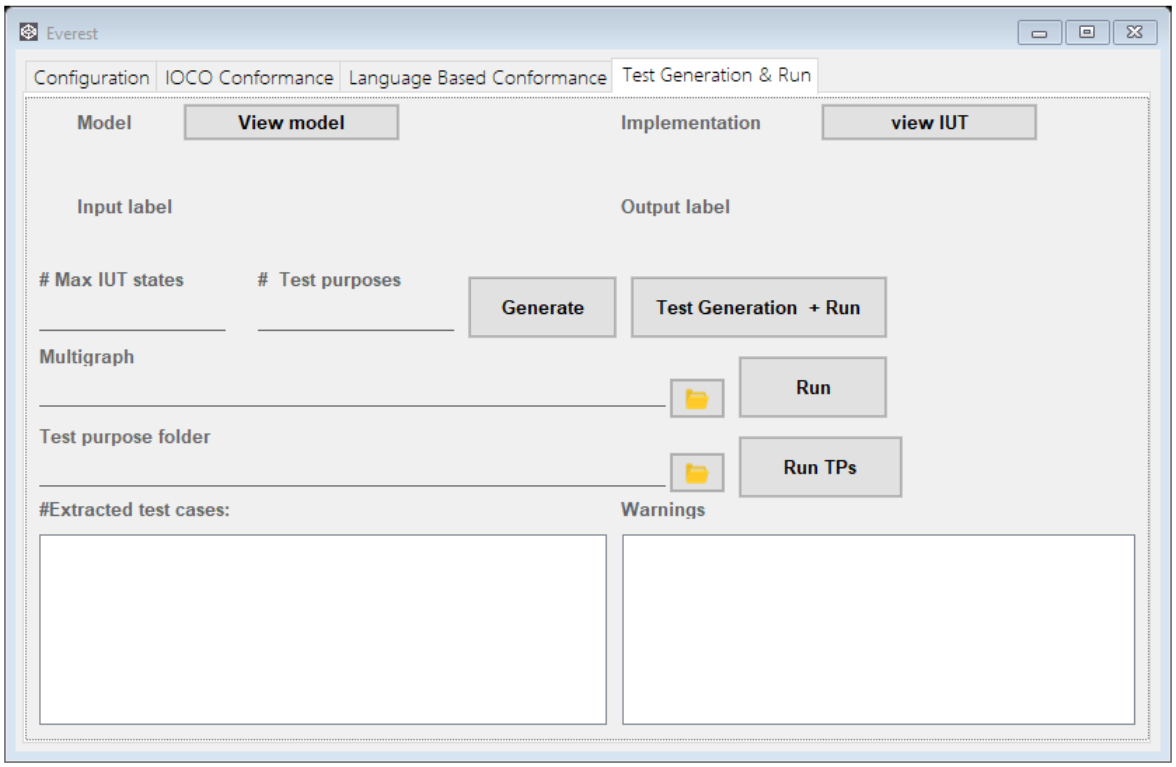


Figura 29 – Interface: geração de TPs e execução de teste.

Entradas	Saídas	Botão
Model + # Max IUT states	Multigrafo	Generate
Model + # Max IUT states + # Test purposes	Multigrafo + TPs	Generate
Multigraph + # Test purposes	TPs	Generate
Model + # Max IUT states + # Test purposes + Implementation	Multigrafo + TPs + Veredito	Test Generation + Run
Multigraph + # Test purposes + Implementation	TPs + Veredito	Run
Test purpose folder + Implementation	Veredito	Run TPs

Tabela 3 – Parâmetros para geração e execução dos testes.

5.3.3 Cenários Práticos de Verificação de Conformidade

A seguir um exemplo de verificação de conformidade **ioco** e outro de conformidade baseada em linguagem são apresentados para demonstrar a aplicação da ferramenta. Os arquivos, em formato Aldebaran, dos modelos de especificação e implementação utilizados estão descritos na Figura 30. Note que os modelos são do tipo IOLTS com os rótulos de entrada e saída identificados pelos marcadores '?' e '!'.

A Figura 31 exibe os campos da interface de configuração preenchidos para a verificação de conformidade utilizando os modelos da Figura 30. No item *Model type* os modelos são definidos como IOLTS e no item *Label* é selecionado a opção $?in, !out$.

A interface de verificação de conformidade **ioco**, para os modelos selecionados, é ilustrada na Figura 32. O veredito da ferramenta mostra que a implementação é **ioco** conforme a

especificação dada.

```
des (s0,9,4)
(s0,?a,s1)
(s0,?b,s3)
(s1,?b,s2)
(s1,!x,s2)
(s1,?a,s3)
(s2,?b,s2)
(s2,!x,s3)
(s3,?b,s0)
(s3,?a,s3)
```

(a) Arquivo iolts-spec.aut contendo o modelo da especificação.

```
des (q0,9,4)
(q0,?a,q1)
(q0,?b,q3)
(q1,?b,q2)
(q1,!x,q2)
(q1,?a,q3)
(q2,?b,q2)
(q2,?a,q3)
(q3,?b,q0)
(q3,?a,q3)
```

(b) Arquivo iolts-impl-q.aut contendo o modelo da implementação.

Figura 30 – Implementação e especificação modelada em arquivo Aldebaran.

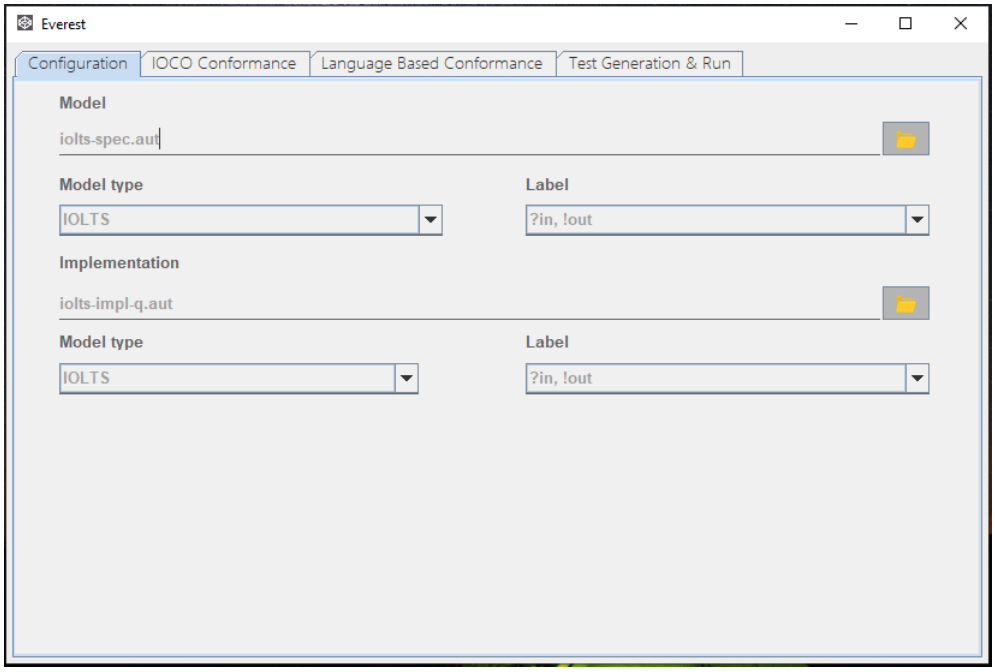


Figura 31 – Interface: exemplo de configuração.

A Figura 33 apresenta os modelos aplicados tanto na verificação **ioco** quanto na baseada em linguagens. A Figura 33a exibe o autômato subjacente ao modelo IOLTS da especificação (Figura 30a) e a Figura 33b corresponde ao autômato subjacente a implementação IOLTS (Figura 30b), ambos obtidos pelo Algoritmo 9. Já a Figura 33c exibe o autômato complemento da especificação que é gerado conforme o Algoritmo 13.

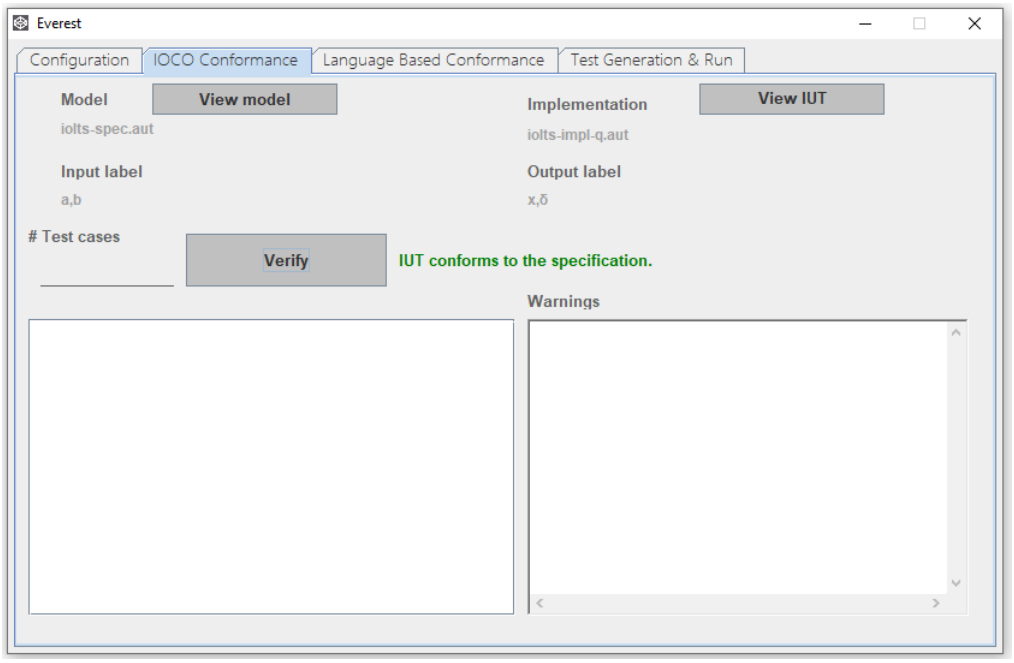
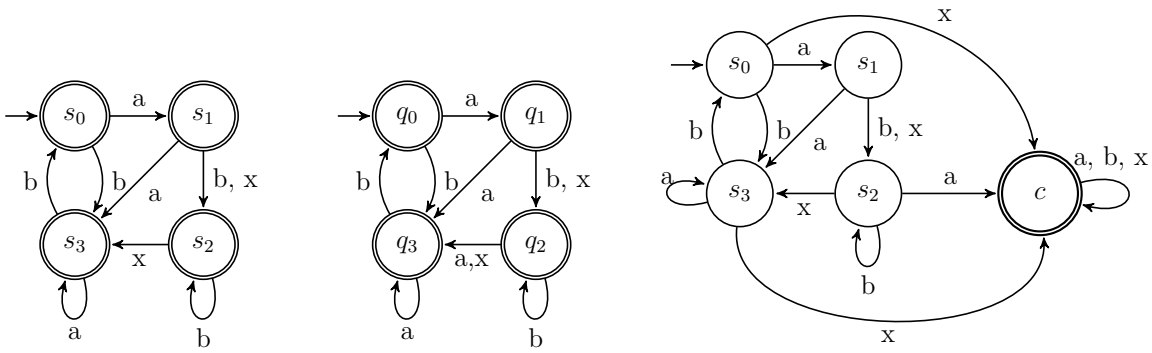


Figura 32 – Interface: exemplo de verificação de conformidade **ioco**.



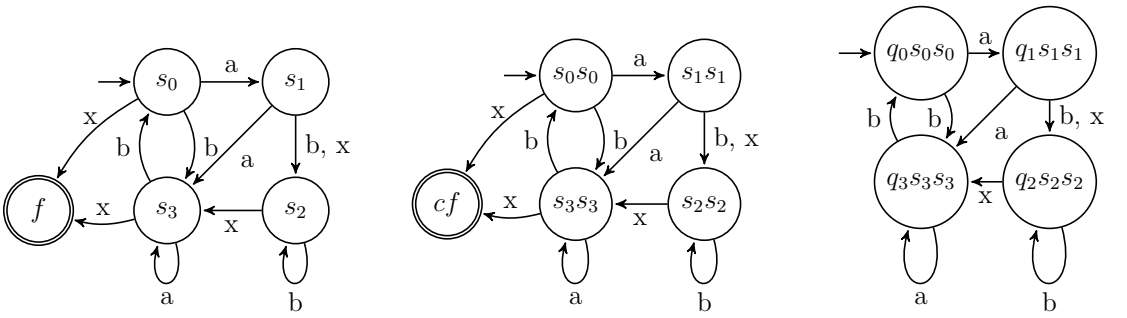
(a) Especificação $\mathcal{S}$ (b) Implementação $\mathcal{Q}$ (c) Complemento da especificação $\overline{\mathcal{S}}$

Figura 33 – Autômatos: especificação $\mathcal{S}$, implementação $\mathcal{Q}$ e complemento da especificação $\overline{\mathcal{S}}$.

A verificação da conformidade **ioco** (Algoritmo 1) primeiramente obtém os autômatos subjacentes a partir dos modelos IOLTS. O autômato D (Figura 34a) é construído de acordo com o Algoritmo 3, para então se obter o modelo de falhas da Figura 34b através da interseção dos autômatos D e o complemento da especificação (Algoritmo 2).

O autômato que representa o conjunto de teste (Figura 34c) é obtido através da interseção entre o modelo de falhas e a implementação dada. Como o autômato do conjunto de teste não possui nenhum estado final, o veredito entre os modelos é de conformidade **ioco**.

A interface de verificação de conformidade baseada em linguagem é ilustrada na Figura 35. No campo *Desirable behavior*, é definida a expressão regular $(a|b)^{\star}ax$ que representa os comportamentos desejáveis, e nenhum comportamento indesejável é informado no campo *Undesirable behavior*. Observe que, embora sejam aplicados os mesmos modelos de implemen-



(a) Autômato $D = otr(S)L_u$ (b) Autômato modelo de falhas (c) Autômato conjunto de teste

Figura 34 – Autômatos: verificação de conformidade **ioco**

tação e especificação da verificação **ioco**, na verificação de conformidade baseada em linguagem o veredito da ferramenta é de não conformidade. Os casos de teste que detectam a não conformidade são apresentados abaixo do botão $Verify$, bem como os caminhos percorridos pelos modelos ao processar cada casos de teste.

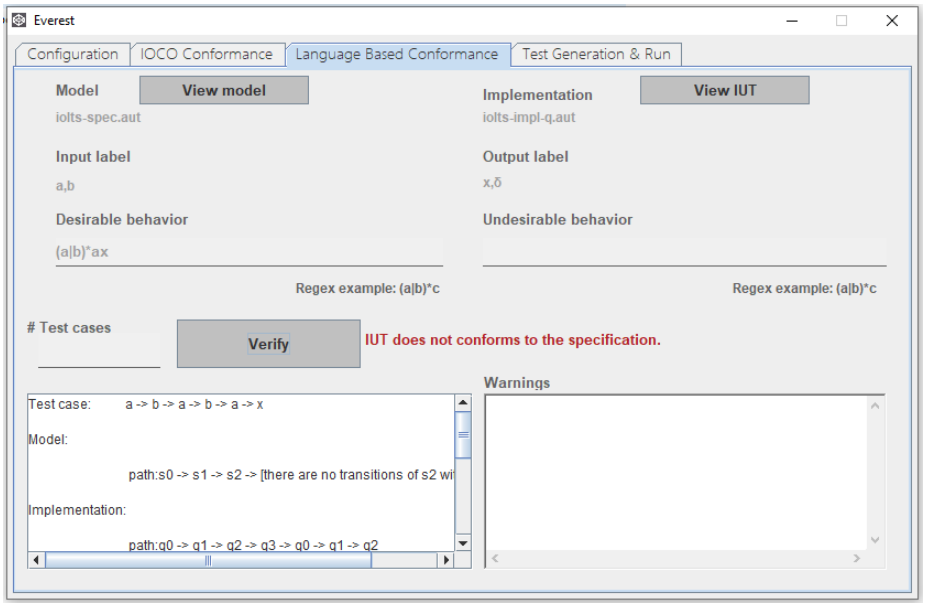
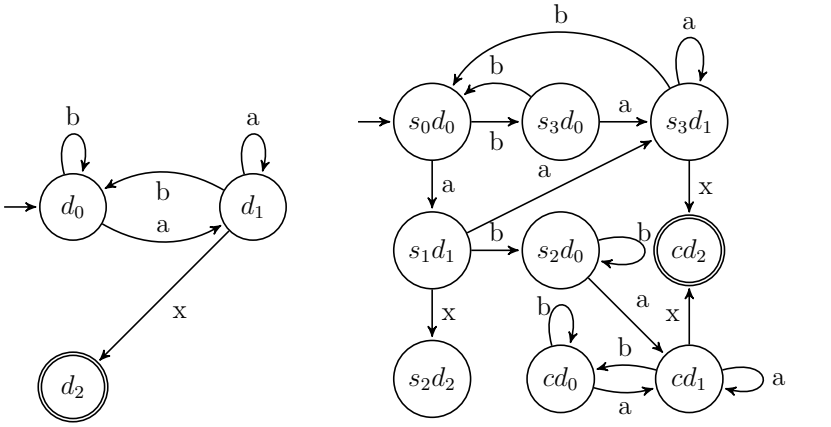


Figura 35 – Interface: exemplo de verificação de conformidade baseada em linguagem.

Na verificação de conformidade baseada em linguagem, assim como na verificação **ioco**, os autômatos subjacentes são obtidos a partir dos modelos IOLTS, conforme a Figura 33. A partir da expressão regular $(a|b)^*ax$ também se obtém o autômato, representado pela Figura 36a, que aceita a respectiva linguagem. Como o modelo de falhas é dado por $[D \cap \overline{otr}(\mathcal{S})] \cup [(F \cap otr(\mathcal{S}))]$ (Algoritmo 5) e nenhum comportamento indesejável F é informado, portanto, $F \cap otr(\mathcal{S}) = \emptyset$, os comportamentos de falha se reduzem a $D \cap \overline{otr}(\mathcal{S})$. O autômato que representa o modelo de falhas é ilustrado na Figura 36b.

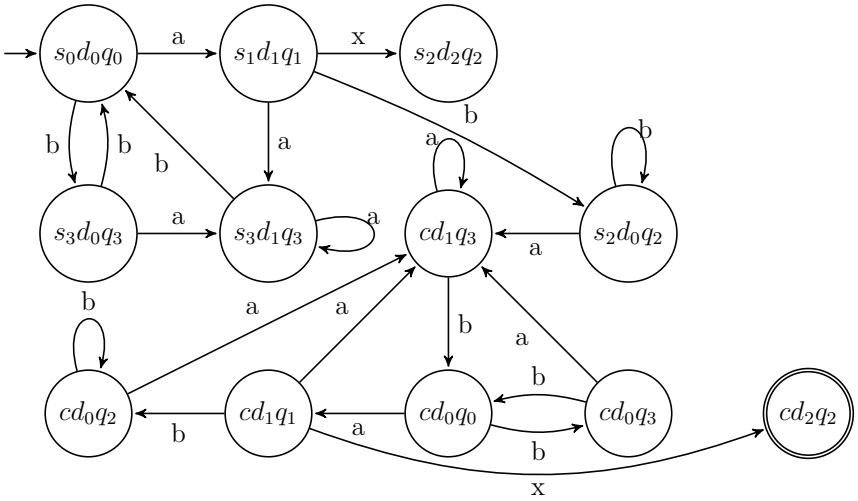
Por fim, o autômato que representa o conjunto de teste é ilustrado pela Figura 36c e obtido pelo Algoritmo 4. Note que este autômato contém um estado final, portanto, as palavras

aceitas pelo autômato fazem parte do conjunto de teste que revelam as falhas e, consequentemente, a não conformidade entre os modelos. Os casos de testes gerados pela ferramenta EVEREST são $\{ababax, abaabax\}$.



(a) Autômato D.

(b) Autômato modelo de falhas $D \cap \overline{otr}(\mathcal{S})$.



(c) Autômato conjunto de teste.

Figura 36 – Autômatos: verificação de conformidade baseada em linguagem.

5.3.4 Cenários Práticos de Geração e Execução de Teste

A fim de demonstrar a aplicação da ferramenta no cenário de geração e execução dos testes, foram utilizados os modelos da especificação $\mathcal{S}$ (Figura 30a) e da IUT $\mathcal{R}$ (Figura 37).

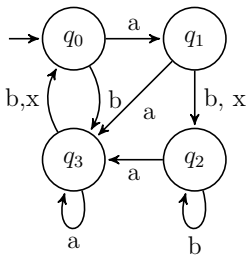


Figura 37 – Implementação IOLTS $\mathcal{R}$.

A Figura 38 exibe a interface de geração de TPs dado a especificação $\mathcal{S}$ da Figura 24b.

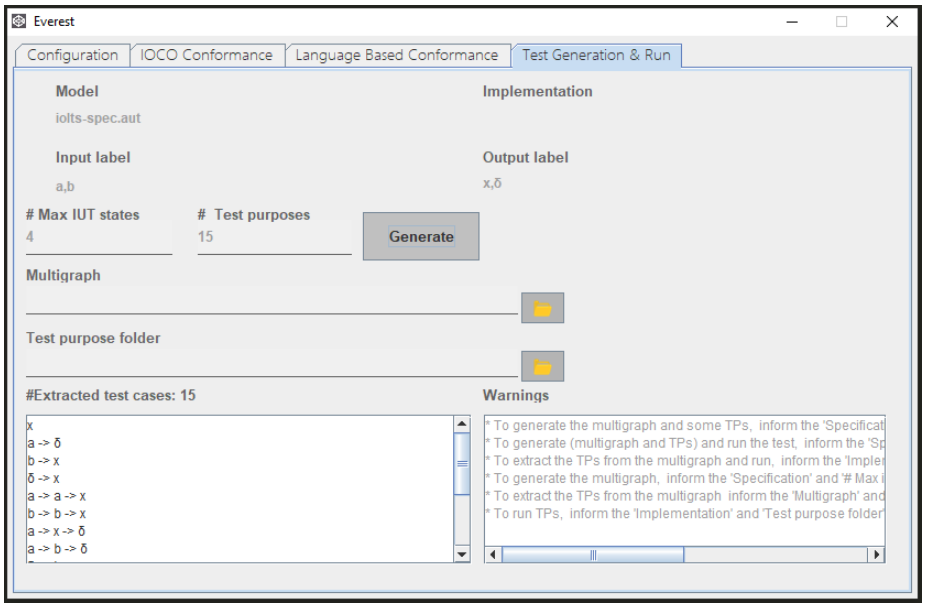


Figura 38 – Interface: geração de TPs.

Neste cenário considerou-se a cobertura de IUTs com até quatro estados ($m = 4$), o mesmo número de estados da especificação, e por isso a quantidade de níveis do multigrafo é $mn + 1 = 17$. O multigrafo, parcialmente representado na Figura 39, é construído com base no Algoritmo 6.

Um algoritmo simples de busca em largura aplicado ao multigrafo é capaz de extrair caminhos a partir do estado inicial $s_{0,0}$ até o estado *fail*. A sequência $\alpha_1 = aabbx$, por exemplo, é extraída através do caminho $s_{0,0} \to s_{1,0} \to s_{3,0} \to s_{0,1} \to s_{3,1} \to fail$ do multigrafo. Em seguida um TP determinístico, acíclico, *input-enabled* e *output-deterministic* pode ser construído pelo Algoritmo 7, como ilustrado na Figura 40a.

A propriedade *input-enabled* é obtida pela adição de um novo estado denominado *pass*, onde para todo estado do TP que não tem alguma entrada definida, uma nova transição desse estado deve ser adicionada para o estado *pass* com o rótulo de entrada. Além disso, são adicionadas transições de *self-loops* rotuladas por todas as ações de entrada no estado *pass* e *fail*. Já a propriedade *output-determinism* é obtida com a introdução de uma nova transição para o estado *pass*, com qualquer rótulo de saída, a partir dos estado do TP que não tenha nenhum rótulo de saída definido.

Agora considere a sequência $\alpha_2 = aaax$. A partir do caminho induzido por α_2, um novo TP, contendo todas as propriedades necessárias, é construído conforme a Figura 40b.

Foram construídos 15 TPs com base nos caminhos induzidos pelas sequências $\{\alpha_1, \alpha_2, x, a\delta, bx, \delta x, aax, bbx, ax\delta, ab\delta, \delta bx, b\delta x, aabx, bbbx, aa\delta x\}$ para a localização de uma falha na IUT $\mathcal{R}$. O conjunto de teste $T = \{\alpha_1, \alpha_2, b, \delta, b\delta, bx, ab, a\delta, aa\delta, aaa, aab, aab\delta, aaba, aaaa, aaab, aaax\delta, aaaxx, aabba, aabbb, aabbx\delta, aabbxx\}$ é obtido através dos TPs da Figura 40.

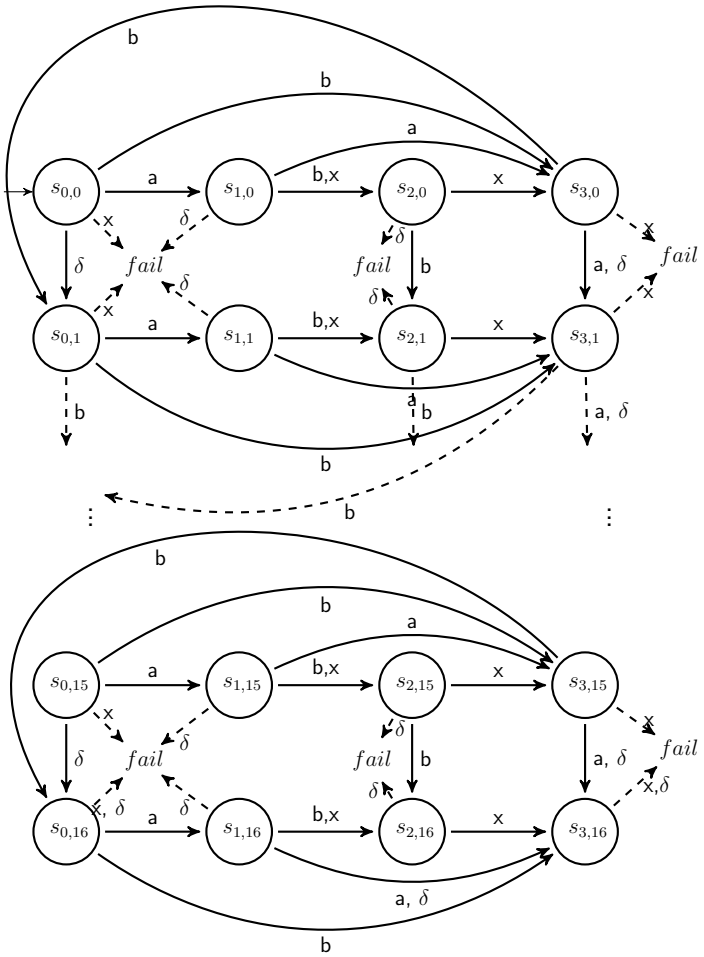
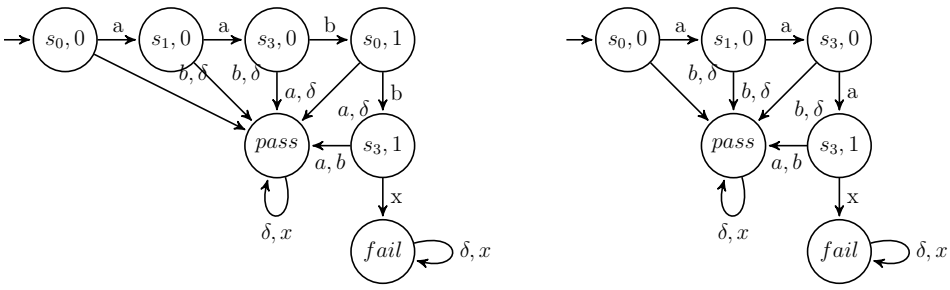


Figura 39 – Multigrafo acíclico D da especificação da Figura 24b.

Uma falha é encontrada ao aplicar o conjunto de teste T na IUT $\mathcal{R}$, mais precisamente, quando os casos de teste $\{\alpha_1, \alpha_2, bx, aaax\delta\}$ são aplicados em $\mathcal{R}$. Note que todos os casos de teste que levam a uma falha partem do estado q_0 e alcança o estado q_0 na IUT $\mathcal{R}$, α_1 por exemplo, induz o caminho $q_0 \to q_1 \to q_3 \to q_0 \to q_3 \to q_0$ na IUT. Logo, $\mathcal{R}$ ~~ioco~~ $\mathcal{S}$.



(a) TP de $aabbx$ (b) TP de $aaax$

Figura 40 – TPs extraídos do multigrafo da Figura 39

5.4 Considerações do capítulo

Neste capítulo foram apresentados os aspectos relacionados ao desenvolvimento da EVEREST, a arquitetura e o projeto UML da ferramenta, que dá suporte ao desenvolvimento de novos módulos, e sua aplicação prática através de uma interface gráfica. A aplicação prática permitiu a constatação que o método baseado em linguagem, implementado na EVEREST, detecta falhas que o método **ioco**, implementado em outras ferramentas, não captura.

A maior contribuição deste trabalho está relacionada a disponibilização de uma ferramenta *open source*, contendo todo seu projeto, e principalmente pela implementação de um método mais geral que o **ioco** clássico. No próximo capitulo é apresentado a análise comparativa entre a EVEREST e as ferramentas da literatura.

6 ANÁLISE COMPARATIVA

Este capítulo apresenta uma análise comparativa entre as ferramentas da literatura e a EVEREST. Estudos de caso também são descritos para evidenciar as diferenças na detecção de falhas de conformidade entre os métodos implementados pelas ferramentas, bem como um cenário real é aplicado em ambas as ferramentas para comparar suas funcionalidades na prática. Em seguida, experimentos práticos relacionados ao desempenho da verificação de conformidade no JTorx e EVEREST são realizados, bem como experimentos para avaliar a geração e execução de testes usando a EVEREST.

6.1 Análise das ferramentas

As ferramentas da literatura e a EVEREST possuem tantos aspectos similares como outras características bastante divergentes. As Tabelas 4 e 5 resumem as principais características e restrições dos métodos e suas respectivas implementações em cada ferramenta.

Tabela 4 – Análise comparativa: métodos e características.

	TGV	Torx	JTorx	**Everest**
Verificação de conformidade				
ioco	-	X	X	**X**
Baseada em linguagem	-	-	-	**X**
Geração				
Geração de conjunto de teste	X	X	X	**X**
Estratégia de teste				
Teste online	X	X	X	**X**
Teste offline	X	X	X	**X**
Propósito de teste	X	X	X	**X**
Abordagem aleatória	-	X	X	**X**
Implementação				
Linguagem	C	C	Java	**Java**
Open source	X	X	X	**X**

A ferramenta EVEREST, assim como o JTorx, foi desenvolvida em Java a fim de simplificar o processo de instalação e manter sua portabilidade. Já as ferramentas TGV e Torx foram desenvolvidas em C. Apesar de todas as ferramentas serem *open source*, algumas classes do código fonte do JTorx estavam ausentes impedindo sua compilação e execução. O mesmo ocorreu com o código do TGV com bibliotecas ausentes o que impossibilitou a sua compilação e execução.

As ferramentas Torx e TGV implementam a noção de conformidade clássica **ioco** [91], enquanto que as ferramentas JTorx e EVEREST implementam a teoria mais recente de **ioco**, chamada de uioco. Além disso, a EVEREST também lida com a relação de conformidade mais

geral, usando linguagens regulares, para especificar propriedades e modelos de falhas. Essas linguagens podem expressar comportamentos desejáveis e/ou indesejáveis à implementação, diferentemente das ferramentas (J)Torx e TGV, que se baseiam apenas na conformidade clássica **ioco** ou suas variações.

Todas as ferramentas suportam os testes de caixa-banca e caixa-preta. O conjunto de teste gerado pelo método baseado em linguagem [27], no cenário de teste de caixa-branca, é completo em relação ao modelo de falha **ioco** e ao modelo de falha baseado em linguagem. Já na geração de testes para implementações caixa-preta, os conjuntos são m-**ioco** completos para IUTs com até m estados. Na (J)TorX a geração é exaustiva em relação à conformidade **ioco** e/ou propósito de teste, e a geração na TGV é *sound* para a relação **ioco** pois nenhuma implementação correta resulta em falha.

A abordagem aleatória, na (J)Torx, é aplicada para definir quais transições da especificação serão induzidas para a geração de conjuntos de teste. Na EVEREST a abordagem aleatória é aplicada somente na verificação de conformidade baseada em linguagem, quando nenhum comportamento desejável e indesejável é fornecido, o que se reduz a uma verificação de isomorfismo entre os modelos. Além da abordagem aleatória a ferramenta (J)Torx, assim como a TGV, possibilita a geração de casos de teste a partir de TPs, para que propriedades específicas das IUTs sejam testadas. A EVEREST disponibiliza a geração de propósitos de teste, onde o multigrafo é construído para testar qualquer IUT com até certo número de estados.

A abordagem de teste *online* está disponível em todas as ferramentas para a geração e execução dos testes, desta forma os casos de teste são gerados e em seguida aplicados às IUTs. A estratégia *online* se deve ao crescimento exponencial no número de casos de testes a serem gerados e aplicados, tornando inviável a geração completa para a posterior aplicação nas IUTs. As ferramentas EVEREST e (J)Torx implementam a verificação de conformidade a partir da estratégia *offline*.

Tabela 5 – Análise comparativa: propriedades e ferramentas.

	TGV	Torx	JTorx	**Everest**
Hipóteses sobre o modelo				
Lida com modelos subespecificados	X	-	X	**X**
Requer modelos *input-enabled*	X	X	X	**-**
Lida com quiescência	-	-	X	**X**
Vereditos				
Execução de teste	A[1]	A e B[2]	A e B	**A**
Conformidade	-	X	X	**X**
Tipo de teste				
Teste de caixa branca	-	X	X	**X**
Teste de caixa preta	X	X	X	**X**

[1] $\{pass, fail, inconclusive\}$.
[2] $\{pass, fail\} X \{hit, miss\}$.

As restrições e propriedades impostas sobre os modelos na relação de conformidade **ioco**, tais como não permitir modelos subespecificados, não são exigidas na verificação baseada em linguagem. A ferramenta Torx, por exemplo, aceita apenas modelos *input-enabled*. Já a JTorx e a TGV apesar de lidarem com modelos subespecificados, manipulam internamente os modelos a fim de garantir a propriedade *input-enabled*. A ferramenta EVEREST naturalmente lida com modelos subespecificados, na verificação de conformidade e geração de conjuntos de teste, sem a necessidade prévia de adequações sobre os modelos. Nas ferramentas JTorx e TGV, os modelos devem ser completamente explorados e novas transições são adicionadas para que as propriedades necessárias sejam garantidas. Portanto, as operações realizadas sobre os modelos, em geral, contém mais transições a serem processadas, interferindo no desempenho destas ferramentas. Para lidar com quiescência nos modelos, tanto o Jtorx quanto a EVEREST adicionam *self-loops* rotulados com δ nos estados quiescentes da implementação e da especificação.

Os vereditos da verificação de conformidade obtidos pelas ferramentas EVEREST e (J)Torx indicam se os modelos são conformes ou não. Já os vereditos de execução dos testes indicam se um propósito de teste e/ou a relação de conformidade **ioco** foram capazes de encontrar alguma falha nas IUTs.

6.2 Estudos de caso

Alguns cenários de teste foram aplicados a ferramenta EVEREST e ao JTorx. Assuma a especificação $\mathcal{S}$ da Figura 33a e a implementação candidata $\mathcal{Q}$ da Figura 33b. Também assuma como implementação candidata o modelo $\mathcal{R}$ da Figura 37. Os alfabetos de entrada e de saída são, respectivamente, $L_I = \{a, b\}$ e $L_U = \{x\}$. Além disso, todos os modelos são determinísticos, embora ambas as ferramentas lidem com modelos não-determinísticos.

No primeiro cenário é verificada a conformidade entre a implementação $\mathcal{R}$ e a especificação $\mathcal{S}$. A ferramenta EVEREST retornou um veredito de não conformidade para a relação **ioco**, bem como os casos de teste $\{b, aa, ba, aaa, ab, ax, abb, axb\}$. O subconjunto de casos de teste $\{b, aa, ba, aaa\}$ induz caminhos de s_0 para s_3 em $\mathcal{S}$ e de q_0 para q_3 em $\mathcal{R}$, onde a saída x é produzida por $\mathcal{R}$, mas em $\mathcal{S}$ é produzida a saída δ, já que s_3 em $\mathcal{S}$ é um estado quiescente. O subconjunto de casos de teste $\{ab, ax, abb, axb\}$ induzem caminhos para o estado s_2 em $\mathcal{S}$ e q_2 em $\mathcal{R}$. Neste caso a saída δ é produzida pela implementação $\mathcal{R}$ e $\mathcal{S}$ produz x. Os casos de teste evidenciam a não conformidade de acordo com a relação **ioco**.

A aplicação do mesmo cenário ao JTorx também retornou um veredito de não conformidade, como já esperado. Porém, os casos de teste gerados foram $\{b, ax, ab\}$, um subconjunto dos casos de teste gerados pela ferramenta EVEREST. A EVEREST exibe todos os casos de teste e caminhos associados para cada falha, de acordo com a cobertura de transição do modelo da especificação, enquanto que o JTorx retorna apenas um caso de teste por falha.

Já no segundo cenário, descrito na Subseção 5.3.3, a verificação de conformidade baseada em linguagem entre a implementação $\mathcal{Q}$ e a especificação $\mathcal{S}$, usando as expressões $D = (a|b) * ax$

e $F = \emptyset$, foi capaz de encontrar falhas. Já a ferramenta JTorx, através da relação uioco, retornou um veredito de conformidade, mostrando que a EVEREST é capaz de encontrar não conformidades, que não podem ser detectadas pelo JTorx, a partir da conformidade mais geral baseada em linguagem. Os casos de teste gerados pela ferramenta EVEREST na verificação de conformidade baseada em linguagem foram $\{ababax, abaabax\}$ que detectam a presença de comportamentos da implementação que terminam em ax e que não estão definidos na especificação.

6.3 Análise de um cenário real

Um cenário real também foi desenvolvido para a aplicação da verificação de conformidade em modelos IOLTS que representam funcionalidades de um caixa eletrônico (*Automatic Teller Machine* - ATM), adaptado de [94, 75]. Os modelos da ATM, aqui retratado, podem receber como estímulos de entrada o conjunto $L_I = \{ic, acc, pin, tra, sta, wd, amo\}$, ou seja, a inserção do cartão na máquina (ic - *insert card*), a senha (pin), a solicitação para transferência (tra - *transfer*), a conta para transferência (acc - *account*), a emissão do extrato (sta - *statment*), a solicitação de saque (wd - *withdrawal*) e a verificação do saldo (amo - *amout*). As saídas emitidas pelos modelos da ATM são $L_U = \{cpi, bpi, mon, rec, ins, sho\}$, ou seja, senha correta (cpi - *correct pin*), senha incorreta (bpi - *bad pin*), dinheiro (mon - *money*), comprovante da transferência (rec- *receipt*), saldo insuficiente (ins- *insufficient balance*) e exibição do extrato (sho - *show statement*).

A Figura 41 representa a modelagem da funcionalidade de saque (wd) numa especificação $\mathcal{A}$ de uma ATM. O modelo especifica que se o valor (amo) solicitado para saque é maior do que o disponível na conta (ins), o saque não é efetuado e o modelo retorna para o estado s_3 onde a opção de saque pode ser selecionada novamente.

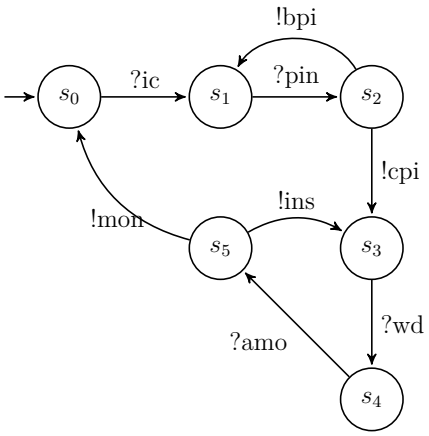


Figura 41 – Especificação da ATM $\mathcal{A}$.

Já a Figura 42 representa uma outra especificação $\mathcal{B}$ de ATM com mais funcionalidades: o saque (wd), a transferência (tra) e a emissão de extrato (sta).

A Figura 43 exibe o modelo de uma IUT $\mathcal{Z}$ de ATM contendo as funcionalidades de saque (wd) e de transferência (tra). Na funcionalidade de saque (wd), caso o valor solicitado

(amo) seja maior do que o disponível na conta é retornado ao estado s_7 para que seja definido um novo valor.

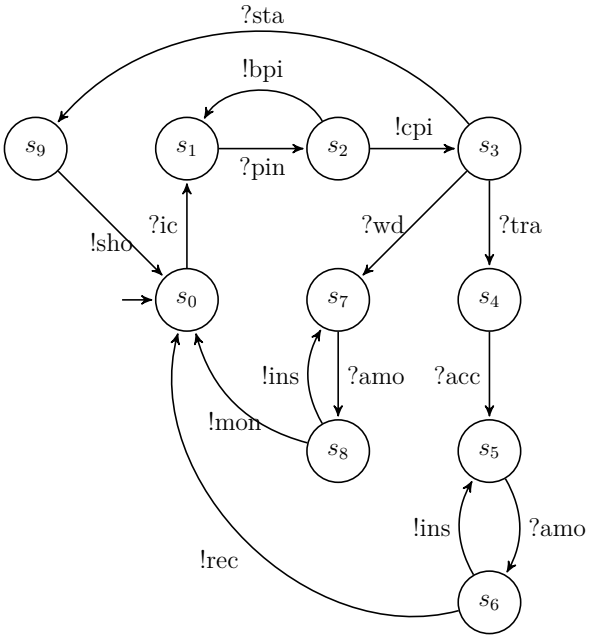


Figura 42 – Especificação da ATM $\mathcal{B}$.

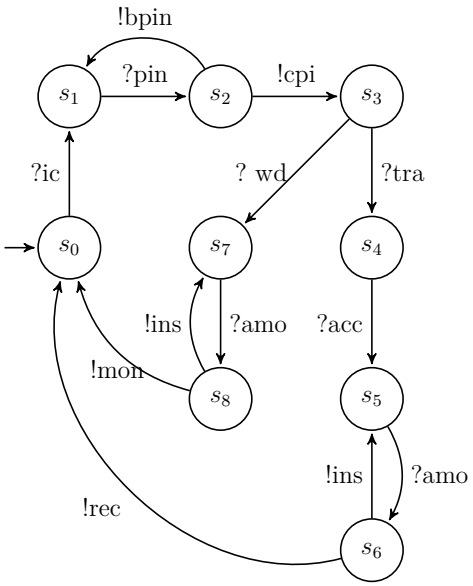


Figura 43 – IUT $\mathcal{Z}$.

Num primeiro cenário o objetivo é verificar se a IUT $\mathcal{Z}$ é **ioco** conforme a Spec $\mathcal{A}$. O veredito obtido é de conformidade **ioco** tanto pela ferramenta JTorx quanto pela Everest. No entanto, a verificação de conformidade baseada em linguagem da Everest retornou uma falha. Através da expressão regular D = ic pin cpi wd amo ins amo, ou seja, uma sequência de ações para um cenário em que o saldo da conta é insuficiente para o valor solicitado no saque, e o usuário deve então informar um novo valor. A verificação de conformidade baseada

em linguagem identifica uma falha, gerando o caso de teste $\{ic \rightarrow pin \rightarrow cpi \rightarrow wd \rightarrow amo \rightarrow ins \rightarrow amo\}$, pois, na especificação o comportamento D não é observado, enquanto que na IUT este comportamento está presente.

No segundo cenário é analisado a confiabilidade do veredito apresentado pelo JTorx, na verificação de conformidade **ioco**, após a ferramenta modificar os modelos subespecificados. Conforme discutido anteriormente, nos estados subespecificados o JTorx adiciona *self-loops* para torná-los *input-enabled*. Como a implementação $\mathcal{Z}$ é subespecificada, a verificação se a IUT $\mathcal{Z}$ é **ioco** conforme a especificação $\mathcal{B}$, pelo JTorx, deve garantir que todos os estados da IUT sejam *input-enabled*, portanto, o comportamento original da implementação é modificado. Uma falha na implementação é então encontrada pelo JTorx, na verificação de conformidade **ioco**, através do caso de teste $\{ic \rightarrow pin \rightarrow cpin \rightarrow sta\}$.

Na ferramenta EVEREST ao aplicar a verificação de conformidade **ioco** neste mesmo cenário nenhuma falha é encontrada, pois na IUT $\mathcal{Z}$ não há especificado o comportamento de falha $\{ic \rightarrow pin \rightarrow cpin \rightarrow sta\}$ identificado pelo JTorx. A falha localizada pela ferramenta JTorx na verificação de conformidade **ioco** representa um falso positivo, pois a falha foi detectada com base no comportamento adicionado pela própria ferramenta. Vale destacar que ao transformar a IUT $\mathcal{Z}$ em *input-enabled* e executar a verificação de conformidade **ioco** na EVEREST novamente, a mesma falha é então detectada, assim como ocorreu no JTorx.

Note que a única entrada habilitada no estado inicial s_0, da IUT $\mathcal{Z}$, é ic (inserir o cartão). Quando o JTorx torna o modelo *input-enabled* todas as entradas se tornam habilitadas, porém é incoerente, por exemplo, que seja permitido informar um valor para o saque (amo) logo de início, pois o usuário pode optar em realizar uma transferência (tra), neste caso a ação de se escolher o valor para saque não deveria estar habilitado em nenhum momento. Observa-se que as modificações nos modelos subespecificados alteram o comportamento original da IUT fornecida, tornando a verificação de conformidade imprecisa sobre o aspecto geral de funcionamento do sistema.

Agora considere a IUT $\mathcal{Y}$ (Figura 44) cuja diferença com a especificação $\mathcal{A}$ (Figura 41) se deve apenas pela transição do estado s_4 para s_5 que passa de $?amo$ para $!mon$. Logo, a IUT permite escolher a função de saque, sendo que o saldo não é verificado (amo) antes do dinheiro (mon) ser disponibilizado. Na especificação o saldo é verificado para que então o dinheiro seja disponibilizado, caso o saldo seja suficiente. De acordo com a geração dos TPs com base na especificação $\mathcal{A}$ e a execução dos testes sobre a IUT $\mathcal{Y}$ o veredito de falha é obtido. Foram extraídos e aplicados 80 TPs, pela ferramenta EVEREST, considerando a cobertura de IUTs com até 6 estados. A falha encontrada na IUT ocorreu através do caminho $ic \rightarrow pin \rightarrow cpin \rightarrow wd$.

O mesmo cenário de teste foi aplicado na ferramenta JTorx. A geração ocorre a partir do estado inicial da especificação sendo realizada uma busca em largura para que as entradas a serem aplicadas na IUT sejam obtidas. Em seguida, as saídas produzidas por ambos os modelos são comparadas. Para este cenário de teste o JTorx encontrou a falha no oitavo passo de teste: $?ic, \delta, ?pin, !bpi, ?pin, !cpi, ?wd, !mon$.

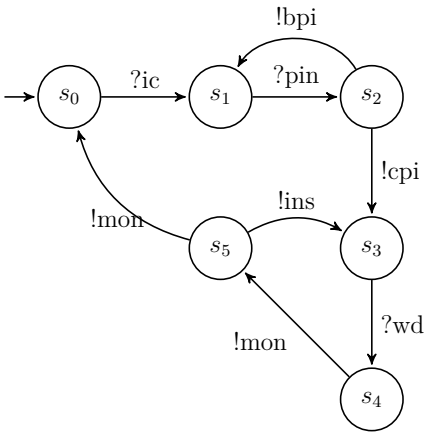


Figura 44 – IUT $\mathcal{Y}$.

6.4 Experimentos práticos

Os experimentos práticos nesta seção exploram o desempenho das ferramentas EVEREST e JTorx com relação a verificação de conformidade, além da análise de desempenho sobre a geração e execução de conjuntos de teste pela EVEREST. Os experimentos foram organizados em questões de pesquisa, que expõem o objetivo de cada experimento de forma clara e específica. Cada teste foi executado uma única vez, devido a quantidade enorme de experimentos distintos realizados e o tempo total consumido. No total foram realizados 35.640 experimentos, entre verificações de conformidade e execuções de TPs nas IUTs, e 740 multigrafos gerados. Os experimentos foram conduzidos numa máquina com processador Intel quad-core i5 de 1.8 GHz, 8 GB de memória RAM e sistema operacional Windows 10.

6.4.1 Verificação de Conformidade

Os principais parâmetros avaliados nos experimentos de verificação de conformidade envolvem especificações e IUTs que resultam tanto em vereditos de conformidade, quanto em vereditos de não conformidade. Além disso, os alfabetos de entrada e saída variam de tamanho, bem como o número de estados dos modelos.

Vale ressaltar que a ferramenta EVEREST não impõe nenhuma restrição aos modelos a serem verificados. No entanto, para efeito de comparação, em todos os experimentos os modelos gerados são *input-enabled* e determinísticos, visto que o JTorx impõe restrições sobre os modelos modificando-os de forma a atender tais propriedades. Além disso, a EVEREST foi adaptada para gerar exatamente o mesmo conjunto de testes gerado pelo JTorx, usando um parâmetro que define o número de casos de teste desejados. Quando nenhum valor é informado a EVEREST gera um conjunto de teste com base na cobertura de transição da especificação.

Os modelos de especificações foram gerados dados o número de estados, bem como o tamanho dos alfabetos de entrada e de saída. As transições foram geradas aleatoriamente, desde que atendidas as propriedades desejadas, para se manter a maior imparcialidade sobre os resultados dos experimentos. Para garantir a construção de IUTs **ioco** conformes, foram geradas

submáquinas das respectivas especificações contendo estados e transições adicionais. As IUTs **ioco** não conformes foram construídas também com base na especificação, porém com uma porcentagem específica de modificação sobre as transições, além da adição de novos estados e transições, no caso de IUTs maiores que a especificação.

As verificações de conformidade foram realizadas com a geração de 10 especificações e 10 IUTs para cada especificação, para um certo número de estados e símbolos definidos em cada modelo de especificação e IUT geradas. Os gráficos apresentados correspondem a média de tempo para executar as verificações. Uma das dificuldades encontradas para se medir o tempo, de execução das verificações de conformidade, na ferramenta JTorx foi que, apesar da ferramenta ter código aberto, os fontes não estavam todos disponíveis, não sendo possível realizar as modificações necessárias para que o tempo de execução fosse registrado. Portanto, para a medição do tempo de execução, das verificações de conformidade, foi desenvolvido um aplicativo de chamada para as ferramentas, passando os parâmetros necessários, e então medindo o tempo total gasto no processo de verificação até a apresentação do veredito. A Seção 6.4.3 apresenta as ameaças à validade dos experimentos realizados.

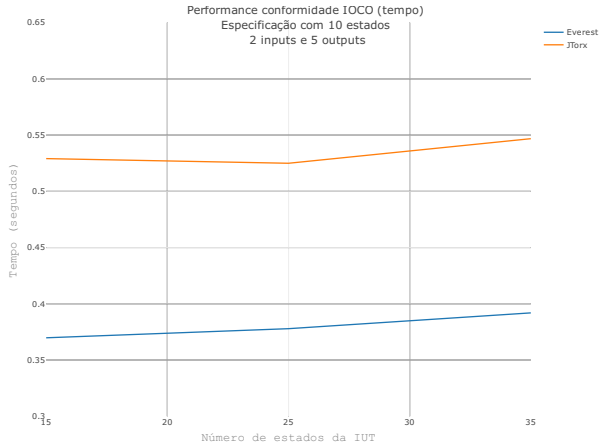
6.4.1.1 Variando o tamanho dos alfabetos

A primeira questão de pesquisa a ser tratada é *"O tamanho dos alfabetos de entrada e saída interfere no tempo da execução das verificações em vereditos de conformidade?"*. Para responder a essa questão foram geradas especificações com 10 estados e IUTs **ioco** conformes com 15, 25 e 30 estados. Os modelos dispunham de um alfabeto de 7 símbolos, com as seguintes variações: 2 entradas e 5 saídas; 3 entradas e 4 saídas; 4 entradas e 3 saídas; 5 entradas e 2 saídas. Foram realizadas um total de 3.600 verificações de conformidade.

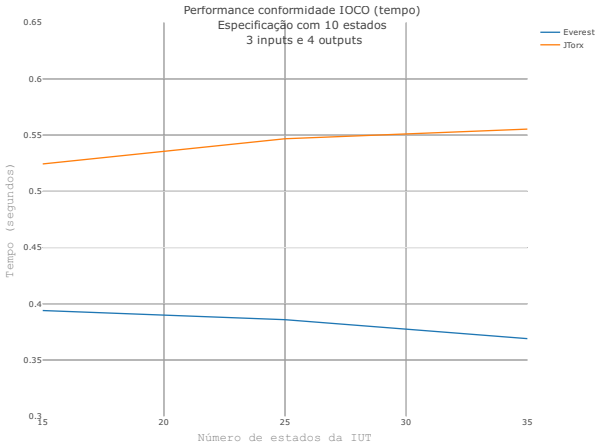
Os resultados apresentados na Figura 45 mostram que o desempenho da ferramenta Everest é melhor em todos os cenários com vereditos de conformidade, em torno de 33% a 66% mais rápido que o JTorx. Essa vantagem se deve provavelmente a forma como funciona o método implementado pela Everest, usando operações sobre linguagens regulares e autômatos. Além disso, os gráficos mostram que, em ambas as ferramentas, a variação no número de entradas e saídas não impacta significativamente no tempo de verificação.

A segunda questão de pesquisa é "Considerando as mesmas variações sobre os alfabetos, da questão anterior, porém com modelos cujo veredito resulta em não conformidade, o tamanho dos alfabetos de entrada e saída interfere no tempo da execução das verificações?". Os resultados observados nos gráficos da Figura 46 mostram que o desempenho da Everest é ainda melhor com relação ao JTorx nos cenários de não conformidade. Neste caso a Everest executou a verificação de conformidade de 42% a 118% mais rápido do que o JTorx. Nota-se também que, devido a geração dos casos de teste nos cenários de não conformidade, o tempo de verificação é maior em ambas as ferramentas, em torno de 25% a 54% na Everest e de 29% a 118% no JTorx, se comparado com os cenários de conformidade. Logo, conclui-se que a variação no tamanho dos alfabetos de entrada e saída tem maior impacto em vereditos de não conformidade,

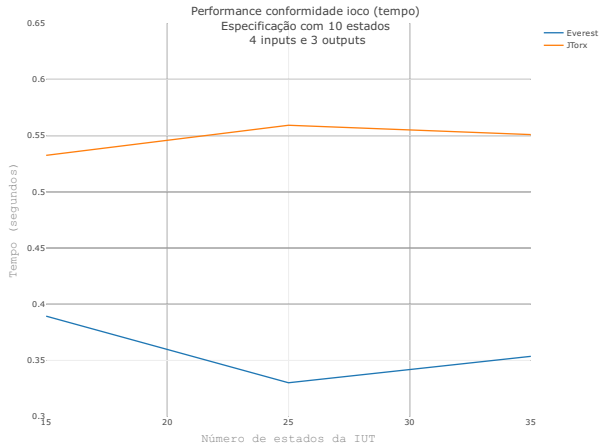
em ambas as ferramentas.



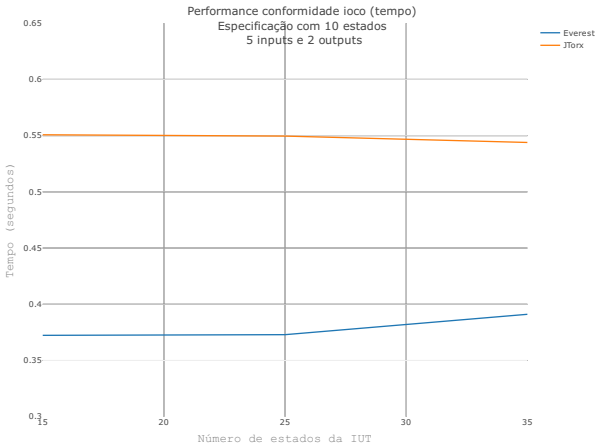
(a) 2 inputs e 5 outputs



(b) 3 inputs e 4 outputs



(c) 4 inputs e 3 outputs



(d) 5 inputs e 2 outputs

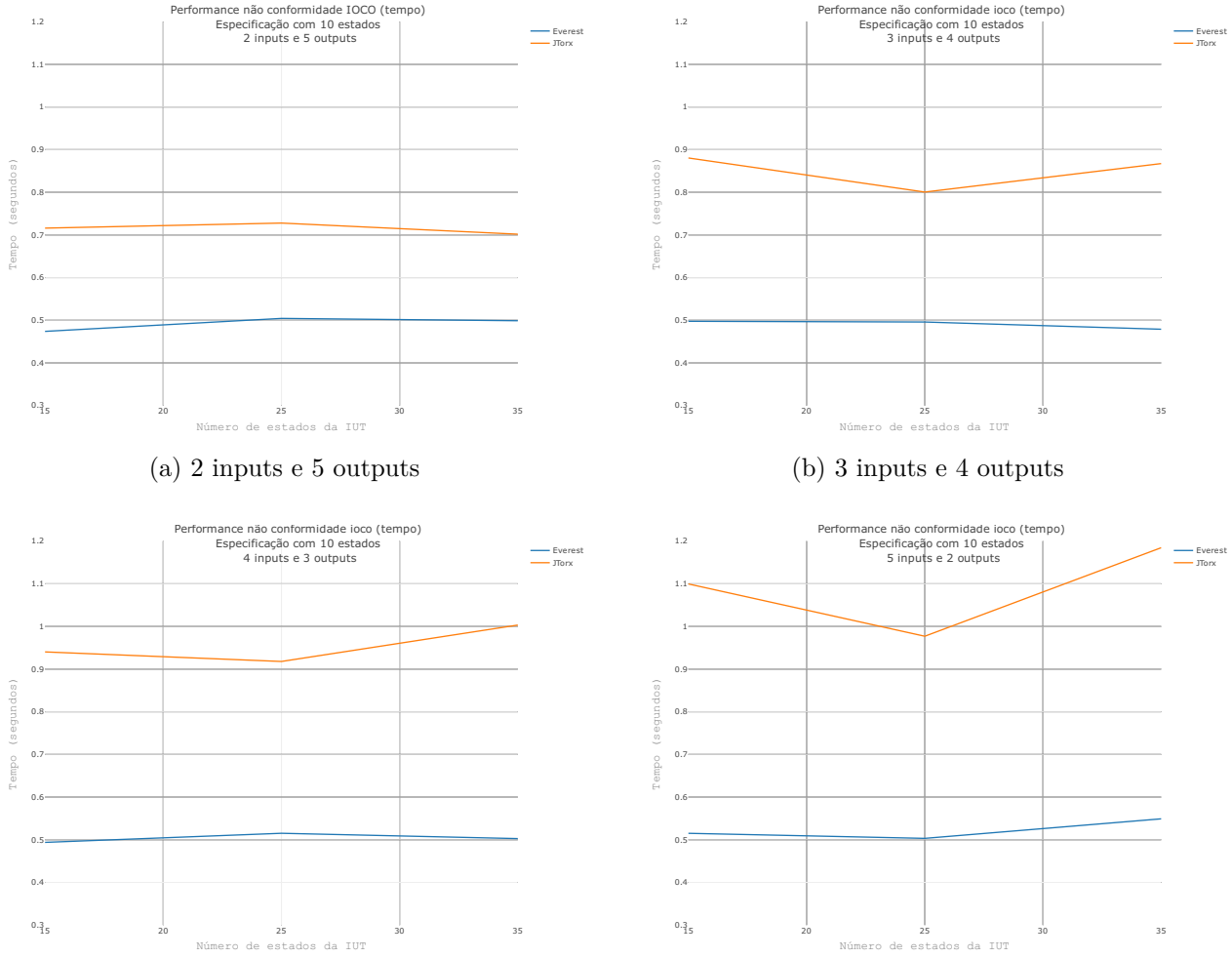
Figura 45 – Experimento: conformidade **ioco** com variação I/O.

Agora a questão de pesquisa é *"Qual o impacto no tempo de verificação sobre modelos com alfabetos, de entradas ou de saídas, maiores?"*. Neste experimento os alfabetos foram definidos com tamanho 12, sendo fixado em 2 entradas e 10 saídas, e também com 10 entradas e 2 saídas. A verificação foi estabelecida tanto para vereditos de conformidade quanto para vereditos de não conformidade, em especificações com 10 estados e IUTs com 15, 25 e 35 estados.

Nota-se nos gráficos da Figura 47 que o impacto da variação das entradas e saídas, em vereditos de conformidade, é mínimo com relação ao tempo no processo de verificação. Também pode se notar que a verificação de conformidade na ferramenta EVEREST é $2,56\%$ mais rápida para modelos com 2 entradas e 10 saídas do que em modelos cujo alfabeto continham 10 entradas e 2 saídas. Já no JTorx o tempo de verificação é em torno de $3,51\%$ mais rápido para modelos com 10 entradas e 2 saídas do que em modelos com 2 entradas e 10 saída.

Nos experimentos com vereditos de não conformidade a variação do número de entradas

e saídas, para o alfabeto de tamanho 12, teve maior influência no tempo de verificação. Em ambas as ferramentas o menor tempo de verificação ocorre em modelos com menor número de entradas (2 entradas e 10 saídas). O tempo de verificação na EVEREST é de 12, 73% a 42, 86% mais rápido para modelos com 2 entradas e 10 saídas em relação aos modelos com 10 entradas e 2 saídas, enquanto que na ferramenta JTorx essa diferença de tempo é de 200% a 352%.



(a) 2 inputs e 5 outputs

(b) 3 inputs e 4 outputs

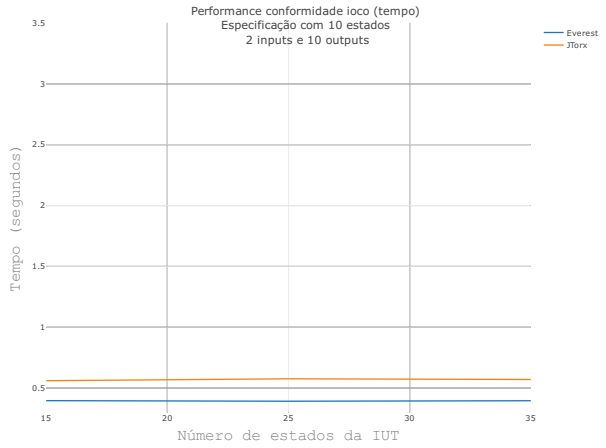
(c) 4 inputs e 3 outputs

(d) 5 inputs e 2 outputs

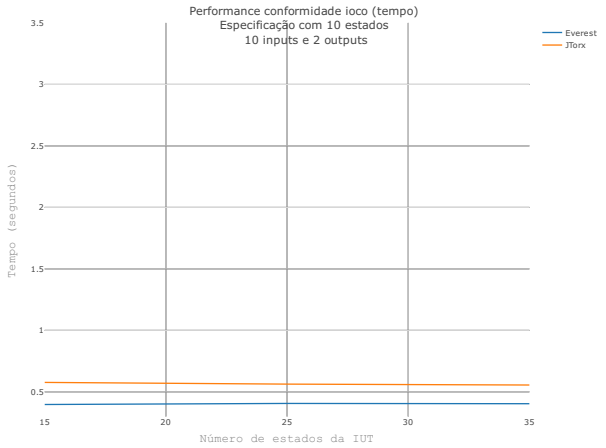
Figura 46 – Experimento: não conformidade **ioco** com variação I/O.

6.4.1.2 Variando o número de estados

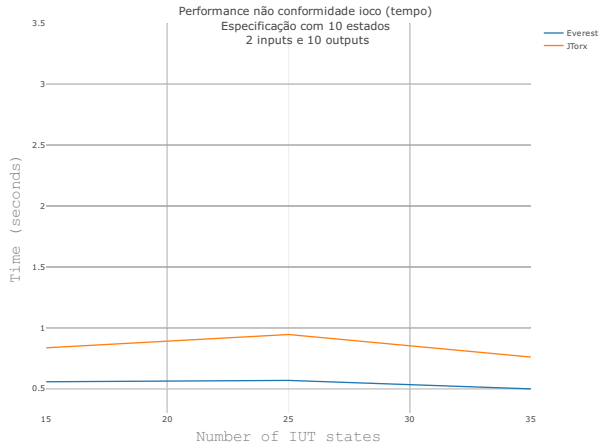
Nesta subseção a questão de pesquisa abordada é *"Qual o impacto do número de estados dos modelos, de especificações e de IUTs, no tempo de execução das verificações com vereditos de conformidade?"*. As verificações para vereditos de conformidade contam com especificações de 10, 20 e 30 estados e IUTs variando de 20 a 50, 30 a 50, e 40 a 50 estados, respectivamente, aumentando de 10 estados cada conjunto. Os alfabetos aqui foram fixados em 10 símbolos, sendo 5 entradas e 5 saídas. Foram realizadas 900 verificações de conformidade no total.



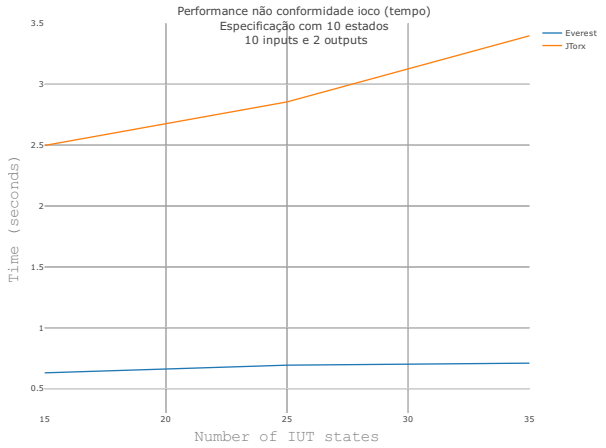
(a) 2 inputs e 10 outputs - conformidade **ioco**



(b) 10 inputs e 2 outputs - conformidade **ioco**



(c) 2 inputs e 10 outputs - não conformidade **ioco**



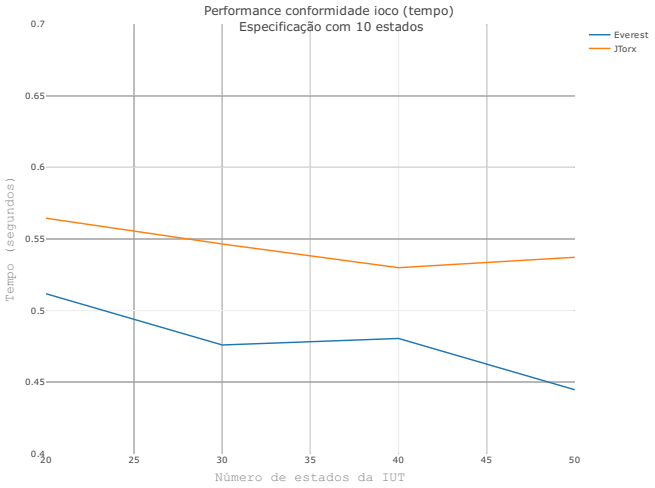
(d) 10 inputs e 2 outputs - não conformidade **ioco**

Figura 47 – Experimento: conformidade e não conformidade **ioco** com variação I/O.

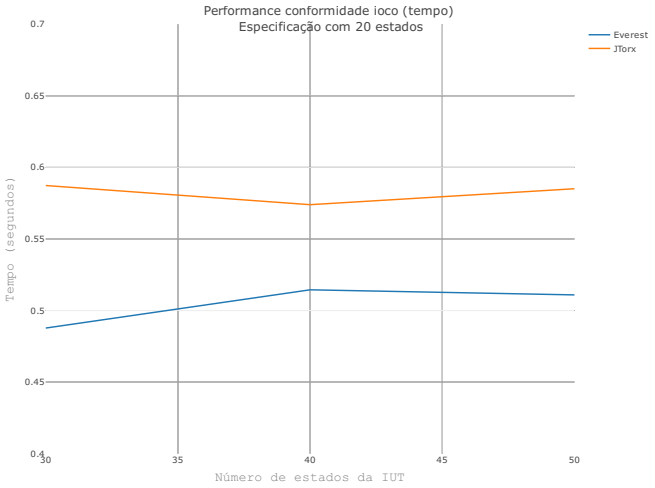
Os gráficos da Figura 48 exibem o desempenho da verificação sobre modelos que resultam em vereditos de conformidade. Observa-se que, em geral, a ferramenta JTorx toma mais tempo de verificação que a EVEREST. Além disso, a medida que o tamanho da especificação aumenta, o desempenho da EVEREST melhora em comparação ao desempenho do JTorx. Nos experimentos com especificações contendo 10 estados a EVEREST executou de $8,33\%$ à $20,45\%$ mais rápido que o JTorx. Note que para IUTs com mais de 40 estados o tempo da execução diminui na EVEREST e aumenta na JTorx, devido a estrategia de construção de autômatos implementada na EVEREST para realizar a verificação de conformidade. Já em especificações com 20 estados o ganho é de $11,76\%$ à $20,83\%$, enquanto que para especificações com 30 estados a EVEREST leva vantagem de $20,37\%$ a 25% sobre o JTorx.

A próxima questão de pesquisa é *"Qual o impacto do número de estados dos modelos de especificações e de IUTs, no tempo de execução das verificações com vereditos de não conformidade?"*. Novamente com especificações de 10, 20 e 30 estados e IUTs contendo, respectivamente, 1%, 2% e 4% de modificação em relação as transições da especificação. Vale ressaltar que, com o intuito de manter a imparcialidade nos resultados dos experimentos, as alterações realizadas

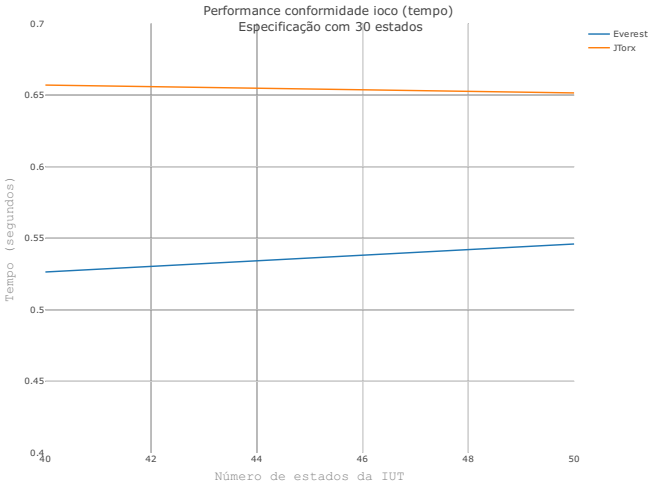
nas transições ocorrem de maneira aleatória com base nas transições da especificação. Já os alfabetos dos modelos foram fixados com tamanho 10, sendo 5 entradas e 5 saídas. Neste caso foram realizadas 3600 verificações de não conformidade.



(a) Especificação com 10 estados



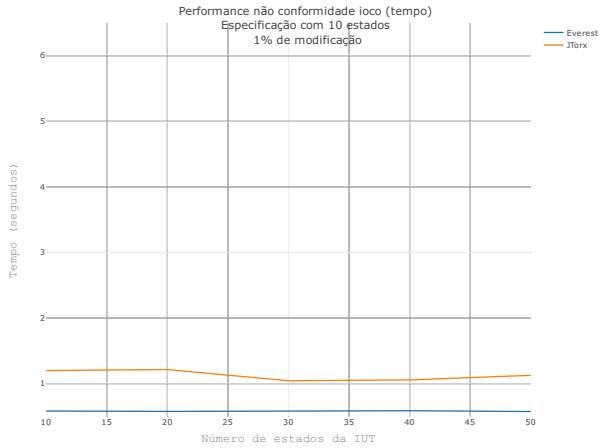
(b) Especificação com 20 estados



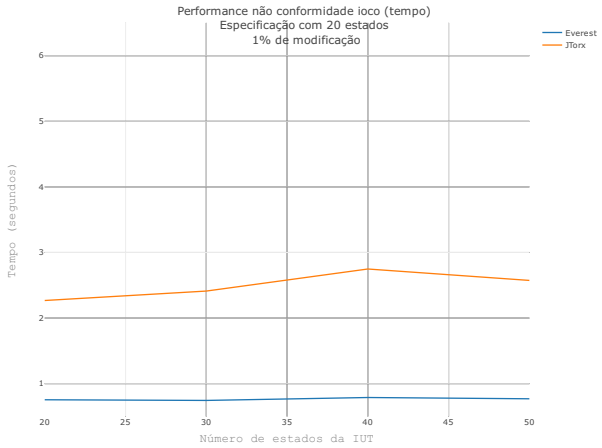
(c) Especificação com 30 estados

Figura 48 – Experimento: conformidade **ioco** - especificações com 10, 20 e 30 estados.

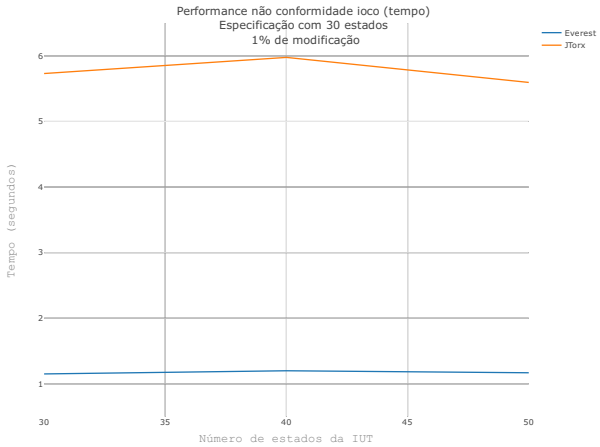
As Figuras 49, 50 e 51 indicam uma tendência de que quanto maior o número de estados dos modelos de especificação, para uma mesma porcentagem de variação das IUTs, maior a diferença no tempo de verificação entre as ferramentas Everest e JTorx. Os gráficos mostram que há uma diferença razoável entre os tempos de verificação na Everest em relação aos tempos obtidos pelo JTorx, nestes casos com IUTs de 1%, 2% e 4% de modificações. No entanto, percebe-se que também há uma proporcionalidade entre as ferramentas, porém sempre com uma eficiência maior na Everest.



(a) Especificação com 10 estados



(b) Especificação com 20 estados



(c) Especificação com 30 estados

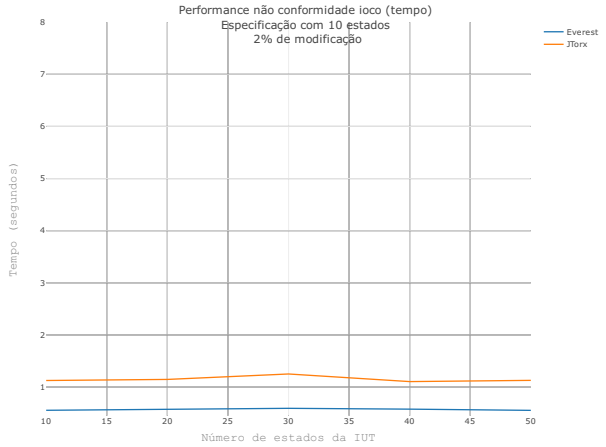
Figura 49 – Experimento: não conformidade **ioco** (1%) - especificações com 10, 20 e 30 estados.

Na Figura 49 a variação no tempo de verificação da EVEREST, considerando os diferentes tamanhos de IUTs, é de até $1,72\%$ para especificações com 10 estados, enquanto que no JTorx essa variação é mais de 14%. Para especificações com 20 estados a variação é de $5,41\%$ na EVEREST e de até $17,52\%$ no JTorx, e para especificações com 30 estados a oscilação do tempo de verificação é de até $3,48\%$ na EVEREST e de até $6,8\%$ no JTorx.

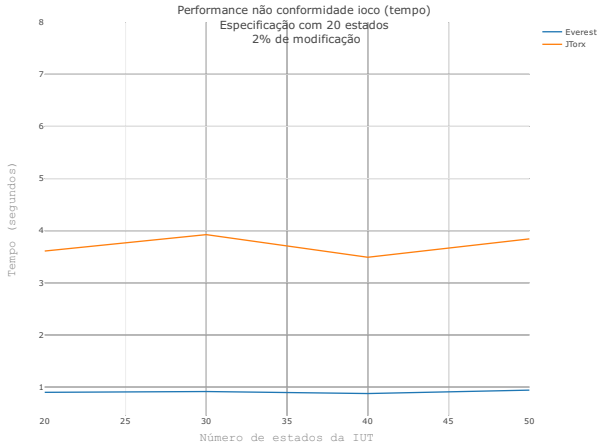
Já pela Figura 50 nota-se uma diferença considerável nos tempos obtidos, por ambas as ferramentas, para cenários de não conformidade, usando IUTs com modificações em 2% das transições.

Na figura 50a, com especificações de 10 estados e IUTs variando de 10 a 50 estados, a EVEREST executou a verificação de conformidade, em média, 0,6 segundos mais rápido do que o JTorx. No gráfico da Figura 50b observa-se que a EVEREST executa as verificações em menos de 1 segundo, para especificações com 20 estados, enquanto o JTorx leva mais de 3,5 segundos. Por fim, a Figura 50c mostra que a ferramenta EVEREST leva menos de 2 segundos para executar

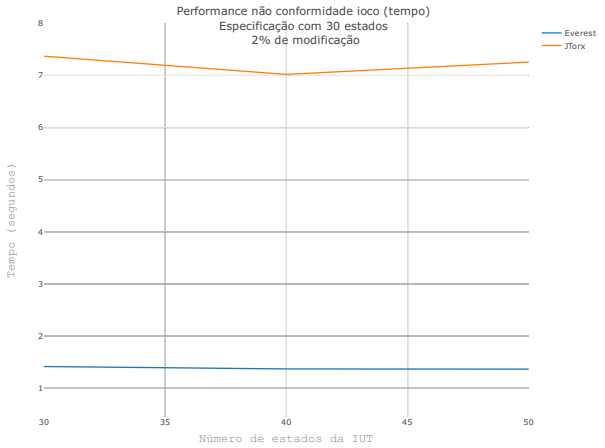
as verificações em especificações com 30 estados e IUTs com 30, 40 e 50 estados, enquanto que o JTorx leva em média mais de 7 segundos. Além disso, a distribuição da média de tempo das execuções da EVEREST se mostrou mais uniforme do que no JTorx. Nestes casos, constatou-se que a ferramenta EVEREST é capaz de lidar com IUTs contendo diferentes números de estados de forma mais estável com relação aos tempos de execução.



(a) Especificação com 10 estados



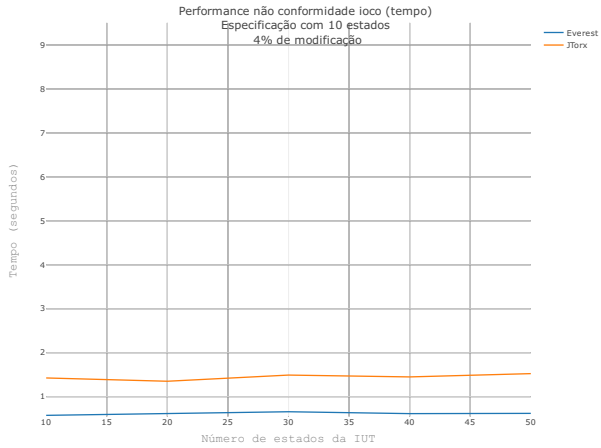
(b) Especificação com 20 estados



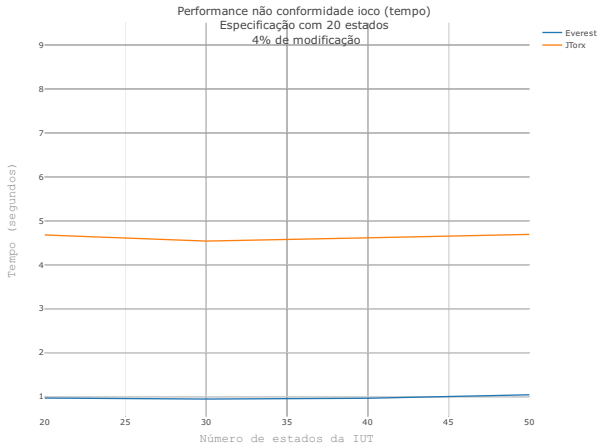
(c) Especificação com 30 estados

Figura 50 – Experimento: não conformidade **ioco** (2%) - especificações com 10, 20 e 30 estados.

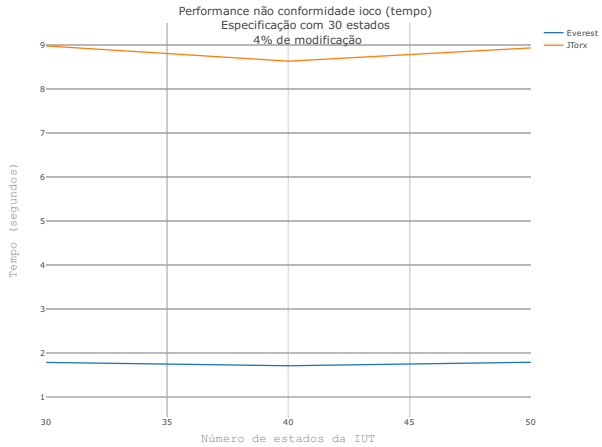
Os resultados apresentados nos gráficos da Figura 51, com modelos de IUTs com 4% de modificações em relação as especificações, deixam mais evidente a diferença no tempo de verificação entre as ferramentas JTorx e EVEREST. Para especificações com 10 estados a EVEREST executa as verificações entre 121, 31% e 149, 18% mais rápida que a ferramenta JTorx, para especificações com 20 estados a EVEREST leva vantagem de 350, 96% à 386, 46%, chegando até uma diferença entre 401, 69% e 407, 65% para especificações com 30 estados.



(a) Especificação com 10 estados



(b) Especificação com 20 estados



(c) Especificação com 30 estados

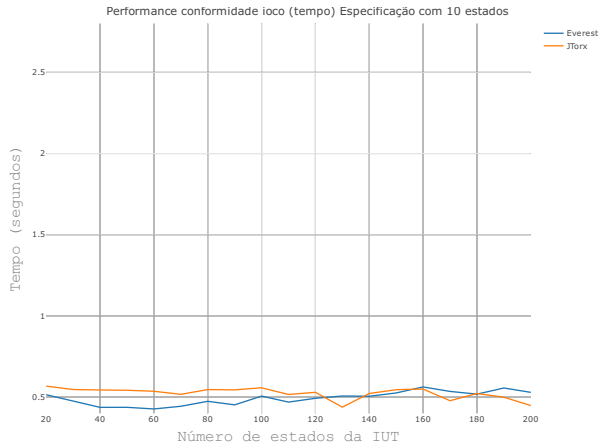
Figura 51 – Experimento: não conformidade **ioco** (4%) - especificações com 10, 20 e 30 estados.

6.4.1.3 Teste de *stress*

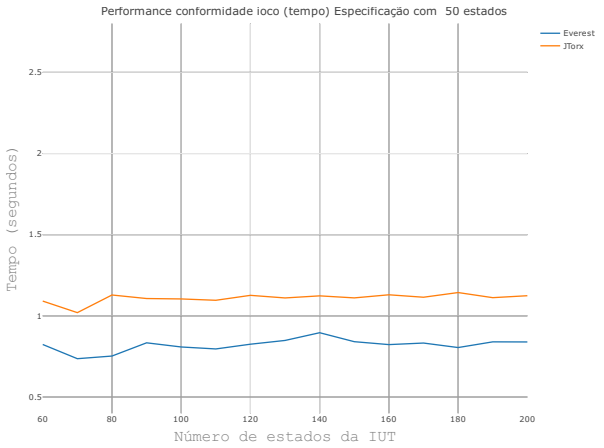
No teste de *stress* foram utilizados modelos com um maior número de estados e transições. O objetivo foi avaliar os limites das ferramentas e seus respectivos comportamentos. As especificações geradas para os experimentos são de 10 estados com IUTs variando de 20 a 200 estados num cenário, de 50 estados e IUTs variando de 60 a 200 estados num segundo cenário, e de 100 estados com IUTs entre 110 e 200 estados, sendo que todos os conjuntos de IUTs aumentam de 10 estados a cada variação.

Nesse experimento a questão de pesquisa é *"Qual o limite no número de estados e transições dos modelos, considerando veredito de conformidade e não conformidade, quando as ferramentas estão próximas de sua capacidade em relação ao tempo de execução?"*. Nos experimentos com especificações de 10 estados (Figura 52a) pode-se observar um melhor desempenho da ferramenta EVEREST para IUTs de até 120 estados. Para verificações em IUTs com mais de 120 estados o desempenho das ferramentas são similares ou ligeiramente melhor no JTorx,

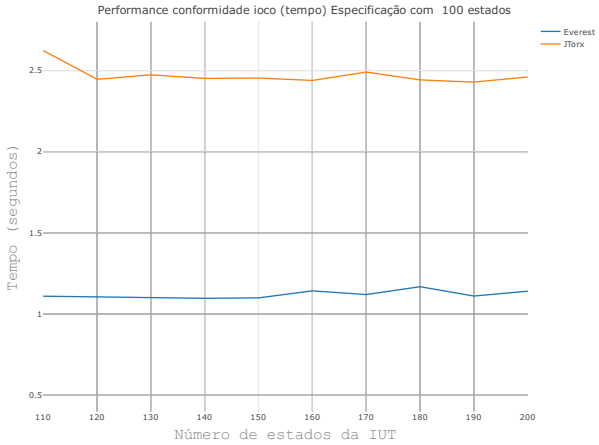
em alguns casos. Já para especificações com 50 e 100 estados (Figura 52b e 52c) a ferramenta EVEREST apresenta melhor desempenho levando-se em conta todos os conjuntos de IUTs até 200 estados.



(a) Especificação com 10 estados



(b) Especificação com 50 estados



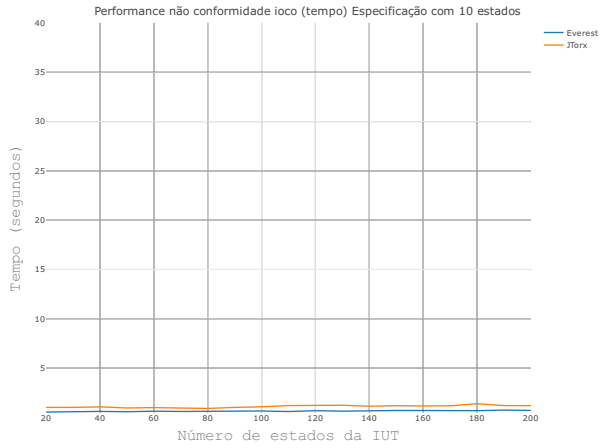
(c) Especificação com 100 estados

Figura 52 – Experimento: conformidade **ioco** - especificações com 10, 50 e 100 estados.

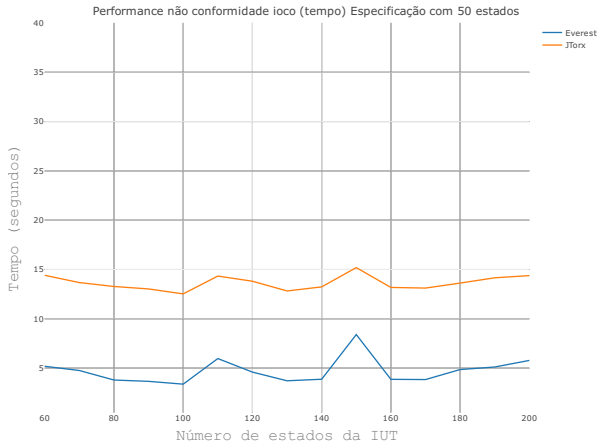
Nas verificações com especificações de 10 e 50 estados (Figura 53a e 53b), considerando vereditos de não conformidade, a EVEREST levou um menor tempo de execução em todos os experimentos. Já nas verificações de especificações com 100 estados, o JTorx apresentou melhor desempenho para IUTs com 200 estados, e pior desempenho que a EVEREST para IUTs entre 110 e 190 estados. Logo, conclui-se que, neste contexto, o JTorx possui melhor desempenho que a EVEREST, quando as IUTs possuem um número muito maior de estados em relação a sua respectiva especificação, ou seja, são substancialmente distintas estruturalmente.

6.4.2 Geração e Execução de Testes

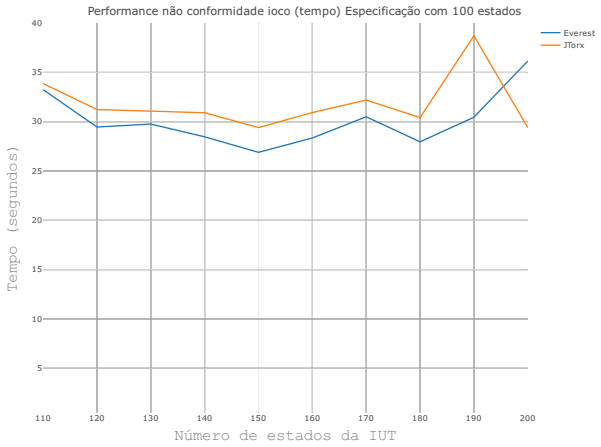
Os parâmetros analisados nos experimentos de geração de conjuntos de teste, da EVEREST, abrangem a variação no número de estados das especificações e a variação do número de estados



(a) Especificação com 10 estados



(b) Especificação com 50 estados



(c) Especificação com 100 estados

Figura 53 – Experimento: não conformidade **ioco** - especificações com 10, 50 e 100 estados.

das implementações a serem cobertas. Já na execução dos testes é analisado o desempenho da ferramenta quando há uma variação de certa porcentagem sobre as transições que distingue os modelos das IUTs em relação a especificação.

Vale destacar os motivos pelos quais não foram executados os experimentos de geração e execução de teste na ferramenta JTorx: (i) a geração de teste no Jtorx só é possível quando uma IUT também é fornecida, visto que a ferramenta adota a estratégia *online*; (ii) como o JTorx realiza a geração e a execução dos testes usando a estratégia *online*, quando a primeira falha é localizada a ferramenta termina sua execução não sendo possível definir um número de testes a serem gerados e executados, como ocorre na EVEREST ; (iii) dificuldade para capturar o tempo de geração e execução dos testes pelo JTorx, já que a ferramenta sobrepõe várias telas e processos, não permitindo uma avaliação de desempenho precisa.

6.4.2.1 Geração de multigrafos

Os experimentos de geração dos multigrafos mostraram que a EVEREST suporta especificações que variam de 5 à 35 estados, com até 250 transições. Neste caso, a ferramenta é capaz de cobrir IUTs com até 55 estados, portanto, o valor de m varia de 5 a 55 estados, crescendo de 10 estados a cada grupo de modelos. Constatou-se então que a ferramenta suporta a geração de multigrafos com até 67.000 estados e 674.000 transições, conforme as limitações computacionais. Vale ressaltar que os modelos de especificação contém alfabeto de tamanho dez, sendo cinco entradas e cinco saídas, e que as transições foram geradas aleatoriamente preservando a imparcialidade dos resultados.

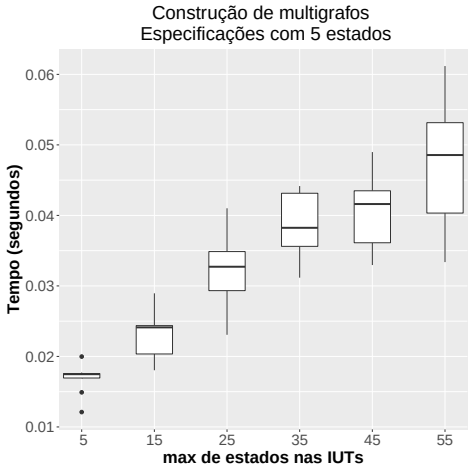
A questão de pesquisa relacionada a geração dos multigrafos é *"Qual o impacto no tempo de geração dos multigrafos conforme se varia o número de estados da especificação e o número de estados esperados na IUTs?"*. Neste experimento foram considerados os seguintes parâmetros: (i) especificações com 5 estados e m de 5 até 55 estados; (ii) especificações com 15 estados e m de 15 até 55 estados; (iii) especificações com 25 estados e m de 25 até 55 estados; e (iv) especificações com 35 estados e m de 35 até 55 estados.

A Figura 54 exibe o tempo gasto na geração dos multigrafos a partir das especificações e valores de m definidos. Observa-se pelos gráficos que o aumento no tamanho das especificações e no valor m induz o aumento no tempo de geração dos multigrafos.

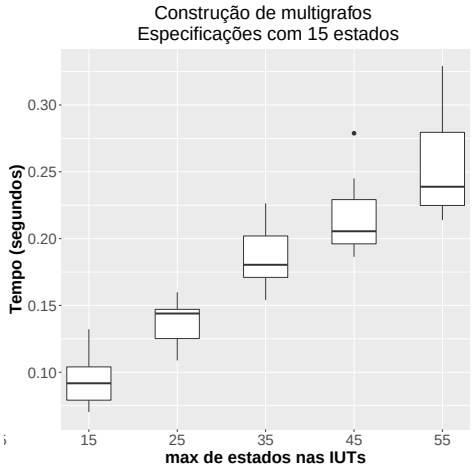
A construção de multigrafos a partir de especificações com 5 estados (Figura 54a) e $m = 35$ leva em média 0,038 segundos, e com $m = 55$ o tempo de geração é de 0,047 segundos, totalizando uma diferença de 23,68%. Para os multigrafos gerados a partir de especificações com 15 estados (Figura 54b) e $m = 35$ em média foram gastos 0,186 segundos e para $m = 55$ cerca de 0,253 segundos, totalizando uma diferença de 36,02%. Com especificações de 25 estados (Figura 54c) e $m = 35$ o tempo médio para a geração dos multigrafos foi de 0,428 segundos e para $m = 55$ foi de 0,676 segundos, resultando em 57,94% de diferença. Por fim, para especificações contendo 35 estados (Figura 54d) e $m = 35$, a média de tempo para a construção dos multigrafos foi de 0,994 segundos e para $m = 55$ foi de 1,867 segundos, totalizando 87,82% de diferença.

A geração de multigrafos para $m = 35$ é em média 46 vezes mais rápida para especificações com 5 estados em comparação as especificações com 35 estados. Já a construção com $m = 55$ e especificações com 5 estados é cerca de 26 vezes mais rápida do que para especificações com 55 estados. Logo, a diferença entre os tempos de geração dos multigrafos diminui entre especificações de 5 a 35 estados conforme se aumenta o parâmetro m de 35 até 55 estados para IUTs.

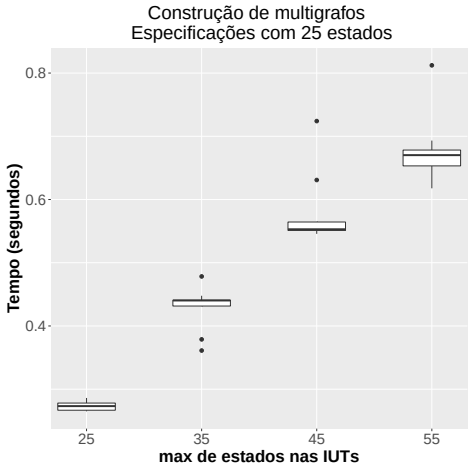
Observa-se em todos os experimentos de geração dos multigrafos poucos valores discrepantes (*outliers*), em relação ao tempo de geração. A maioria dos *outliers* ocorrem para os multigrafos das especificações com 25 estados.



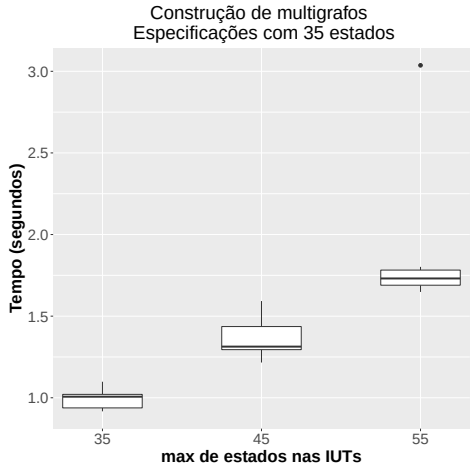
(a) Geração de multigrafos - especificações com 5 estados



(b) Geração de multigrafos - especificações com 15 estados



(c) Geração de multigrafos - especificações com 25 estados



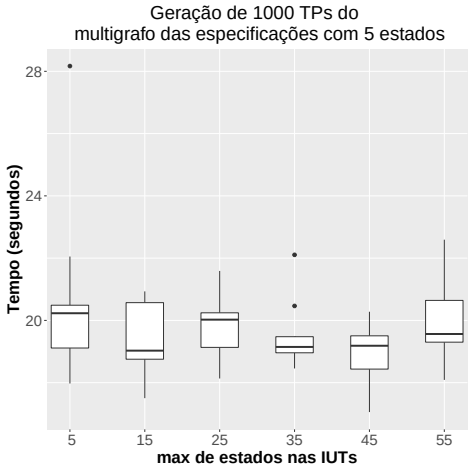
(d) Geração de multigrafos - especificações com 35 estados

Figura 54 – Experimento: geração de multigrafos - especificações com 5,15,25 e 35 estados.

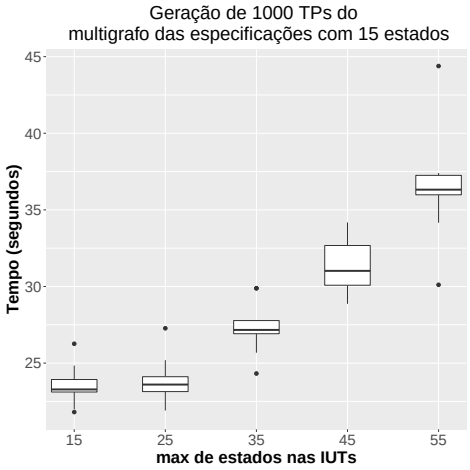
6.4.2.2 Geração de TPs

Na geração dos TPs a questão de pesquisa levantada é *"Qual o impacto no tempo de geração dos TPs a partir dos multigrafos, considerando as variações no número de estados da especificação e no número de estados esperados nas IUTs?"*. A partir dos multigrafos, das especificações de 5 a 35 estados e m variando entre 5 e 55, foram gerados 1000 TPs. A Figura 55 apresenta o tempo de geração dos TPs a partir dos multigrafos. Observa-se um aumento significativo no tempo de extração dos TPs em comparação com o tempo gasto na construção dos multigrafos.

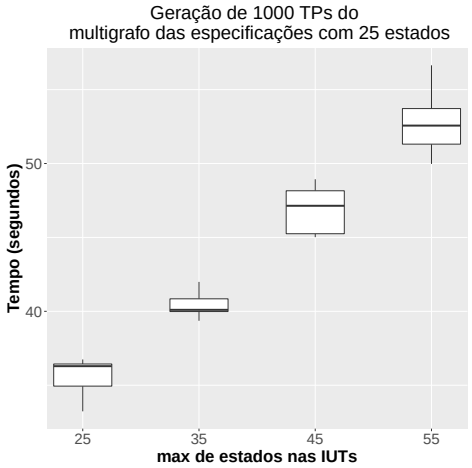
O gráfico da Figura 55a evidencia que o tempo de geração dos TPs, a partir dos multigrafos das especificações com 5 estados, apresentam medianas mais próximas. Este aspecto mostra uma maior homogeneidade no tempo de geração independente da variação de m do multigrafo.



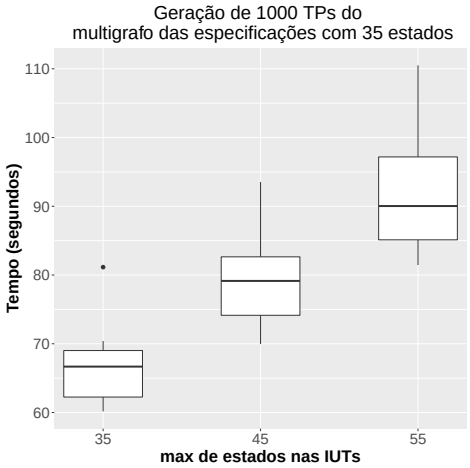
(a) Geração de TPs - especificações com 5 estados



(b) Geração de TPs - especificações com 15 estados



(c) Geração de TPs - especificações com 25 estados



(d) Geração de TPs - especificações com 35 estados

Figura 55 – Experimento: geração de TPs.

Os multigrafos gerados a partir de especificações com maior número de estados, 35 estados, apresentam uma maior dispersão de tempo para a geração dos TPs, como mostra a Figura 55d. Neste caso, o tempo médio para a geração dos TPs a partir dos multigrafos com $m = 35$ foi de 66,82 segundos e com $m = 55$ foi de 91,67 segundos, totalizando uma diferença de 37,19%. Na geração dos TPs, a partir dos multigrafos das especificações com 15 estados (Figura 55b), a diferença no tempo de geração entre $m = 35$ e $m = 55$ foi de 33,75%, embora este cenário tenha apresentado um maior número de *outliers*. Já o tempo gasto na geração dos TPs, a partir dos multigrafos construídos com as especificações de 25 estados (Figura 55c), para $m = 35$ foi em média 30,48% mais rápido do que em cenários com $m = 55$.

A geração dos TPs, a partir dos multigrafos das especificações com 5 estados e $m = 35$, foi em média 4 vezes mais rápida do que em multigrafos das especificações com 35 estados. O tempo de geração, considerando $m = 55$, foi em média 3 vezes mais rápido nos multigrafos das especificações com 5 estados se comparado com a geração dos TPs dos multigrafos originários

de especificações com 35 estados.

6.4.2.3 Execução dos casos de teste

A fim de avaliar o desempenho da execução dos testes, a questão de pesquisa é *"Qual o impacto no tempo de execução dos TPs em IUTs com 1%, 2% e 4% de modificações, sobre as transições, em relação as respectivas especificações utilizadas para gerar os multigrafos e extrair os TPs?"*. Foram considerados os TPs gerados a partir dos multigrafos das especificações com 15 e 25 estados, e valores de $m = n$, ou seja, o número de estados da especificação. Logo, para os multigrafos das especificações com 15 estados considerou-se o valor $m = 15$ e para os multigrafos das especificações com 25 estados o valor de $m = 25$. Para cada conjunto de TPs, extraídos de cada multigrafo, e cada percentual de variação das IUTs, foram geradas 10 IUTs contendo o mesmo número de estados que as especificações que deram origem ao respectivos multigrafos.

A Figura 56 apresenta os tempo de execução dos testes e o impacto da variação do tamanho das especificações com relação ao percentual de modificação das IUTs. A execução dos testes em IUTs com 1% de modificação é mais rápida quando são aplicados os TPs gerados a partir de multigrafos com $m = 15$ se comparado com $m = 25$. Já em IUTs com 2% e 4% de modificação os testes são executados de forma mais eficiente com os TPs gerados a partir de multigrafos com $m = 25$.

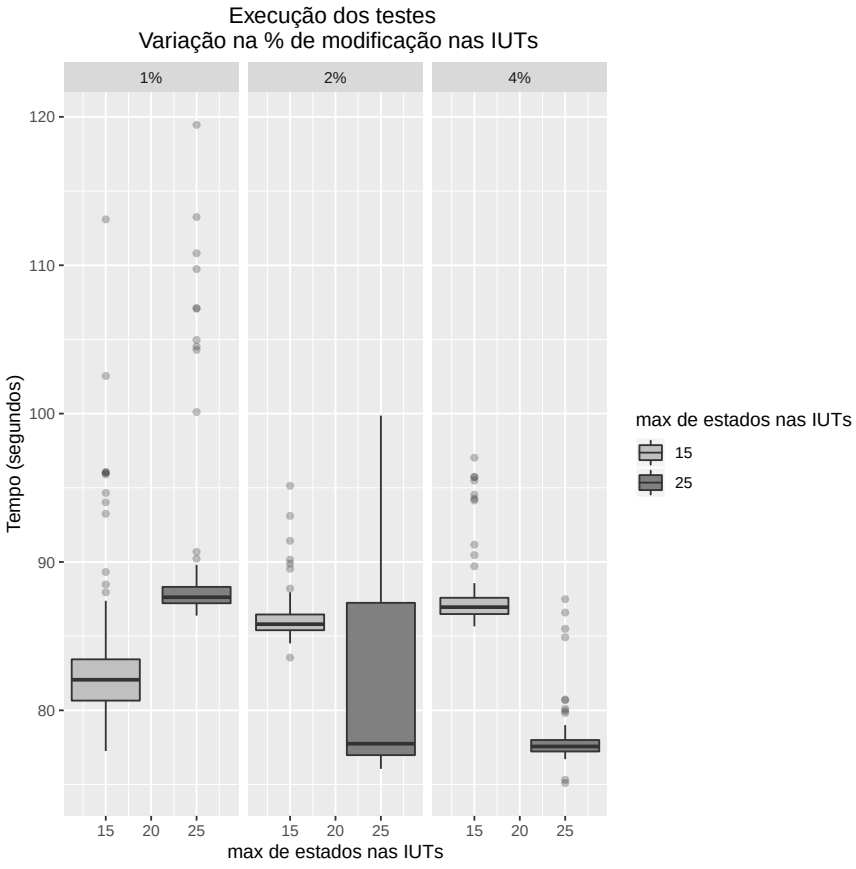


Figura 56 – Experimento: execução de teste.

Em geral a dispersão de tempo da execução dos testes é pequena dentro de um mesmo percentual de modificação das IUTs e mesmo valor de m. A maior dispersão de tempo está na execução dos testes em IUTs com 2% de modificação e TPs gerados para $m = 25$. A quantidade de *outliers* em cada *box* é de 10 a 15 %.

O tempo médio para a execução dos testes em IUTs com 1% de modificação aplicando os Tps gerados para $m = 15$ é de 83,03 segundos enquanto que para os TPs gerados com $m = 25$ a média do tempo de execução é de 89,78 segundos. Nas IUTs com 2% de modificação a execução dos TPs gerados para $m = 15$ leva em média 86,19 segundos, enquanto que para $m = 25$ o tempo é de 80,83 segundos. Já para a execução dos testes em IUTs com 4% de modificação, e TPs com $m = 15$ foram gastos em média de 87,54 segundos e com $m = 25$ o tempo de 77,83 segundos.

A análise sobre os tempos de execução dos testes em IUTs com 1% e 4% mostra que, para TPs cujo $m = 15$, as IUTs com 1% de variação executam 5,15% mais rápidas do que as IUTs com 4% de variação. Em TPs com cobertura de $m = 25$, as IUTs com 4% de modificação executam 13,31% mais rápidas do que em IUTs com 1% de modificação.

6.4.3 Ameaças a validade

Alguns fatores surgem como ameaça a validade dos experimentos realizados neste trabalho. Um primeiro fator está relacionado a dificuldade de obtenção do código fonte do JTorx, não permitindo uma adequação direta em código para uma captura mais precisa do tempo de execução dos experimentos. Os recursos computacionais onde foram realizados os experimentos também podem ser caracterizados como uma ameça, já que não eram máquinas dedicadas a realização exclusiva dos experimentos, podendo assim influenciar no resultado dos tempos de execuções obtidos.

Outra ameaça levantada está relacionada a geração aleatória das transições dos modelos, já que para modelos com mesmo número de estados, um número distinto de transições pode ser construído, podendo então interferir no tempo de execução dos experimentos. A geração das IUTs **ioco** conformes às respectivas especificações, como submáquinas, ou das IUTs **ioco** não conformes, de acordo com um percentual de modificação em relação a respectiva especificação, são também exemplos de ameaças. Essas IUTs podem trazer uma variabilidade menor para os testes já que outras IUTs **ioco** conformes, que não sejam submáquinas das especificações, ou modelos não conformes, que representam apenas partes das IUTs, não estão sendo consideradas. As propriedades presentes nos modelos gerados, devido as restrições do JTorx, também são ameaças a validade, pois o tamanho dos alfabetos e o número de estados/transições dos modelos de especificações e IUTs se modificam dos modelos originalmente objetos de teste.

Além disso, cada verificação de conformidade, entre uma especificação e uma IUT, e cada geração de teste foi executada apenas uma vez. Optou-se pela realização de um número maior de experimentos distintos, em torno de 34.500, para manter a imparcialidade, ao invés da repetição de um mesmo teste. A execução de todos estes testes mais de uma vez tornaria

inviável a realização dos experimentos devido ao tempo total que a tarefa demandaria.

6.5 Considerações do capítulo

Este capítulo apresentou os experimentos realizados para avaliação das ferramentas Everest e JTorx em forma de questões de pesquisa. A análise de desempenho das ferramentas leva em consideração o tempo de execução das verificações de conformidade, bem como para a avaliação da geração de testes no caso da Everest. Nos experimentos de verificação de conformidade, com a variação do tamanho do alfabeto de entrada e saída e vereditos de conformidade, nota-se que a variação do alfabeto não apresentou impacto significativo. O maior impacto no tempo de execução é constatado quando o veredito é de não conformidade. Ao variar o número de estados da especificação o desempenho da Everest melhora em relação ao tempo que consumido pelo JTorx, quando os cenários são de conformidade. Nos cenários de não conformidade observou-se que a Everest é capaz de lidar com IUTs contendo diferentes números de estados de modo mais estável que em relação ao tempo de execução do JTorx.

Os experimentos também revelaram que na maioria dos casos de teste a Everest obteve melhor desempenho que o JTorx, demonstrando que, embora o método implementado na Everest seja mais geral, seu desempenho não é afetado de forma negativa em relação ao tempo de execução das verificações de conformidade. Nos experimentos de geração e execução dos testes constatou-se que a Everest lida com modelos que possuem uma quantidade de estados e transições suficientes para modelar sistemas reais, considerando as limitações computacionais de memória. A geração e execução de testes usando a Everest também se mostrou ágil, visto que cada etapa, seja da construção do multigrafo, da extração dos TPs ou da execução dos testes, leva em média menos de 1.5 minuto. As possíveis ameaças relacionadas a validação dos experimentos também foram descritos neste capítulo.

7 CONCLUSÕES

A verificação de conformidade e a geração de testes são importantes atividades no desenvolvimento de sistemas reativos, cujo objetivo é aumentar a confiabilidade dos sistemas [22]. No teste de conformidade é verificado se uma implementação é válida com base na especificação, pois, a relação de conformidade serve para verificar matematicamente se a implementação esta correta. Na geração de teste, conjuntos de testes são automaticamente gerados dado uma especificação e um modelo de falha.

Este trabalho apresentou a EVEREST, uma ferramenta para teste automatizado de sistemas reativos. A ferramenta permite a verificação de conformidade entre IUTs e especificações através de uma relação de conformidade mais geral que a relação clássica **ioco**. O método mais geral, e consequentemente a EVEREST, impõe um número menor de restrições sobre os modelos e permite uma verificação de conformidade usando modelos de falhas descritos por linguagens regulares. A ferramenta é então capaz de verificar automaticamente a conformidade entre uma implementação e uma especificação, usando tanto a conformidade **ioco** quanto a conformidade baseada em linguagens. A EVEREST também implementa o módulo de geração de teste **ioco** completo usando a estrategia baseada em autômatos finitos. Assim, propósitos de teste **ioco** completo podem ser gerados sobre especificações fornecidas, e executados sobre implementações candidatas.

O trabalho inclui as construções das operações sobre linguagens regulares e dos respectivos algoritmos, bem como os algoritmos para o método teórico de verificação de conformidade e geração de testes proposto por Bonifácio e Moura [27]. A arquitetura e o projeto da EVEREST baseado na UML fornece suporte ao entendimento da ferramenta para a construção de novos módulos no futuro. O desenvolvimento da EVEREST em Java usando a biblioteca gráfica Swing também deve colaborar para a extensão da ferramenta em trabalhos futuros.

Estudos de caso também foram apresentados com o objetivo de explorar na prática as funcionalidades e diferenças entre as ferramentas EVEREST e JTorx. Estes estudos proporcionaram uma análise comparativa da EVEREST com a ferramenta JTorx, onde se constatou que o JTorx, por adotar a teoria **ioco**, não é capaz de capturar falhas que a ferramenta EVEREST detecta por meio da verificação de conformidade baseada em linguagens.

Experimentos práticos foram realizados para avaliar e comparar o desempenho das ferramentas, EVEREST e JTorx, na verificação de conformidade. Nos experimentos com variação no tamanho dos alfabetos de entrada e saída não houve impacto significativo no tempo das verificações com vereditos de conformidade, enquanto que nos experimentos para vereditos de não conformidade notou-se um maior impacto no tempo de execução das verificações. No geral, observou-se que a EVEREST tem melhor desempenho que a ferramenta JTorx na maioria dos cenários descritos, exceto quando a estrutura da IUT é significativamente diferente de sua respectiva especificação. Logo, embora a EVEREST implemente um método mais geral de

conformidade do que o JTorx não implica que o tempo de execução das verificações sofram um impacto negativo. Outra característica que se observa através dos experimentos é que a Everest possui um comportamento mais estável em relação ao tempo de execução mesmo lidando com IUTs com diferentes números de estados.

O desempenho sobre a geração e execução dos testes também foi realizado com a Everest. A ferramenta suportou a geração de conjuntos de teste para especificações com até 35 estados e IUTs com até 55 estados. A limitação ocorre particularmente devido a sua natureza combinatória na tarefa de extração dos TPs com base nos multigrafos gerados. Portanto, a geração e execução dos testes na Everest permite lidar com especificações e implementações candidatas com um número suficiente de estados para modelar cenários reais de sistemas. Já a verificação de conformidade permite lidar com modelos contendo um número bem maior de estados, visto que os experimentos realizados continham modelos com até 200 estados, permitindo ainda um possível aumento para modelos ainda maiores.

Como trabalhos futuros pretende-se investigar melhor alguns comportamentos da ferramenta apresentados pelos gráficos dos experimentos, como por exemplo aquele apresentado no gráfico da Figura 45b, variando o número de entradas e saídas. Neste caso, o tempo de verificação de conformidade da Everest diminui e na JTorx aumenta para IUTs a partir de 25 estados. Pretende-se também estender a interface de modo a permitir que a ferramenta realize verificação de conformidade e execução dos testes em modo *batch*, ou seja, a execução do teste em várias IUTs de modo automático. Outra pesquisa deixada para trabalhos futuros envolve a otimização dos algoritmos de geração de TPs e execução dos testes, permitindo assim modelos com um número maior de estados e transições. Espera-se também que em extensões futuras a ferramenta possa gerar e testar IUTs caixa-preta de maneira direta através de protocolos de comunicação com sistemas reais.

REFERÊNCIAS

[1] CADP: Construction and Analysis of Distributed Processes. Http://cadp.inria.fr/. Acessado em: 2019-02-28.

[2] Jararaca: Explore Traces Generated from Regular Expressions. Http://fmt.cs.utwente.nl/tools/torx/jararaca.1.html. Acessado em: 2019-02-28.

[3] JGraphX. Https://github.com/jgraph/jgraphx. Acessado em: 2019-02-25.

[4] JTorX: A Tool for Model-Based Testing. Https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/. Acessado em: 2018-03-23.

[5] TGV: The Test Sequence Generator TGV. Http://www.irisa.fr/pampa/VALIDATION/TGV/TGV.html. Acessado em: 2018-03-25.

[6] yEd Graph Editor. Https://www.yworks.com/products/yed. Acessado em: 2019-02-28.

[7] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*. ISO/IEC, 1 edition, 2002.

[8] M. S. AbouTrab, S. Counsell, and R. M. Hierons. Specification Mutation Analysis for Validating Timed Testing Approaches Based on Timed Automata. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 660–669, July 2012.

[9] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[10] Luca Aceto, Kim G. LarsenAnna, Kim Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, January 2007.

[11] Adenilso da Silva Simao and Alexandre Petrenko. Generating Complete and Finite Test Suite for ioco: Is It Possible? In *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, pages 56–70, 2014.

[12] M. Aggarwal and S. Sabharwal. Test Case Generation from UML State Machine Diagram: A Survey. In *2012 Third International Conference on Computer and Communication Technology*, pages 133–140, November 2012.

[13] Bernhard K. Aichernig, Elisabeth Jobstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97:383–404, 2015. Special Issue: Selected Papers from the 12th International Conference on Quality Software (QSIC 2012).

[14] Bernhard K. Aichernig and Martin Tappler. Symbolic Input-Output Conformance Checking for Model-Based Mutation Testing. *Electron. Notes Theor. Comput. Sci.*, 320:3–19, 2016. The 1st workshop on Uses of Symbolic Execution (USE).

[15] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[16] R. Anbunathan and A. Basu. Dataflow test case generation from UML Class diagrams. In *2013 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–9, December 2013.

[17] A.W.Roscoe. *Understanding Concurrent Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[18] Jones C. B. *Systematic Software Development Using VDM*. Prentice-Hall, 2 edition, 1990.

[19] A. Bandyopadhyay and S. Ghosh. Using UML Sequence Diagrams and State Machines for Test Input Generation. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–310, November 2008.

[20] Axel Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[21] Axel Belinfante. JTorX: Exploring Model-Based Testing, September 2014. IPA Dissertation series no. 2014-09.

[22] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, and Lex Heerink. *Formal Test Automation: A Simple Experiment*, pages 179–196. Springer US, Boston, MA, 1999.

[23] Axel Belinfante, Lars Frantzen, and Christian Schallhart. *14 Tools for Test Case Generation*, pages 391–438. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[24] Puneet Bhateja. A TGV-like Approach for Asynchronous Testing. In *Proceedings of the 7th India Software Engineering Conference*, ISEC '14, New York, NY, USA, 2014. ACM.

[25] Rex Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, second edition, 2002.

[26] Adilson Bonifacio, Arnaldo Moura, and Adenilso Simao. Model Partitions and Compact Test Case Suites. *International Journal of Foundations of Computer Science*, 23:147–172, April 2012.

[27] Adilson Luiz Bonifacio and Arnaldo Vieira Moura. Complete Test Suites for Input/Output Systems, 2019.

[28] brics. dk brics automaton. <http://www.brics.dk/automaton/>. Accessed: 2019-02-05.

[29] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2005.

[30] Christian Bucanac. The V Method. <http://www.bucanac.com/documents/The_V-Model.pdf>. Accessed: 2019-02-10.

[31] cadp. AUT manual page. <https://cadp.inria.fr/man/aut.html>. Accessed: 2019-02-05.

[32] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Test Case Generation By Means Of Uml Sequence Diagrams And Labeled Transition Systems. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 1292–1297, October 2007.

[33] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, and Laurent Mounier. Using UML For Automatic Test Generation. In *In International Symposium On Software Testing And Analysis Issta*. Springer-Verlag, 2002.

[34] Donghuo Chen, Xuandong Li, and Shizhong Zhao. Auto-generation and redundancy reduction of test cases for reactive systems. In *2010 2nd International Conference on Software Technology and Engineering*, volume 1, October 2010.

[35] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.*, SE-4(3):178–187, May 1978.

[36] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Symbolic Test Generation Tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–475, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[37] P. Daca, T. A. Henzinger, W. Krenn, and D. Nickovic. Compositional Specifications for ioco Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 373–382, March 2014.

[38] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 220–229, December 1998.

[39] A. En-Nouaary and G. Liu. Timed test cases generation based on test purposes expressed as message sequence charts. In *Proceedings. 2004 International Conference on Information and Communication Technologies: From Theory to Applications, 2004.*, pages 585–586, April 2004.

[40] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[41] H. Fouchal, E. Petitjean, and S. Salva. Testing timed systems with timed purposes. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, pages 166–171, December 2000.

[42] Eclipse Foundation. Oxygen 3 - Eclipse Java EE IDE for Web Developers. <https://www.eclipse.org/>. Accessed: 2019-02-04.

[43] Free Software Foundation. GNU Geberal Public License. 2007. <https://www.gnu.org/licenses/gpl-3.0.html>. Accessed: 2019-02-10.

[44] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, June 1991.

[45] Joseph R. Kiniry Gary T. Leavens, Patrice Chalin and Erik Poll. The java modeling language (jml) home page. <http://www.jmlspecs.org>, 2006. Accessed: 2019-09-20.

[46] Berry Gerard. Real Time Programming: Special Purpose or General Purpose Languages. *IFIP Word Computer Congress*, page 11, 1989.

[47] Hendrik Brinksma G.J. Tretmans. TorX: Automated Model-Based Testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[48] S. Goren and F. J. Ferguson. Testing finite state machines based on a structural coverage metric. In *Proceedings. International Test Conference*, pages 773–780, October 2002.

[49] Grady Booch and James Rumbaugh and Ivar Jacobson. *UML Guia do Usuario*. Elsevier, 2006.

[50] Gilleanes T. A. Guedes. *UML 2 Uma Abordagem Pratica*. Novatec, February 2018.

[51] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*, volume 215. Springer US, first edition, 1993.

[52] D. Harel and A. Pnueli. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*, pages 477–498. Springer Berlin Heidelberg, 1985.

[53] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, New York, 1998.

[54] F. C. Hennine. Fault detecting experiments for sequential circuits. In *1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, November 1964.

[55] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[56] T. Hosokawa, H. Date, and M. Muraoka. A state reduction method for non-scan based FSM testing with don't care inputs identification technique. In *Proceedings of the 11th Asian Test Symposium, 2002. (ATS '02).*, pages 55–60, November 2002.

[57] J. C. Huang. An Approach to Program Testing. *ACM Comput. Surv.*, 7(3):113–128, September 1975.

[58] Bowen J. Z website. <http://vl.zuser.org.>, 2006. Accessed: 2019-09-20.

[59] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, August 2005.

[60] Jean Claude Fernandez and Claude Jard and Thierry Jeron and Cesar Viho. An Experiment In Automatic Generation Of Test Suites For Protocols With Verification Technology. *Science of Computer Programming*, (29):123–146, 1997.

[61] K. E. Maadani and J. Geffroy. Identification of structured automata for test evaluation. In *Digest of Papers 1991 VLSI Test Symposium 'Chip-to-System Test Concerns for the 90's*, pages 40–46, April 1991.

[62] Muhammad Uzair khan, Sidra iftikhar, Muhammad Zohaib Iqbal, and Salman Sherin. Empirical Studies Omit Reporting Necessary Details: A Systematic Literature Review Of Reporting Quality In Model Based Testing. *Computer Standards and Interfaces*, 55:156–170, 2018.

[63] Axel Lamsweerde. Formal specification: A roadmap. pages 147–159, 01 2000.

[64] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306. ACM, 2005.

[65] William E. Lewis. *Software Testing and Continuous Quality Improvement*. Auerbach Publications, 2008.

[66] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer New York, 1995.

[67] Bruno Marre and Agnes Arnould. Test sequences generation from lustre descriptions: Gatel. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ASE '00, pages 229–, Washington, DC, USA, 2000. IEEE Computer Society.

[68] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Higher Education, 1997.

[69] MathWork. Mathwork website. <https://www.mathworks.com/>, 1994. Accessed: 2019-09-20.

[70] A. Matrosova and S. Ostanin. Self-checking FSM design with observing only FSM outputs. In *Proceedings 6th IEEE International On-Line Testing Workshop (Cat. No.PR00646)*, pages 153–154, July 2000.

[71] M. A. Mehmood, M. N. A. Khan, and W. Afzal. Transforming context-aware application development model into a testing model. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 177–182, November 2017.

[72] A. M. Mirza and M. N. A. Khan. An Automated Functional Testing Framework for Context-Aware Applications. *IEEE Access*, 6:46568–46583, 2018.

[73] Wojciech Mostowski, Erik Poll, Julien Schmaltz, Jan Tretmans, and Ronny Wichers Schreur. Model-Based Testing of Electronic Passports. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, pages 207–209, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[74] G. Mrugalski, J. Rajski, and J. Tyszer. Cellular automata-based test pattern generators with phase shifters. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 19(8):878–893, August 2000.

[75] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley Publishing, second edition, 2018.

[76] Sachio Naito and Masahiro Tsunoyama. Fault-Detection Efficiency of Sequential Machine Using Transition-Tour. *IEEE Fault Tolerant Computing Conference*, pages 238–243, 1980.

[77] Oracle. Java SE Development Kit 8. <http://www.oracle.com/technetwork/pt/java/javase/>. Accessed: 2019-01-30.

[78] Oracle. Package javax swing. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>. Accessed: 2019-01-31.

[79] Sofia Costa Paiva, Adenilso Simao, Mahsa Varshosaz, and Mohammad Reza Mousavi. Complete IOCO Test Cases: A Case Study. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 38–44, New York, NY, USA, 2016. ACM.

[80] Ron Patton. *Software Testing*. Sams, Indianapolis, IN, USA, second edition, 2005.

[81] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 2010.

[82] Roger Pressman. *Engenharia de Software: Uma Abordagem Profissional*. AMGH Editora Ltda, 2011.

[83] Jens R. Calame. Specification-based test generation with TGV. 2005.

[84] Microsoft Research. Spec<#>. <http://research.microsoft.com/specsharp>, 2006. Accessed: 2019-09-20.

[85] H. Reza and L. Cheng. Context-Based Testing of COTs Using Petri Nets. In *2012 Ninth International Conference on Information Technology - New Generations*, pages 572–577, April 2012.

[86] R.J.Wieringa and Jackson, M. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. ITPro collection. Elsevier Science, 2003.

[87] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[88] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2006.

[89] G.J. Tretmans and Hendrik Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, 12 2003.

[90] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.

[91] Jan Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, December 1996.

[92] Jan Tretmans. Testing Concurrent Systems: A Formal Approach. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR'99 Concurrency Theory*, pages 46–65, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[93] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[94] Mark Utting and Bruno Legeard. *practical model-based testing a tools approach*. Elsevier, first edition, 2007.

[95] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional Testing With Ioco. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, pages 86–100, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[96] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, July 1973.

[97] H. Wang and W. K. Chan. Weaving Context Sensitivity into Test Suite Construction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 610–614, November 2009.

[98] X. Wang, L. Guo, and H. Miao. An Approach to Transforming UML Model to FSM Model for Automatic Testing. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 251–254, December 2008.

[99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[100]J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, Oct 1994.

[101]B. Yang and H. Ural. Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping. *SIGCOMM Comput. Commun. Rev.*, 20(4):118–125, August 1990.

[102]Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich. Advances in test generation for testing software and systems. *Int. J. Software Tools Technol. Trans.*, 18(3):245–249, June 2016.

[103]F. Zeng, Z. Chen, Q. Cao, and L. Mao. Research on Method of Object-Oriented Test Cases Generation Based on UML and LTS. In *2009 First International Conference on Information Science and Engineering*, pages 5055–5058, December 2009.