Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Critérios de Geração de Casos de Teste de Sistemas de Tempo Real

Diego Rodrigues de Almeida

Campina Grande – Pb
2012

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Critérios de Geração de Casos de Teste de Sistemas de Tempo Real

Diego Rodrigues de Almeida

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Metodologia e Técnicas da Computação

Patrícia Duarte de Lima Machado e Wilkerson de Lucena Andrade

(Orientadores)

Campina Grande, Paraíba, Brasil

A447c Almeida, Diego Rodrigues de.
Critérios de geração de casos de teste de sistemas de tempo real / Diego Rodrigues de Almeida. - Campina Grande, 2012.
150 f.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2012.
"Orientação : Profª. Ph.D Patrícia Duarte de Lima Machado, Prof. D.Sc. Wilkerson de Lucena Andrade".
Referências.

1. Testes. 2. Sistemas de Tempo Real. 3. Teste Baseado em Modelos. 4. Dissertação - Ciência da Computação. I. Machado, Patrícia Duarte de Lima. II. Andrade, Wilkerson de Lucena. III. Universidade Federal de Campina Grande - Campina Grande (PB). IV. Título

CDU 004.415.53(043)

"CRITÉRIOS DE GERAÇÃO DE CASOS DE TESTE DE SISTEMAS DE TEMPO REAL"

DIEGO RODRIGUES DE ALMEIDA

DISSERTAÇÃO APROVADA EM 10/09/2012

PATRICIA DUARTE DE LIMA MACHADO, Ph.D
Orientador(a)

WILKERSON DE LUCENA ANDRADE, D.Sc
Orientador(a)

ADALBERTO CAJUEIRO DE FARIAS, Dr.
Examinador(a)

EDUARDO HENRIQUE DA SILVA ARANHA, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Atualmente, sistemas computacionais têm cada vez mais tomado espaço na vida da sociedade nos mais diversos setores. É possível encontrar software em funcionamento em sistemas de monitoramento de pacientes, sistemas de controle de tráfego aéreo, sistemas robóticos, veículos, etc. A maior parte dos sistemas em uso trabalha sob restrições de tempo. Sistemas cujo funcionamento correto não depende apenas das saídas produzidas, mas também do instante em que foram geradas são conhecidos como sistemas de tempo real. Testar é uma atividade que demanda um custo muito elevado e testar sistemas de tempo real é uma atividade ainda mais desafiadora e custosa. Assim, Teste Baseado em Modelos vem sendo uma técnica muito utilizada na geração de casos de teste tanto para sistemas em geral quanto para sistemas de tempo real. Para isso, ferramentas de geração de casos de teste baseadas em modelo recebem como entrada tanto o modelo do sistema sob teste quanto a forma como os testes são gerados. Essa forma como os testes são gerados é conhecida como critério de geração. O critério de geração reflete o algoritmo de geração de casos de teste e por sua vez determina quais elementos do modelo serão cobertos e, portanto, quais partes do sistema serão testadas. Há na literatura trabalhos que estudam critérios de geração para sistemas de tempo real, mas não os analisam observando sua efetividade, ou seja, relacionando tamanho do conjunto de casos de teste gerado com sua capacidade de revelar falhas. Não foram encontrados trabalhos na literatura que analisem critérios de geração para sistemas de tempo real nesse sentido. Assim, nesse trabalho foi proposto um conjunto de critérios de geração para geração de casos de teste baseados em modelos de sistemas de tempo real identificado através de uma revisão sistemática. Para este trabalho foi escolhido o modelo simbólico TIOSTS para descrição de sistemas de tempo real. A ferramenta SYMBOLRT foi estendida de forma a dar suporte à geração de casos de teste baseada em critérios de geração e um estudo experimental foi realizado utilizando seis modelos distintos e executados juntamente com todos os critérios de geração selecionados. Através do estudo experimental, pôde-se concluir que há diferença entre os critérios investigados em relação à capacidade de revelar falhas e ao tamanho do conjunto de casos de teste gerados por cada critério de geração. Assim, algumas conclusões subjetivas puderam ser obtidas as quais podem ser utilizadas para

auxiliar o testador na hora de escolher qual critério de geração adotar para geração de casos de teste de sistemas de tempo real.

Abstract

Nowadays, computer systems have increasingly taken place in society in many different sectors. You can find software operating systems in patient monitoring systems, air traffic control, robotic systems, vehicles, etc.. The majority part of this systems work under time constraints. Systems whose functioning depends not only on the correctness of the outputs produced, but also on the moment when these outputs are generated are known as real-time systems. Testing is an activity that demands a very high cost, but testing real-time systems is even more challenging and costly. Thus, Model Based Test has been a widely used technique to generate test cases for both systems in general and for real-time systems. So, model based test cases generation tools receive as input the model of the system under test and the way at which the tests will be generated. The way this tests are generated is known as generation criterion. The generation criterion reflects the test case generation algorithm and, therefore, determines which model elements will be covered. There are other works that study criteria for generating real-time systems, but do not analyze them watching their effectiveness, ie. relating the size of the set of test cases generated with ability to reveal faults. It were not find in the literature works that examine generation criteria for real-time systems in this sense. Thus, it was proposed in this work a set of generation criteria to generation of real-time systems model based test cases identified through a systematic review. For this work was chosen the symbolic model TIOSTS to describe real-time systems. The SYMBOLRT tool was extended in order to support the generation of test cases based on generation criteria and an experimental study was performed using six different models and executed along with all the selected generation criteria. By the experimental study, we concluded that there is difference between the investigated criteria in relation to the ability to reveal faults and the size of the set of test cases generated by each generation criterion. Thus, some subjective conclusions could be made which can be used to assist the tester when choosing which criteria to adopt for generating test cases for real-time systems.

Agradecimentos

Agradeço primeiramente a Deus pela saúde e oportunidade de poder ter tido uma boa educação tanto moral quanto acadêmica. Agradeço a Ele ainda, por ter ouvido muitas das preces que fiz e muito mais preces que não fiz. Por ter me guiado sempre pelo melhor caminho, mesmo que, muitas vezes, contra minha vontade.

Agradeço também aos meus pais José Neide e Maria de Fátima por terem sempre estado ao meu lado. Por terem tido a coragem de acreditar em mim, ainda quando eu mesmo não acreditava. Por terem sempre me apoiado nos meus momentos de alegria e mais ainda nos meus momentos de tristeza. Por terem aberto mão do próprio conforto para daí poder financiar a melhor educação que estivesse ao nosso alcance financeiro. Agradeço em especial ao meu pai, que mesmo não estando mais presente entre nós, o sinto sempre presente na batalha da minha vida e que com toda certeza continua, no plano em que se encontra, abrindo portas e me guiando nas minhas decisões.

Agradeço também aos meus irmãos Thiago Danillo e Diogo Rodrigues por me apoiarem nas minhas decisões corretas e por criticarem as equivocadas. Por terem sido companheiros e sempre zelarem pelo melhor para mim.

Agradeço aos meus orientadores Patrícia Machado e Wilkerson Andrade por terem sido muito pacientes e me mostrado o caminho a ser seguido para a conclusão deste trabalho. Além disso, agradeço a eles por terem sido companheiros e me apoiado quando tive problemas tanto acadêmicos quanto pessoais.

Por fim, agradeço a todas as pessoas as quais posso chamar de amigos e que de forma direta ou indireta contribuíram para este trabalho.

Conteúdo

Lista de Símbolos

ACSR - *Algebra of Communicating Shared Resources*

ART - *Adaptive Random Testing*

DBM - *Difference Bound Matrix*

ERA - *Event Recording Automata*

LTS - *Labeled Transition System*

MS - *Modelo Simbólico*

RT - *Random Testing*

RTES - *Real Time Embedded Systems*

SBT - *Search-Based Testing*

STR - *Sistema de Tempo Real*

SUT - *System Under Test*

SYMBOLRT - *Symbolic Model-Based Test Case Generation for Real-Time Systems*

TBM - *Teste Baseado em Modelo*

TIOA - *Timed Input Output Automata*

TIOSTS - *Timed Input-Output Symbolic Transition Systems*

TROM - *Timed Reactive Object Model*

ZSET - *Zone-based Symbolic Execution Tree*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Um sistema de tempo real (STR) é um sistema cujo comportamento correto depende não somente das respostas produzidas pelo sistema, mas também do momento em que são produzidas [48]. Tais sistemas podem ter restrições de tempo não somente nas saídas produzidas, mas também no momento em que aceitam entradas do ambiente em que estão inseridos. Assim, para testar um sistema de tempo real, devemos levar em consideração não somente quais entradas estão sendo fornecidas para o STR, mas também quando são fornecidas. Para o comportamento correto de um STR, uma resposta não deve apenas produzir valores corretos, mas também os valores também devem ser produzidos no momento correto.

Restrição de tempo em uma resposta de um STR é comumente associada à rapidez da resposta. Uma solução equivocada em resolver problemas de restrição de tempo é melhorar desempenho de hardware de forma a permitir que o sistema possa produzir respostas mais rapidamente. No entanto, em alguns casos, rapidez na resposta dos STR pode ser, inclusive, uma falha do sistema. Em uma bomba de insulina, o sistema deve verificar o nível de glicose em intervalos periódicos e ele não precisa responder muito rápido a eventos externos [48].

Uma outra maneira de ver STRs é como um sistema de estímulos/respostas. Considerando um dado estímulo, o sistema deve produzir a sua resposta correspondente no momento certo. Estímulos podem ser classificados em duas maneiras:

- **Periódicos**: Ocorrem em intervalos de tempo previsíveis. Por exemplo, o sistema deve examinar o nível de glicose do paciente a cada segundo.
- **Aperiódicos**: Ocorrem em intervalos de tempo imprevisíveis. São geralmente tratados

por mecanismos de interrupção. Por exemplo, o sistema informa ao usuário quando a cópia de um arquivo de um pen-drive acabou.

Atualmente, sistemas de tempo real estão inseridos em grande variedade de situações do cotidiano, desde aparelhos domésticos, como forno de micro-ondas e máquinas de lavar, até sistemas mais complexos e críticos, como controladores de voo e monitoramento hospitalar. O fato de que sistemas de computadores não confiáveis podem causar graves problemas na nossa sociedade é indiscutível. Muitos são os casos em que tragédias ocorridas são associadas a falhas de sistema. Além de danos pessoais ou materias que um sistema incorreto pode causar para seu usuário ou proprietário, ele também pode ser caro para a empresa que o produz. Por esse motivo, desenvolvedores se esforçam para tornar seus sistemas o mais livre de falhas possível.

Verificação e validação (V & V) é o nome dado ao processo de verificação e análise do software a ser desenvolvido. V & V tem lugar em cada fase do processo de desenvolvimento do software [48] começando pela revisão dos requisitos, passando pela revisão do projeto e finalizando na implantação do produto.

O objetivo principal do processo de verificação e validação é estabelecer a confiança de que o sistema de software está 'apto para o propósito' [48]. Assim, através do processo de V & V, há técnicas para **validar** que o software a ser desenvolvido faz o que deve fazer e **verificar** que o faz de corretamente. Dentro do processo de V & V, existem duas abordagens complementares para a verificação e análise do sistema:

- **Inspeções de software**. Analisar e verificar as representações do sistema, tais como o documento de requisitos, diagramas de projeto e o código fonte do software. Inspeções de software e análises automatizadas são técnicas estáticas V & V, ou seja, não há necessidade de executar o software em um computador. São exemplos de inspeções de software que podem ser utilizadas para inspeção de STRs: *Model Checking* [26], [19], [11], Análise Estática de Código [53]

- **Teste de software**. Envolve a execução de uma implementação do software com dados de teste. Examina-se as saídas do software e seu comportamento operacional para verificar se ele está funcionando como necessário. O teste é uma técnica dinâmica de

verificação e validação. Como exemplo de teste de software que pode ser utilizado para teste de STRs podemos citar o Teste Baseado em Modelos (TBM) [43], [12]

Segundo Utting et al. [51], testar é uma atividade realizada para avaliar qualidade de produto, e para melhorá-lo, através da identificação de falhas e problemas. O teste é o método de detecção de falhas dominante para aumentar a confiança em um sistema de computador. É o processo de exercitar um sistema em um ambiente controlado e examinar se seu comportamento reflete os requisitos do sistema.

Teste de software pode seguir duas abordagens distintas: (i) teste caixa preta (também conhecido como teste funcional) é o teste em que partes internas do sistema ou componente são ignoradas e o foco do teste se dá na observação das saídas produzidas pelo sistema em resposta às entradas fornecidas; (ii) teste caixa branca (também conhecido como teste estrutural) é o teste que leva em consideração a estrutura interna do sistema ou componente.

TBM é uma técnica em que casos de teste são derivados de um modelo que especifica o comportamento esperado de um sistema a ser testado (SUT para *System Under Testing*). Os testes gerados servem para averiguar se a implementação está se comportando de acordo com o especificado nos modelos. Como vantagens de TBM podemos citar: i) a diminuição do tempo gasto para geração dos testes; ii) a efetividade dos testes gerados; e iii) a possibilidade da reflexão automática de mudanças de requisitos nos testes.

Em TBM, modelos formais com semânticas precisas são de grande importância, pois eles são factíveis de geração automática de casos de teste. Atualmente, há na literatura modelos formais que permitem modelar sistemas de tempo real [9], [37], [4]. Na prática, STRs manipulam tanto variáveis quanto parâmetros de ações. Porém, dentre os modelos formais conhecidos apenas o TIOSTS (*Timed Input-Output Symbolic Transition System*) proposto por Wilkerson [4] trata parâmetros, variáveis e clocks de maneira simbólica evitando assim o problema da explosão do espaço de estados. Além disso, já existe, inclusive, uma ferramenta para automação de geração de casos de teste para sistemas de tempo real modelados em TIOSTS. A ferramenta SYMBOLRT (*SYmbolic Model-Based test case generation toOL for Real-Time systems*) [5] implementa todo o processo de geração de caso de teste baseado em propósito de teste de sistemas de tempo real modelados em TIOSTS.

Outro fator importante em TBM é o critério de geração. Ferramentas que implementam geração de casos de teste seguindo TBM necessitam, além do modelo formal do SUT, as

diretrizes de como gerar, ou extrair, os casos de teste a partir do modelo. Critérios de geração de casos de teste são os meios de comunicação da sua escolha de testes para uma ferramenta de teste baseado em modelo [51]. Através deles, o testador comunica a ferramenta de TBM que tipos de casos de teste deseja extrair do modelo, quais elementos do modelo o conjunto de casos de teste deve cobrir e quais cenários de execução do SUT se deseja testar. Uma boa ferramenta de TBM deve dar suporte a vários tipos de critérios de geração, para permitir o maior controle possível sobre a geração de testes. A escolha do critério de geração influencia o algoritmo que as ferramentas usam para gerar testes que, por sua vez, tem impacto no tamanho do conjunto de casos de teste que será gerado, quanto tempo leva para gerá-los, e quais partes do modelo serão testadas. Por isso, a qualidade dos casos de teste gerados no tocante ao custo e capacidade de revelar falhas está relacionada ao critério de geração adotado.

Um aspecto importante no teste de um sistema consiste em gerar a menor quantidade de casos de teste que possam revelar o maior número de falhas possível. O poder de uma técnica de geração de casos de teste está usualmente relacionado à cobertura de falhas. Assim, idealmente, um critério de geração deve ser capaz de revelar o maior número de falhas possível. No entanto, há um *trade-off* entre capacidade de revelar falhas e tamanho do conjunto de casos de teste resultante da geração. Portanto, é importante conhecer tanto a capacidade de revelar falhas quanto o tamanho do conjunto de casos de teste que um critério de geração pode gerar. No contexto de TBM, apesar de o estudo de critérios de geração ser bem difundido na academia [46], [15], [45], [30], não foram encontrados na literatura estudos que relacionem capacidade de revelar falhas e tamanho de conjunto de casos de teste de critérios de geração de casos de teste para sistemas de tempo real. Estima-se que deve haver critérios de geração a partir dos quais podem-se gerar casos de teste com elevada capacidade de revelar falhas e com tamanho de conjunto de casos de teste não muito elevado e que, além disso, possibilite focar o teste em cenários de maior interesse (i.e. falhas provenientes de restrições de tempo). Assim, existe a necessidade de um estudo mais detalhado que possibilite o testador conhecer a capacidade de revelar falhas e tamanho de conjunto de casos de teste gerado por cada critério de geração, permitindo que o testador possa utilizar o critério correto para cada sistema de tempo real a ser testado. Além disso, é importante conhecer quais são os critérios de geração utilizados na literatura para gerar casos de teste de sistemas de tempo

real.

1.1 Objetivos

O objetivo geral desse trabalho é identificar e analisar critérios de geração de casos de teste para sistemas de tempo real. Mais especificamente, é objetivo desse trabalho identificar quais são os critérios de geração de casos de teste para sistemas de tempo real ao nível de modelo. Critérios de geração a partir de código estão fora do escopo desse trabalho. Os critérios são comparados dentro de um processo de TBM e a sua análise se baseia em comparar a capacidade de revelar falhas e relacionar essa capacidade com o tamanho do conjunto de casos de teste de cada critério.

Para isto, foram definidas as seguintes metas:

1. Realizar uma revisão sistemática com o objetivo de levantar os critérios de geração já propostos na literatura;
2. Buscar e modelar exemplos de sistemas de tempo real para serem utilizados como exemplos de um estudo experimental;
3. Implementar o processo de geração de casos de teste em TBM segundo cada critério que foi objeto de estudo;
4. Realizar o estudo experimental sobre os critérios já investigados.

1.2 Contribuição

Casos de teste em TBM são extraídos do modelo segundo algum critério de geração. A escolha de qual critério de geração adotar é uma decisão importante e muito difícil de se tomar. É necessário conhecer não somente o perfil dos casos de teste gerados, mas também qual sua capacidade de revelar falhas, o tamanho do conjunto de casos de teste gerados e quais elementos do modelo são mais cobertos pelos testes. Há trabalhos que avaliam critérios de geração para sistemas sem restrições de tempo ([46], [15], [45], [30], etc) no entanto não foram encontrados trabalhos que avaliem critérios de geração para sistemas de tempo real. Este trabalho realiza um estudo nesse sentido e traz como principais contribuições:

- **Revisão sistemática**: para selecionar com confiança quais são os critérios de geração de casos de teste para sistemas de tempo real em TBM, foi conduzida uma revisão sistemática. Essa revisão permitiu a produção de vários artefatos como: documentos explicando as fontes de pesquisa, termos (palavras-chave) consultados e todas as etapas de seleção, lista integral de todos os artigos retornados pela pesquisa. Tal material pode ser utilizado para propiciar a possível replicação do estudo por parte de outro pesquisador e no futuro encontrar possíveis novos critérios de geração que possam ser investigados;

- **Implementação do processo de geração de casos de teste para sistemas de tempo real baseado em critérios de geração**: para a realização do experimento, os critérios foram implementados na ferramenta SYMBOLRT [5]. A implementação dos critérios permite não somente a execução do experimento conduzido nesse trabalho, mas também possibilita que sistemas de tempo real sejam testados através da ferramenta SYMBOLRT e os casos de teste possam ser gerados segundo os critérios estudados nesse trabalho;

- **Algoritmos de critérios de geração**: Os algoritmos que descrevem o caminhamento no modelo e sua extração de casos de teste estão no Apêndice B desse trabalho e podem ser utilizados e/ou adaptados por outros trabalhos para implementação dos critérios de geração em outra ferramenta de automação de geração de casos de teste para sistemas de tempo real.

- **Experimento**: Através do experimento, pôde-se dar garantias estatísticas sobre as afirmações realizadas a respeito dos critérios de geração e, portanto, permite que haja mais segurança por parte do testador sobre a escolha de qual critério utilizar na geração de casos de teste do sistema de tempo real que queira testar;

- **Hierarquia de critérios**: Além das análises realizadas no experimento, a organização dos critérios em uma hierarquia de critérios permite que o testador possa avaliar melhor qual critério utilizar. A relação de inclusão da hierarquia permite que o testador conheça quais critérios de geração estão implicitamente sendo gerados a partir de um critério de geração mais alto na hierarquia de critérios.

1.3 Estrutura

As demais partes deste documento estão estruturadas da seguinte forma:

- **capítulo 2: Fundamentação Teórica.** Apresenta uma descrição de conceitos básicos necessários para compreender melhor este trabalho. Os conceitos descritos estão relacionados a teste de software, Sistemas de Tempo Real, Modelos Simbólicos, e Critérios de Geração de Teste;

- **capítulo 3: Critérios de Geração de Casos de Teste.** O capítulo apresenta uma descrição sobre os critérios de geração de casos de teste para sistemas de tempo real em TBM encontrados. Além disso, explica em mais detalhes cada um dos critérios que foram objetos de estudo do experimento conduzido nesse trabalho;

- **capítulo 4: Geração de Casos de Teste Baseada em Critérios de Geração.** O capítulo apresenta o processo de geração de casos de teste para sistemas de tempo real baseada em critérios de geração. O capítulo também descreve a ferramenta SYMBOLRT e a sua nova arquitetura de forma a dar suporte tanto ao processo de geração de casos de teste baseado em propósito quanto ao novo processo de geração de casos de teste baseado em critérios de geração;

- **capítulo 5: Experimento.** O capítulo detalha o experimento conduzido nesse trabalho. Nele, constam todas as etapas do experimento: definição, planejamento, instrumentação, execução e análise dos resultados;

- **capítulo 6: Trabalhos Relacionados.** O capítulo apresenta os principais trabalhos da literatura relacionados a critérios de geração de casos de teste para sistemas de tempo real em TBM. Os trabalhos foram resultados da revisão sistemática do Apêndice A e são resumidos neste capítulo. Além disso, neste capítulo são apresentadas argumentações que diferenciem os trabalhos relacionados com o apresentado neste documento;

- **capítulo 7: Conclusões.** Neste capítulo final, o trabalho desenvolvido é concluído através da apresentação dos resultados alcançados e as perspectivas para trabalhos futuros são apontadas.

Capítulo 2

Fundamentação Teórica

O objetivo deste capítulo é fornecer embasamento teórico para os leitores acerca dos conceitos utilizados neste trabalho. São apresentados os principais conceitos relacionados a teste, destacando o teste baseado no modelo simbólico TIOSTS, conceitos inerentes a Sistemas de Tempo Real, e critérios de geração de casos de teste.

2.1 Teste de Software

Teste de software é o processo de descobrir evidências de defeitos em um sistema de software. Um defeito pode ser introduzido em qualquer uma das fases de desenvolvimento, desde a concepção dos requisitos até a manutenção do produto após a entrega. Defeitos podem ser decorrentes de omissões, inconsistências ou mau entendimento dos requisitos ou especificações por parte do desenvolvedor [42]. Há, no contexto de teste de software, três termos muito utilizados na literatura e que são muitas vezes utilizados erroneamente como sinônimos [35]:

- **Erro:** Um erro é um engano ou omissão causada por uma ação humana. Tende a se propagar desde o desenvolvimento até a entrega do produto final;
- **Defeito:** Um defeito é o resultado ou uma representação de um erro. O tipo de defeito mais conhecido é em código cujo sinônimo é bug, mas também é possível e bem provável haver defeitos em modelos, textos, etc. Defeitos podem ser classificadas por omissão quando algum requisito não é implementado no sistema ou por comissão

quando algum requisito é implementado erroneamente;

- **Falha:** A falha é o resultado de um defeito. Quando o defeito é em código, a falha é o resultado da execução do código com defeito. Segundo Binder [10], uma falha é a manifestação da inabilidade do software em executar de forma correta uma determinada funcionalidade.

Teste de software pode ser utilizado para duas finalidades distintas [48]:

1. Demostrar para a equipe de desenvolvimento e para o cliente que o software atende às suas especificações. Nesse sentido, deve haver testes para cada funcionalidade do sistema a ser incorporada nas novas versões do software.

2. Para descobrir falhas no software. Nesse sentido a atividade de teste tem por objetivo descobrir todos os comportamentos indesejáveis do sistema tais como interações indesejáveis com outros sistemas, computações incorretas, corrupção de dados e aborto de execução do sistema.

O primeiro objetivo conduz ao teste de validação em que o teste é executado utilizando um dado conjunto de testes que refletem o uso esperado do sistema por um usuário. O segundo conduz ao teste de falha em que o conjunto de testes é projetado para expor falhas podendo, inclusive, o conjunto de testes não refletir ao uso normal do sistema (teste de *stress*). Esse conjunto de testes é conhecido em teste de software como casos de teste (Seção 2.1.3). Um teste de validação tem sucesso quando o sistema executa corretamente. Um teste de falha tem sucesso quando expõe falhas que causam a execução incorreta do sistema.

Além disso, teste de software pode seguir duas abordagens distintas focando as funcionalidades ou comportamentos do sistema (Teste Funcional, Seção 2.1.1) ou focando na estrutura interna do sistema (Teste Estrutural, Seção 2.1.2). Esse trabalho está inserido no contexto de Teste Funcional.

2.1.1 Teste Funcional

Teste Funcional (também muito conhecido como teste de caixa preta ou *black-box testing*) é um tipo de teste em que casos de teste são derivados a partir da especificação do sistema

a ser testado, isto é, o testador necessita apenas de informações sobre os dados de entrada, tais como domínio, faixa de valores, valores válidos e inválidos, etc., e conhecer as saídas esperadas para os dados de entrada, ou seja, o testador não precisa saber como o sistema funciona ou como ele computa os dados de entrada para produzir as saídas.

Em teste funcional, programas podem ser vistos como funções matemáticas que mapeiam valores de entrada para valores de saída [35]. A única informação utilizada é a especificação do software. Por isso, uma das vantagens do teste funcional é a independência de implementação, tornando possível que um componente do software possa ser alterado sem que seja necessário mudar os casos de teste que o testavam. Além disso, a atividade de teste pode ser desenvolvida em paralelo à implementação do código podendo, inclusive, os casos de teste serem gerados antes mesmo de iniciar a implementação de código. Uma desvantagem do teste funcional é a dificuldade de quantificar a atividade de teste. Em teste funcional é difícil dizer que um determinado conjunto de testes é suficiente ou tem uma quantidade ótima para testar todos os possíveis cenários de falhas. Outra desvantagem é garantir que partes essenciais ou críticas do software foram testadas [35].

2.1.2 Teste Estrutural

Teste estrutural (também muito conhecido como teste de caixa branca ou *White Box Testing*) é uma abordagem fundamental na identificação de casos de teste. Nele, o testador conhece e examina a estrutura e a lógica interna do programa ou sistema. Dados de teste são derivados examinando a lógica do programa ou do sistema sem levar em consideração os seus requisitos [40].

Uma vantagem de teste estrutural é que por esse tipo de teste estar focado em código, defeitos em partes específicas do sistema podem ser mais provavelmente detectadas. Outra vantagem é que técnicas de cobertura de código estão bem aperfeiçoadas [42].

Uma desvantagem é que essa abordagem de teste não verifica se a especificação está correta focando apenas na logica interna e não verificando a lógica da especificação [42]. Outra desvantagem é que não há como verificar falhas por omissão, por exemplo, se deveria ser implementada a condição $if|a-b| < 10$, mas foi implementada a condição $if(a-b) < 1$ os testes não irão revelar falhas devido a esse defeito.

2.1.3 Casos de Teste

A essência do teste de software é determinar o conjunto de casos de teste a serem testados. Um caso de teste é um conjunto de condições usadas para determinar se um sistema está implementado corretamente ou não. O conjunto de casos de teste especifica o que deve ser testado em termos de estímulos de entrada e respostas esperadas. Um caso de teste deve conter pelo menos as seguintes informações [35]:

- Entradas
  - **Pré-condições:** Condições que devem ser válidas antes da execução do caso de teste;
  - **Dados de entrada:** Dados utilizados na execução do caso de teste.
- Saídas
  - **Pós-condições:** Condições que devem ser válidas após a execução do caso de teste;
  - **Dados de saída:** Resultados esperados após a execução do caso de teste.

Estando o caso de teste definido, a atividade de teste engloba a definição das pré-condições necessárias, escolha dos dados de entrada, execução do caso de teste, observação dos resultados e, por fim, a comparação dos resultados obtidos com os resultados esperados para determinar se o teste passou ou não.

2.1.4 Níveis de Teste

Tradicionalmente, teste pode ser dividido em níveis, sendo eles: Teste de Unidade, Teste de Integração, Teste de Sistema e Teste de Aceitação [35]. Cada nível está associado a um estágio do desenvolvimento. O **Teste de Unidade** é o nível mais conhecido. Nele as unidades de código são testadas de forma isolada. O **Teste de Integração** verifica se há problemas nas interfaces entre as unidades, ou seja, verifica se as unidades quando trabalhando juntas resultam em alguma falha. Em processos de desenvolvimento baseados no modelo cascata, o **Teste de Sistema** verifica se o sistema como um todo executa de forma correta enquanto que em processos incrementais o Teste de Sistema verifica se uma *release* do sistema funciona

corretamente. Por fim, o **Teste de Aceitação** é realizado pelo usuário final do sistema e tem por objetivo verificar se os requisitos atendem as necessidades do cliente.

Após o teste de integração, o sistema é testado (seja ele uma *release* ou o produto final) com o objetivo de encontrar falhas ou má adequação dos requisitos que influenciam na qualidade final do produto. No teste de sistema um conjunto de testes podem ser executados para assegurar os atributos de qualidade que foram especificados no plano de garantia de qualidade de software e garantir que o teste de aceitação vá ocorrer livre de falhas [40]. Teste de sistema verifica se funções são executadas corretamente além de verificar se certas características não funcionais também estão presentes (i.e. disponibilidade, restrições de tempo, confiabilidade, etc.).

Alguns exemplos de testes que são executados no nível de teste de sistema são: teste de usabilidade, teste de performance, teste de *stress*, teste de compatibilidade, teste de conversão, teste de segurança, teste de escalabilidade, etc. O teste do sistema se insere no escopo dos testes funcionais (Seção 2.1.1) e não se detém apenas ao projeto do sistema, mas também ao comportamento. Este trabalho se insere no nível de teste de sistema.

2.1.5 Teste Baseado em Modelo

A primeira e talvez uma das mais importantes etapas no processo de desenvolvimento de software é o planejamento. Nela, são elaboradas as especificações que descrevem, através de um conjunto de documentos, o que é e como é o software a ser desenvolvido. Essas especificações documentam desde questões arquiteturais como tecnologias, linguagens de programação e padrões de projeto a serem usados na etapa de implementação até a descrição de como o software deve funcionar mediante interações com o seu ambiente. Na etapa de planejamento, modelos são elaborados com a finalidade de facilitar a comunicação entre equipes e documentar em alto nível o sistema a ser desenvolvido. Uma outra etapa fundamental do processo de desenvolvimento de software é a fase de testes. Ela tem por objetivo identificar evidências de defeitos que são inseridas por desenvolvedores. Defeitos são inseridos quando se omite, não entende ou se implementa de forma errada requisitos da especificação do sistema. Caso as falhas causadas por esses defeitos sejam identificadas tardiamente, pode ser muito custosa a correção do sistema. Assim sendo, torna-se imprescindível uma técnica que permita que testes acompanhem o desenvolvimento do sistema desde a concepção de sua

especificação.

Teste Baseado em Modelo (TBM) é o termo geral dado a um conjunto de técnicas baseadas nos modelos de aplicações a serem testadas com o objetivo de executar atividades de teste [22]. As atividades de TBM podem ser tanto de geração de casos de teste como de avaliação dos resultados de teste em relação aos resultados esperados podendo ter início na concepção dos modelos na fase de planejamento acompanhando todo o desenvolvimento do sistema.

As principais atividades do processo de TBM são [22]:

1. Construir o modelo: nessa atividade é construído um modelo partindo-se das especificações do sistema sob teste;

2. Gerar casos de teste: nessa atividade casos de teste são extraídos do modelo com a finalidade de verificar se o sistema está de acordo com as suas especificações;

3. Gerar oráculos de teste: nessa atividade o oráculo de teste é gerado a partir do modelo do sistema. Os oráculos são os elementos responsáveis por decidir se as saídas encontradas estão ou não de acordo com as especificações;

4. Executar os testes: nessa atividade o SUT (*System Under Test*) é executado com os casos de teste gerados, produzindo novas saídas;

5. Comparar resultados obtidos com os esperados: o oráculo compara as saídas produzidas pela execução dos testes com os resultados esperados.

Na Figura 2.1, é ilustrada a sequência das atividades descritas.

Uma vantagem em utilizar TBM é que existem diversos modelos formais com teorias bem fundamentadas bem como ferramentas para automação da geração e execução de casos de teste. Outra vantagem é que o modelo para a geração de casos de teste pode servir como comunicação entre equipes de desenvolvimento, porém, por haver vários formalismos e consequentemente notações diferentes, há uma necessidade de um investimento inicial para que as equipes possam se familiarizar com os conceitos acerca do modelo. Outra desvantagem de TBM consiste na ligação entre qualidade de modelo e qualidade dos testes de forma que modelos mal elaborados refletem em problemas na qualidade dos testes.

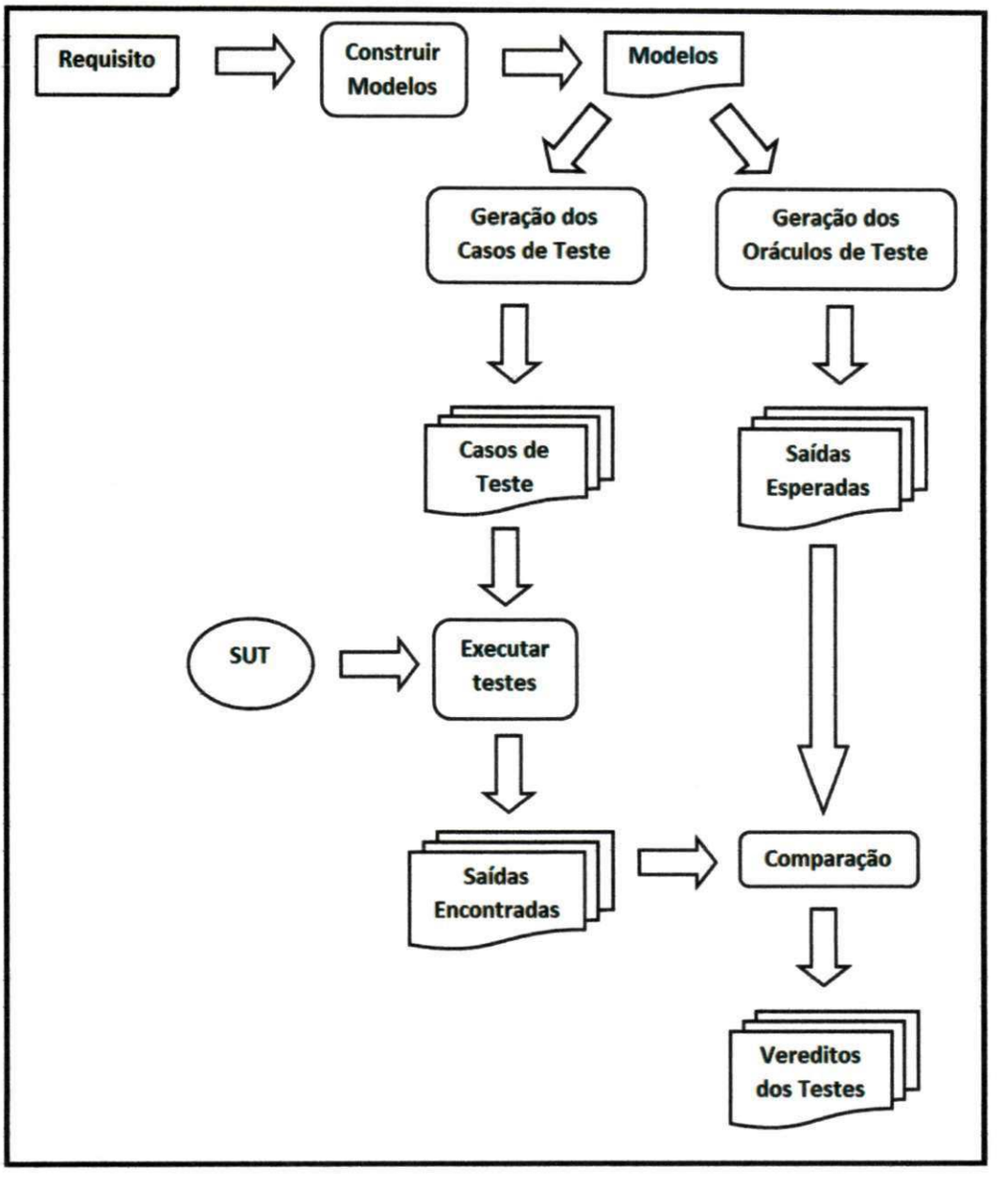


Figura 2.1: Atividades de TBM

2.2 Sistemas de Tempo Real

Ao longo dos anos, os sistemas computacionais têm se tornado cada vez mais complexos e utilizados nas mais diversas áreas. Muitas das aplicações desses sistemas necessitam não somente que ele responda corretamente aos estímulos do seu ambiente, mas que as respostas ocorram no momento correto. Sistemas computacionais com esse tipo de restrição são chamados de Sistemas de Tempo Real (STR). STR estão presentes nas mais diversas áreas, desde sistemas embarcados (e.g. celulares), até sistemas mais críticos (e.g. aparelhos hospitalares de monitoramento de pacientes). Dependendo do nível de confiança dos requisitos temporais de um sistema, ele pode ser classificado como *soft* ou *hard* [38].

Um STR *soft* é um sistema cujo desempenho é prejudicado caso os resultados não sejam produzidos no tempo esperado, ou seja, caso os resultados sejam produzidos em um tempo fora do esperado, o sistema continua funcionando e apenas a qualidade do seu serviço prestado é afetada.

Um STR *hard* é um sistema cujo desempenho é considerado incorreto caso os resultados esperados não sejam produzidos no tempo especificado. Os resultados produzidos no tempo fora do esperado em um sistema *hard* podem causar grandes prejuízos econômicos, ambientais ou perdas de vidas. Sistemas de controle de voo, sistemas de monitoramento como em hospitais e monitores de usinas nucleares são bons exemplos de sistemas de tempo real *hard*.

Um STR também pode ser visto como um sistema reativo com restrições de tempo. Sistemas reativos são sistemas que respondem (reagem) a eventos oriundos do ambiente o qual o sistema está inserido. Tais estímulos podem ser classificados como:

- **Periódicos:** ocorrem em intervalos de tempo previsíveis;
- **Aperiódicos:** ocorrem irregularmente e, normalmente, são tratados usando os mecanismos de interrupção.

Outra característica de sistemas de tempo real, principalmente sistemas de tempo real *hard*, e é tais sistemas devem apresentar comportamentos previsíveis, mesmo com recursos limitados, atendendo às restrições temporais impostas pelo ambiente ou pelo usuário.

2.3 Modelos Simbólicos

Atualmente, há várias teorias e técnicas de geração de casos de teste baseadas em variantes do modelo LTS (*Labeled Transition System*) clássico [50], [20], [39], [33], [14]. Modelos LTS, bem como suas variações, representam o comportamento de um sistema através de um grafo cujos vértices (também chamados de nós) são os possíveis estados que o sistema pode assumir e as arestas (também chamadas de transições) representam a mudança de um estado para outro a partir da ocorrência de ações.

No entanto, modelos LTS não são adequados quando as especificações requerem um domínio de dados muito grande ou infinito, pois cada valor desse domínio de dados é representado por um nó o que levaria o modelo a ser um grafo muito grande ou, caso o domínio seja infinito, seria necessário um grafo com infinitos nós tornando, portanto, a geração de casos de teste inviável [6]. Além disso, a execução de um caso de teste pode necessitar da implementação de um programa com parâmetros para permitir a troca de mensagens com o ambiente do SUT e cujas informações de estados do sistema devam ser armazenadas em variáveis. Por isso, buscou-se a criação de modelos mais poderosos que fossem capazes de representar variáveis e parâmetros de forma simbólica. Os Modelos Simbólicos (MS) são modelos capazes de descrever o comportamento de um sistema sem a necessidade de enumerar os valores dos dados desse sistema.

2.3.1 TIOSTS

Dentre as variações de MS existentes, podemos destacar um deles que possui um formalismo que se adequa a modelagem de STR, o Timed Input-Output Symbolic Transition Systems (TIOSTS) [7]. TIOSTS é um autômato simbólico, estendido do Input-Output Symbolic Transition Systems (IOSTS) [47], [18], [34], com características do Timed Automata [2]. O TIOSTS possui um conjunto finito de nós, um conjunto finito de arestas com ações carregando parâmetros para comunicação com o ambiente, um conjunto de variáveis tipadas usadas para representar os dados do sistema, e um conjunto finito de relógios usado para representar a evolução do tempo.

Através de TIOSTS é modelado o comportamento esperado do sistema de tempo real. Assim, descrições de situações de falha e anomalias do sistema não são modeladas em

TIOSTS sendo, portanto, documentadas em outros formalismos complementares na especificação do sistema de tempo real.

Na Figura 2.2, é ilustrado, graficamente, um exemplo de um TIOSTS que especifica o comportamento de uma máquina para carregar crédito no cartão de passagem de um metrô. O sistema permite adicionar créditos que serão consumidos por outra máquina no momento em que o passageiro vai embarcar no metrô. Para carregar crédito na máquina não é necessário inserir o valor exato a ser carregado, pois a máquina é capaz de devolver troco ao cliente.

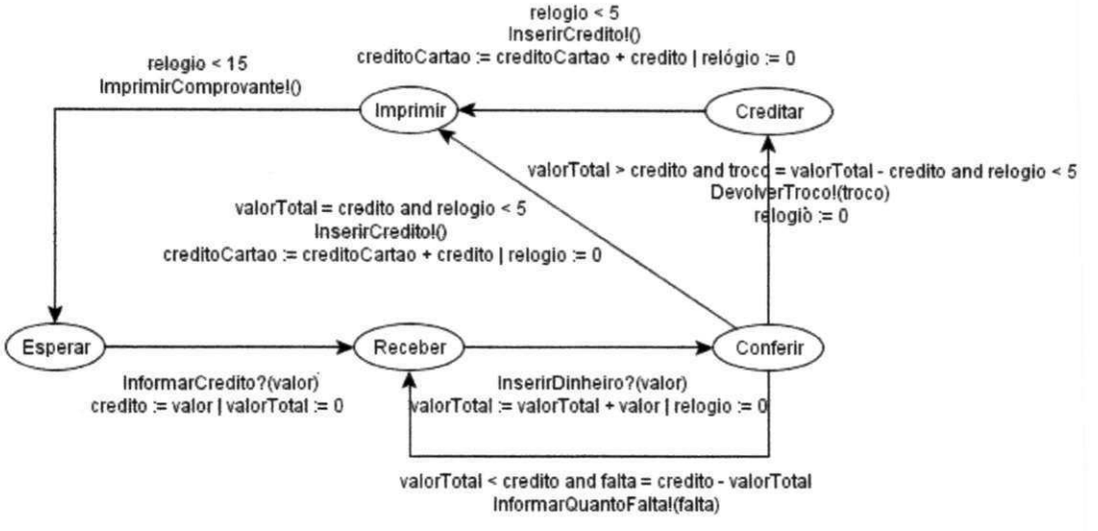


Figura 2.2: Exemplo de um TIOSTS que representa o comportamento de uma máquina para carregar crédito no cartão de passagem de um metrô.

Os nós *Esperar*, *Receber*, *Conferir*, *Creditar* e *Imprimir* representam as possíveis configurações que o sistema pode assumir durante sua execução. Ações de entrada são seguidas do símbolo '?' e ações de saída são seguidas do símbolo '!'. Esses símbolos são usados apenas como notação e não fazem parte do nome da ação.

Partindo do nó *Esperar*, o sistema vai para *Receber* quando o cliente informa o valor a ser creditado no cartão. Neste momento, a variável *credito* recebe o valor da variável *valor* fornecida como parâmetro da ação *InformarCredito* e a variável *valorTotal* é zerada. Em *Receber*, o sistema vai para *Conferir* quando o cliente insere moedas e/ou notas. Neste momento o sistema soma à variável *valorTotal* o valor monetário inserido. Além disso, o sistema também zera o *clock relogio*. Em *Conferir*, se o cliente tiver inserido um valor

inferior ao valor a ser creditado então o sistema volta para *Receber* e informa quanto falta para completar o valor a ser creditado através da ação *InformarQuantoFalta*. Em *Conferir*, caso o cliente tiver inserido o valor exato a ser creditado então o sistema vai para *Imprimir* e insere o crédito através da ação *InserirCredito* em menos de 5 unidades de tempo. Além disso, o novo valor de crédito do cartão, representado pela variável *creditoCartao* passará a ser o valor atual acrescido do valor da variável *credito* e a variável *relogio* é zerada. Ainda em *Conferir*, caso o cliente tenha inserido um valor superior ao valor a ser creditado então o sistema vai para *Creditar* e devolve o troco através da ação *DevolverTroco* e zera a variável *relogio*. A operação de devolver o troco não deve ultrapassar 5 unidades de tempo. Após devolvido o troco, o sistema atualiza o crédito em menos de 5 unidades de tempo e vai para *Imprimir*. Atualizado o crédito do cartão, o sistema imprime o comprovante através da ação *ImprimirComprovante* em menos de 15 unidades de tempo.

O TIOSTS pode ser formalmente expresso através da Definição 2.1.

Definição 2.1 (TIOSTS). *Um TIOSTS é uma óctupla* $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$, *onde:*

- V *é um conjunto finito de variáveis com tipos bem definidos;*
- P *é um conjunto finito de parâmetros de ações. Para* $x \in V \cup P$, $type(x)$ *denota o tipo de* x*;*
- Θ *é a condição inicial, um predicado composto de variáveis em* V*;*
- L *é um conjunto finito não vazio de nós;*
- $l^0 \in L$ *é o nó inicial;*
- $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ *é o alfabeto finito e não vazio formado pela união disjunta dos conjuntos* $\Sigma^?$ *de ações de entradas,* $\Sigma^!$ *de ações de saída e* Σ^τ *de ações internas. Cada ação* $a \in \Sigma$ *possui uma assinatura* $sig(a) = \langle p_1, ..., p_n \rangle$, *que é uma tupla de parâmetros distintos. A assinatura de ações internas é uma tupla vazia;*
- C *é um conjunto finitos de relógios;*
- $\mathcal{T}$ *é o conjunto de transições. Cada transição* $t \in \mathcal{T}$ *é uma sêxtupla* $\langle l, a, G, A, y, l' \rangle$, *onde:*

- $l \in L$ *é o nó origem da transição,*
- $a \in \Sigma$ *é a ação,*
- $G = G^D \wedge G^C$ *é a guarda da transição, onde* G^D *é um predicado com variáveis em* $V \cup sig(a)$[1] *e* G^C *é uma restrição sobre os relógios de* C *definida como uma conjunção de restrições na forma* $\alpha \# c$*, onde* $\alpha \in C$*,* c *é um valor constante do tipo inteiro e* $\# \in \{<, \leq, =, \geq, >\}$*,*
- $A = A^D \cup A^C$ *representa as atribuições da transição. Para cada variável* $x \in V$ *há exatamente uma atribuição em* A^D*, na forma* $x := A^{D^x}$*, onde* A^{D^x} *é uma expressão considerando* $V \cup sig(a)$*.* $A^C \subseteq C$ *é o conjunto de relógios a reinicializar,*
- $y \in \{lazy, delayable, eager\}$ *é o prazo limite da transição,*
- $l' \in L$ *é o nó de destino da transição.*

◇

2.3.2 Geração de casos de teste com o TIOSTS e propósito de teste

A geração de casos de teste para sistemas de tempo real que utiliza o TIOSTS como modelo formal está implementada na ferramenta SYMBOLRT [5]. A geração considera como critério de geração um propósito de teste que é descrito também em TIOSTS. O propósito de teste descreve explicitamente todos os cenários os quais o testador deseja gerar casos de teste e, portanto, se adequa à família de critérios da seção 2.4.5. Na Figura 2.3 é ilustrado um exemplo do propósito de teste do modelo da máquina de crédito da Figura 2.2. Nele, percebemos que é interesse do testador testar a situação em que o usuário insere exatamente o valor a ser creditado enquanto que não há interesse em testar a situação a qual o cliente fornece uma quantia superior ao valor a ser creditado; bem como também não há interesse na situação a qual o cliente fornece um valor inferior ao valor a ser creditado.

A geração de casos de teste envolve uma sequência de atividades bem definidas que recebe como entrada a especificação do STR modelado em TIOSTS e um propósito de teste descrito pelo mesmo formalismo. O resultado de cada atividade serve como entrada para

[1] Supõe-se que G^D é expressa em uma teoria na qual a satisfatibilidade é decidível.

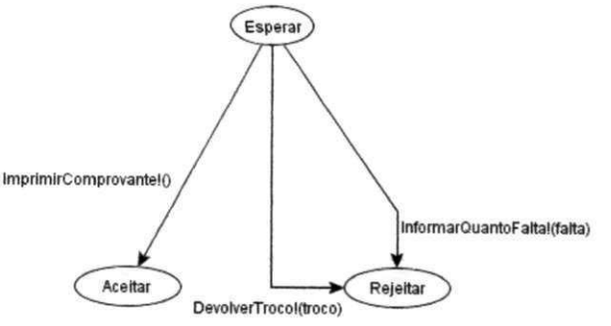


Figura 2.3: Propósito de Teste da Máquina de Comprar Crédito

a próxima até que, após a última atividade, tem-se como produto final os casos de teste conforme pode-se ver na Figura 2.4.

A seleção dos casos de teste é realizada através do produto síncrono. O produto síncrono permite obter um modelo que represente a execução paralela de dois outros modelos, porém sincronizando as partes que possuem ações compartilhadas permitindo identificar na especificação os comportamentos aceitos pelo propósito. Porém, para que essa atividade seja possível, é necessário que o propósito de teste esteja completo (i.e. para cada *location* deve ser possível habilitar qualquer ação do modelo). Assim, antes do produto síncrono é necessário tornar o propósito de teste completo. O processo de completude pode ser descrito através dos seguintes passos:

1. Em cada *location* é adicionado um autoloop para cada ação que esse *location* não possui;

2. Para cada transição com guarda G e ação a, cria-se uma nova transição que leva ao *location Reject* com a mesma ação e a negação da disjunção de todas as guardas associadas com a.

O resultado produzido por SYMBOLRT após a operação de completude do propósito de teste da Figura 2.3 pode ser visto na Figura 2.5. Percebe-se que ao fim da atividade em cada *location* há uma transição que pode ser habilitada para cada ação que contiver no propósito de teste tornando possível a operação de produto síncrono. Assim, o resultado da operação do produto síncrono do modelo da máquina de crédito da Figura 2.2 e o propósito de teste completo da Figura 2.5 pode ser visto na Figura 2.6.

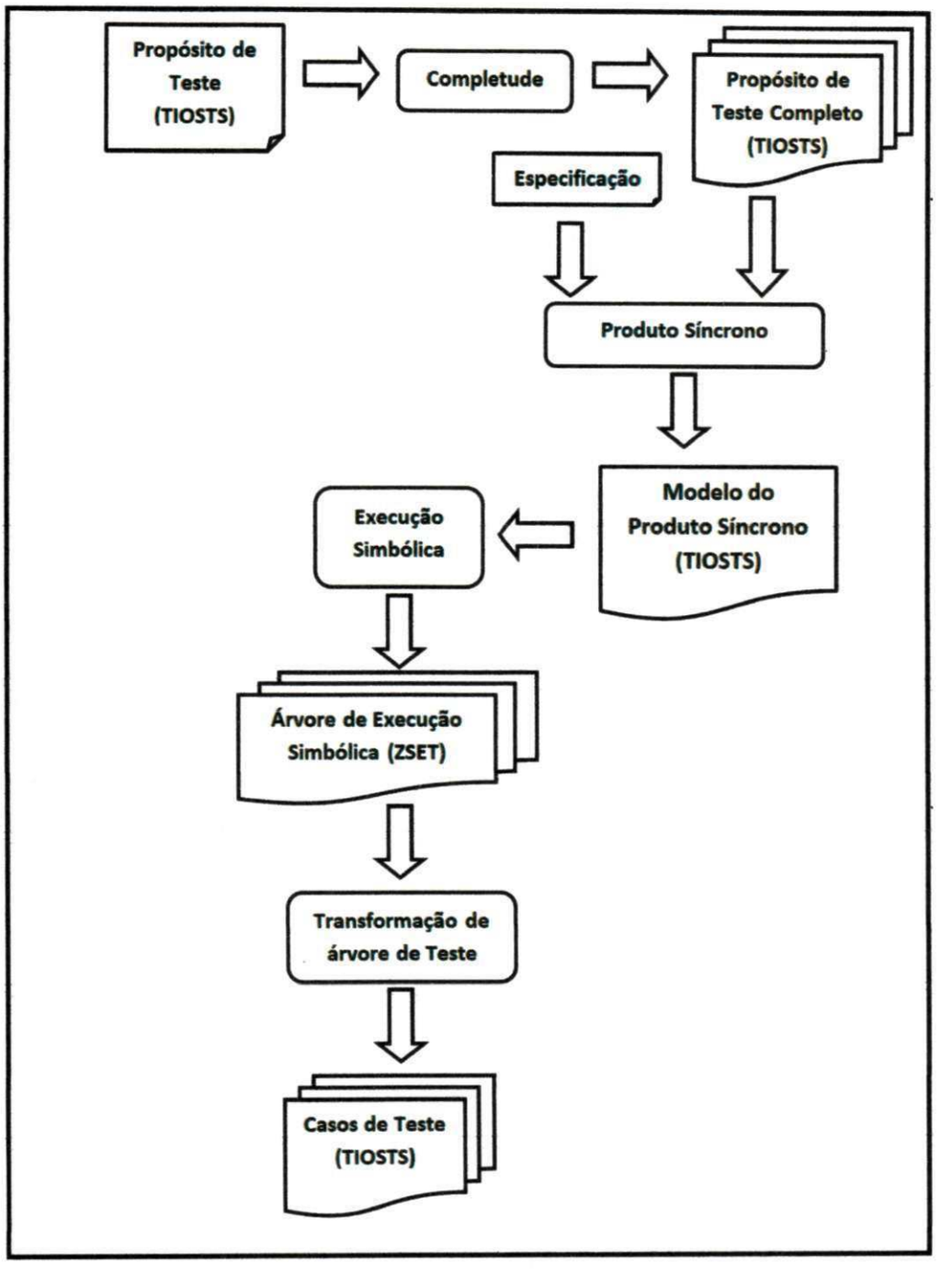


Figura 2.4: Atividades de Geração de Casos de Teste do SYMBOLRT

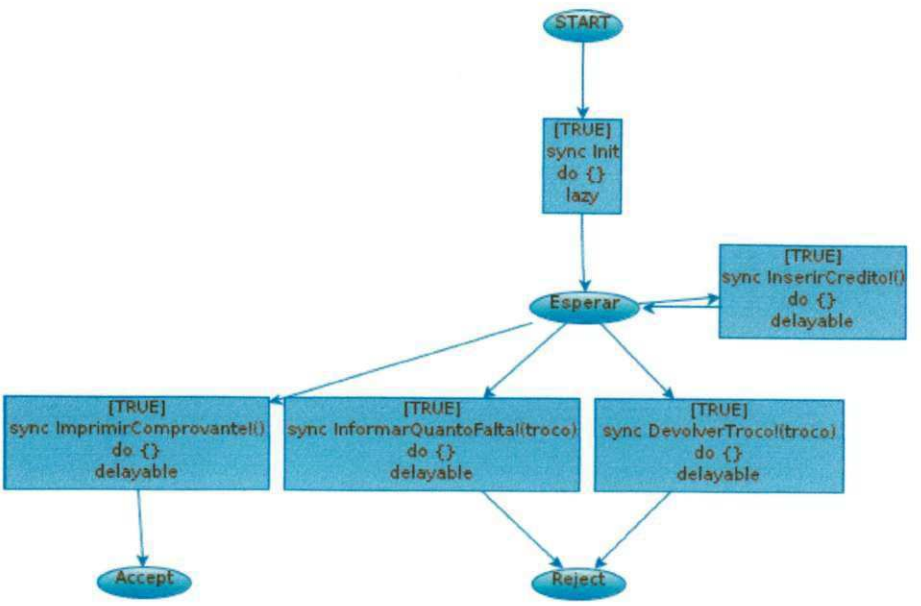


Figura 2.5: Propósito de Teste Completo

Após o produto síncrono, então o TIOSTS resultante é simbolicamente executado para identificar e selecionar possíveis traces que levam até o *location Accept*. A ideia principal é executar simbolicamente modelos TIOSTS usando a mesma técnica usada para execução simbólica de programas. Portanto, todos os traces possíveis são identificados usando valores simbólicos ao invés de valores concretos de cada parâmetro de ação e variáveis do modelo, evitando assim o problema da explosão do espaço de estados e identificando os estados alcançáveis. Os traces resultantes são representados como uma árvore de execução simbólica baseada em zonas [21], [27], (Zone-Based Symbolic Execution Tree). Uma zona representa o conjunto máximo de *clocks* que satisfazem uma restrição. Uma vez que foram identificados todos os possíveis traces pela execução simbólica o próximo passo é selecionar um caso de teste que leve ao estado *Accept*. Para isso é necessário selecionar uma sub árvore do ZSET gerado chamada de árvore de teste. Finalmente o trace selecionado é traduzido em casos de teste considerando a notação do TIOSTS. A Figura 2.7 ilustra o caso de teste resultante desse processo de geração de casos de teste.

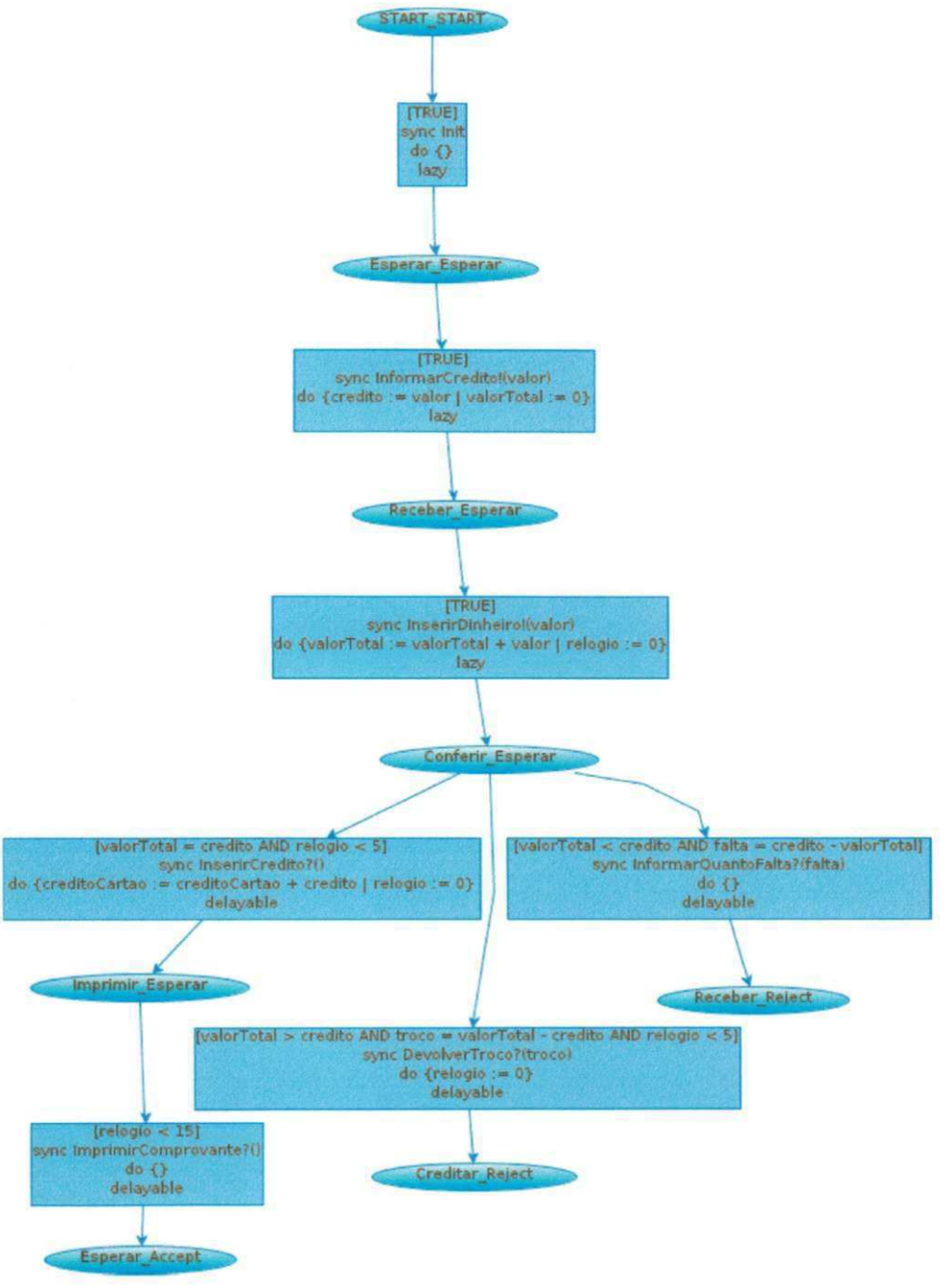


Figura 2.6: Exemplo de Produto Síncrono

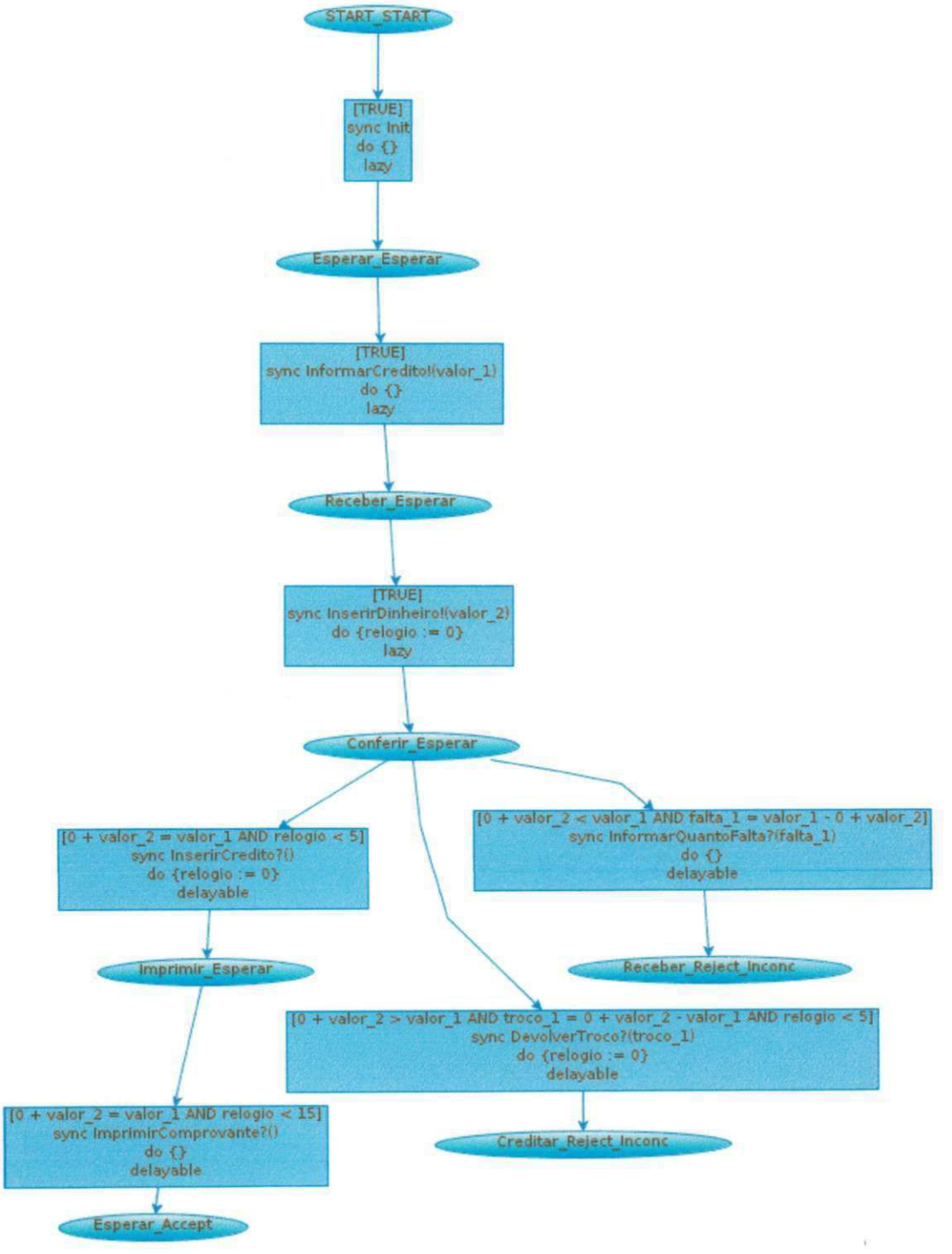


Figura 2.7: Caso de Teste

2.4 Critérios de Geração de Teste

Casos de teste em TBM são gerados segundo algum critério de geração. Eles determinam quais partes do sistema vão ser testadas, quantas vezes e sob quais circunstâncias serão testadas. Neste trabalho, o termo critério de geração será adotado, ao invés de seleção, visto que os critérios serão investigados no contexto da geração automática de casos de teste. A escolha do critério de geração influencia o algoritmo que as ferramentas usam para gerar testes que por sua vez influencia no tamanho do conjunto de casos de teste que será gerado, em quanto tempo leva para gerá-los, e quais partes do modelo serão testadas [42]. Por isso, a qualidade dos casos de teste gerados no tocante ao custo e capacidade de revelar falhas está diretamente relacionada ao critério de geração adotado. Dessa forma, uma boa ferramenta de geração de casos de teste deve dar suporte a vários tipos de critérios de geração de casos de teste.

Critérios de geração podem ser utilizados para dois objetivos principais [51]:

- **Medir a adequação do conjunto de testes:** Uma vez já gerado o conjunto de testes, é possível medir sua adequação ao nível de qualidade de teste exigido no desenvolvimento. Pode-se utilizar critérios de geração para determinar se o conjunto de testes previamente gerado cobre todos os elementos desejados pela equipe de teste.

- **Decidir quando parar de gerar teste:** O critério de geração pode servir como uma condição de parada na geração dos casos de teste, ou seja, testes são gerados até que uma dada condição seja satisfeita (e.g. gerar casos de teste até que todos os *branches* de um modelo sejam testados).

Segundo [51] um critério de geração pode ser classificado em uma das seguintes famílias:

- Critério de cobertura estrutural de modelo;
- Critério de cobertura de dados;
- Critério de modelo de falha;
- Critério baseado em requisitos;
- Especificação explícita de casos de teste;

- Métodos de geração de teste estatísticos.

Nas próximas seções, estas famílias serão descritas em mais detalhes.

2.4.1 Critério de cobertura estrutural de modelo

Refere-se à cobertura do controle de fluxo por meio do modelo, baseado nas ideias de controle de fluxo de programas. Dado que o modelo do sistema descreve seu comportamento, então cobrir elementos do modelo é importante para garantir que o teste inclua cenários desejados.

Algumas famílias de critérios de cobertura estrutural de modelo são oriundas de critérios de cobertura baseada em código e incluem controle de fluxo e controle de dados de um programa. Outras famílias de critérios de cobertura estrutural de modelo são derivadas de conceitos de teste baseado em modelos.

Critério de cobertura orientado a controle de fluxo

Critérios de cobertura orientados a controle de fluxo são originários de critérios de cobertura de código que por sua vez são baseados em declarações, decisões, caminhos, e loops no modelo.

Alguns exemplos de critérios de cobertura orientados a controle de fluxo são:

- *State coverage* [55]: O conjunto de casos de teste deve garantir que todos os estados possíveis do sistema serão testados.

- *All traces* [23]: Um trace é um conjunto alternado de nós e arestas consecutivas de um grafo. Sabendo disso, o conjunto de casos de teste gerado seguindo esse critério deve cobrir todos os traces possíveis do modelo.

- *All paths* [23]: Um caminho é um trace cujo nó inicial é o nó inicial do grafo. Sabendo disso, o conjunto de casos de teste gerado seguindo esse critério deve cobrir cada caminho possível do modelo.

Critério de cobertura orientado a fluxo de dados

Para esse tipo de critério, o modelo deve dar suporte à modelagem de definição e uso de variáveis. TIOSTS é um bom exemplo desse tipo de modelo. Nele, variáveis podem receber valores através de associações e serem utilizadas em guardas de transições. Critérios de fluxo de dados tentam cobrir todos os caminhos com definições e uso, todas as definições, ou todos os usos. Em TIOSTS, critérios de cobertura orientados a fluxo de dados tendem a cobrir transições com associações de variáveis, transições com guarda que utilizam variáveis, ou caminhos que partem de uma transição com definição de variável até uma transição com uso da mesma variável. Informalmente a definição de uma variável é a associação de um valor na variável enquanto que o uso é a leitura do valor de uma variável.

Alguns exemplos de critérios de cobertura orientados a fluxo de dados são:

- *All defs* [28]: Esse critério é satisfeito se para cada definição há pelo menos um caso de teste que leve ao seu uso.
- *Definition-use pair coverage* [28]: Esse critério requer que um conjunto de testes inclua todos os caminhos da definição de uma variável x para todas as transições alcançáveis onde x é usada.
- *All-p-uses* [28]: Similar ao Definition-use pair coverage, mas o uso da variável deve ser em um predicado.
- *All-c-uses* [28]: Similar ao Definition-use pair coverage, mas o uso da variável deve ser uma associação.

2.4.2 Critério de cobertura de dados

Critérios de cobertura de dados levam em consideração os possíveis valores de entrada de uma operação, transição ou ação em um modelo. Dado que os possíveis valores a serem passados como entrada em testes pode ser um conjunto infinito e que entre esses valores muitos são funcionalmente equivalentes podendo resultar em casos de teste redundantes, critérios de cobertura de dados são úteis na escolha de um subconjunto desses valores de forma a gerar testes menos redundantes e mais efetivos.

Alguns exemplos de critérios de cobertura de dados são:

- *One-value* [51]: O critério requer que simplesmente um valor de um domínio escolhido seja testado.
- *Random-value coverage* [51]: O critério requer que um valor do domínio seja aleatoriamente escolhido e testado.
- *All-boundaries coverage* [51]: Para cada predicado que é aplicado, o critério requer que os valores nas bordas dos intervalos de valores delimitados pelos predicados sejam testados.

2.4.3 Critério de modelo de falha

Esse critério gera casos de teste que são direcionados a detectar certos tipos de falhas bem conhecidas. Assumindo que o sistema sob teste tem certos tipos similares de falhas, então o conjunto de testes provavelmente detectará essas falhas. Essa família de critérios é utilizada também em testes de aceitação no intuito de demostrar a ausência de certos tipos de falhas comuns no domínio da aplicação. Esses critérios são geralmente baseados em modelos de falhas específicos [42].

2.4.4 Critério baseado em requisitos

O objetivo do critério baseado em requisitos é garantir que todos os requisitos do sistema tenham sido testados. Visando assim, garantir que todos os requisitos tenham sido testados é o principal objetivo do processo de validação. Isso implica em que se o teste passa, então as funcionalidades dos requisitos foram corretamente incluídas no sistema.

2.4.5 Especificação explícita de casos de teste

Uma outra forma de gerar casos de teste é dizer explicitamente à ferramenta de geração quais partes do sistema deseja-se gerar testes. Por exemplo, pode-se restringir caminhos no modelo os quais devem ser gerados testes. A principal vantagem nessa forma de geração é que o testador tem controle extremo sobre os casos de teste gerados. A desvantagem é que o processo de determinar os testes a serem gerados pode ser muito trabalhoso, muito mais do

que escolher alguns critérios de cobertura estrutural do modelo e, além disso, a qualidade do teste também fica muito relacionada à experiência do testador.

Um termo muito conhecido em TBM que expressa à geração de casos de teste com especificação explicita é 'Propósito de Teste'. O propósito de teste descreve explicitamente quais partes do modelo deseja-se gerar casos de teste. Na geração de casos de teste da Seção 2.3.2 o propósito de teste é descrito em TIOSTS e é através dele que a ferramenta SYMBOLRT sabe quais partes do modelo serão gerados casos de teste.

2.4.6 Métodos de geração de teste estatísticos

Em teste baseado em modelos, geração estatística de teste é frequentemente utilizada para gerar sequências de teste dos modelos do ambiente porque é o ambiente que determina os padrões de utilização do SUT (*System Under Test*) [51]. Uma abordagem típica é utilizar cadeias de Markov para especificar perfis de uso esperado do SUT. Casos de teste são gerados através de caminhamento aleatório na cadeia de Markov, onde a escolha aleatória da próxima transição é feita utilizando distribuição de probabilidade da transição de saída. Isso significa que casos de teste com maior probabilidade provavelmente são gerados primeiro. Nessa abordagem, o modelo utilizado é uma representação do uso do sistema, não seu comportamento.

2.5 Considerações Finais

Este capítulo apresentou o embasamento teórico necessário ao entendimento deste trabalho. Foram apresentados os conceitos sobre teste de software ao nível de sistema, teste funcional e teste baseado em modelo. Além disso, foi descrito o que é um sistema de tempo real, quais suas peculiaridades e tipos. Também foram apresentados os conceitos relacionados a modelos simbólicos dando ênfase ao modelo TIOSTS e ao processo de geração de casos de teste através da ferramenta SYMBOLRT que utiliza o TIOSTS para descrever tanto o sistema sob teste quanto o propósito de teste. Por fim, foram apresentados os conceitos relacionados aos critérios de geração descrevendo de forma geral quais são os tipos existentes de critérios de geração.

Capítulo 3

Critérios de Geração de Casos de Teste

Este capítulo apresenta os critérios de geração de casos de teste levantados através da revisão sistemática do Apêndice A. Através dela, foram encontrados critérios para sistemas de tempo real bem como critérios para sistemas sem restrição de tempo, mas que podem ser utilizados em sistemas de tempo real. A Seção 3.1 apresenta de forma geral todos os critérios explicando em linhas gerais como é o conjunto de casos de teste gerado a partir de cada critério. Além disso, os critérios são organizados em uma hierarquia segundo a relação de inclusão de conjuntos de casos de teste conforme Nouaary et al. [24]. Ainda na Seção 3.1, é mostrado quais dos critérios identificados se enquadram no escopo desse trabalho justificando o motivo da exclusão de alguns deles do estudo experimental do Capítulo 5. Na Seção 3.2, são explicados em mais detalhes os critérios que foram objeto de investigação do estudo experimental do Capítulo 5. Para tal, é utilizado um exemplo genérico de um TIOSTS como modelo para geração dos casos de teste de cada critério com a finalidade de ilustrar e tornar mais claro o entendimento de cada critério através desse formalismo.

3.1 Critérios de Geração Encontrados

A escolha de quais casos de teste devem ser gerados para testar um SUT é um fator importante na etapa de teste. Ter um conjunto de teste grande não garante efetividade no descobrimento de falhas no sistema. Assim, conforme foi visto na Seção 2.4 do Capítulo 2, em TBM os critérios de geração influenciam os algoritmos que as ferramentas utilizam para extrair os casos de teste a partir do modelo do SUT. Portanto, a capacidade de revelar falhas do con-

junto de casos de teste gerados está relacionada ao critério de geração utilizado. Atualmente, há muitos critérios de geração clássicos que podem ser utilizados na geração de casos de teste de STRs, no entanto, não foram encontrados trabalhos que estudem a efetividade desses e de outros critérios em STRs. No intuito de levantar estudos sobre critérios de geração de casos de teste para sistemas de tempo real, foi realizada uma revisão sistemática que pode ser encontrada no Apêndice A. Através da revisão sistemática realizada neste trabalho foram identificados e organizados em hierarquia 30 critérios de geração os quais podem ser vistos na Figura 3.1.

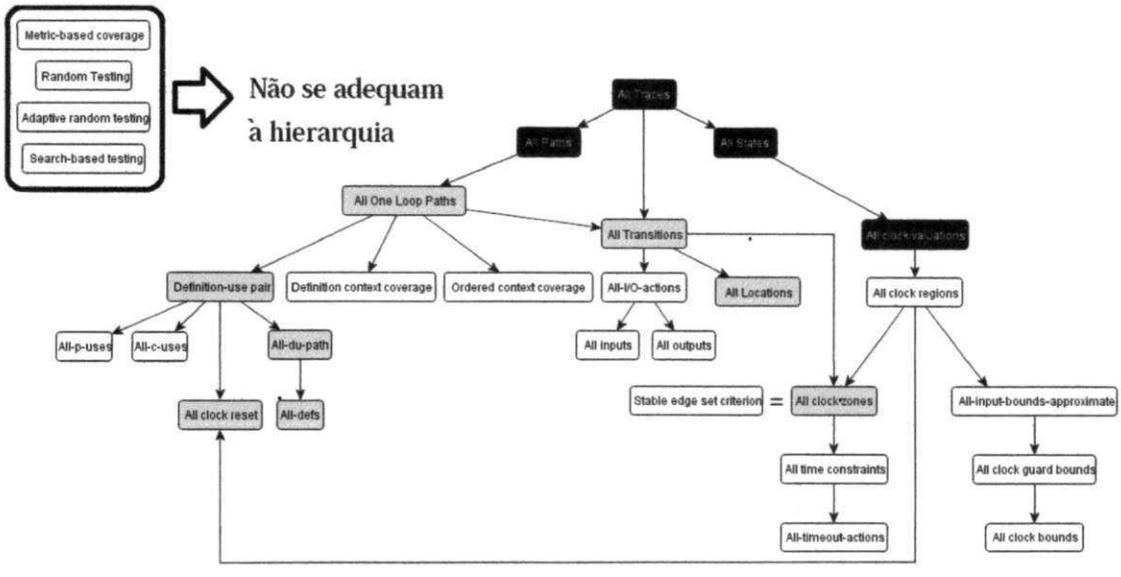


Figura 3.1: Hiearquia de Critérios

A Tabela 3.1 apresenta uma descrição resumida de cada um dos critérios da Figura 3.1.

Critério	**Descrição**
All Locations [23], [36]	Este critério é satisfeito se cada *location* do modelo é visitado por pelo menos um caso de teste.
All Transitions [23], [36]	Este critério é satisfeito se cada transição do modelo é exercitada por pelo menos um caso de teste.
All States [55], [1], [23]	Este critério é satisfeito se cada estado do modelo é visitado por pelo menos um caso de teste.

Metric-based coverage [55], [1]	Este critério seleciona por similaridade casos de teste oriundos de geração de casos de teste de um grid automaton pelos critérios *all transitions* e *all states*.
All paths [23], [29], [28]	Este critério é satisfeito se cada caminho do modelo é visitado por pelo menos um caso de teste.
All one loop paths [10]	Este critério é satisfeito se cada caminho do modelo é visitado por pelo menos um caso de teste. Caso haja loops os caminhos dão apenas uma volta em cada loop.
All traces [23]	Este critério é satisfeito se cada trace do modelo é visitado por pelo menos um caso de teste.
All inputs [17], [23]	Este critério é satisfeito se cada ação de entrada do modelo é exercitada por pelo menos um caso de teste.
All outputs [17], [23]	Este critério é satisfeito se cada ação de saída do modelo é exercitada por pelo menos um caso de teste.
All clock valuations [23]	Este critério é satisfeito se cada valor de *clock* do modelo é exercitado por pelo menos um caso de teste.
All clock regions [23]	Este critério é satisfeito se cada região do modelo é visitado por pelo menos um caso de teste.
All clock zones [23], [49]	Este critério é satisfeito se cada zona do modelo é visitada por pelo menos um caso de teste.
All clock guard bounds [23], [24]	Este critério é satisfeito se cada borda de guarda de *clock* do modelo é exercitada por pelo menos um caso de teste.
All clock bounds [23]	Este critério é satisfeito se cada borda de *clock* do modelo é exercitada por pelo menos um caso de teste. A borda de um *clock* é o maior valor que este *clock* pode assumir considerando todas as guardas com restrições de tempo.
All time constraints [23]	Este critério é satisfeito se cada restrição de tempo do modelo é exercitada por pelo menos um caso de teste.
All clock reset [23]	Este critério é satisfeito se cada reset de *clock* do modelo é exercitado por pelo menos um caso de teste.

Random Testing [8]	Gera aleatoriamente valores de *timeouts* para transições dos casos de teste em UML/MARTE.
Adaptive random testing [8]	Gera valores aleatórios diferentes de *timeouts* para transições dos casos de teste em UML/MARTE. Estes valores diferem de um caso de teste para outro.
Search-based testing [8]	Usa algoritmos genéticos para gerar valores de *timeouts* para transições dos casos de teste em UML/MARTE.
Stable edge set criterion [44]	O critério de geração particiona o espaço de estados da especificação em classes de equivalência. Os estados (par consistindo de *locations* e valores de *clock*) do autômato são particionados de forma que dois valores de *clock* pertencem à mesma classe de equivalência se, e somente se, eles habilitam precisamente as mesmas arestas partindo do conjunto de estados em que o autômato está ocupando atualmente. Esse critério se assemelha ao *all-clock-zones*.
Definition-use pair coverage [29], [28]	Esse critério requer que um conjunto de testes inclua todos os caminhos da definição de uma variável x para todas as transições alcançáveis onde x é usada.
Context coverage [29], [28]	Um contexto de uma definição de variável é as arestas em que as variáveis usadas para a definição são definidas. Por exemplo, para a associação $x := y+z$ o contexto é (e_y, e_z) se y foi definido em e_y e z foi definido em e_z. O critério requer que um conjunto de teste inclua todos os caminhos tal que para toda definição de uma variável x, cada contexto diferente da definição é representada.
Ordered context coverage [29], [28]	Similar a context coverage, no entanto, as arestas no contexto são listadas na ordem de suas definições.
All defs [29], [28]	Esse critério é satisfeito se para cada definição há pelo menos um caso de teste que leve ao seu uso.

All-p-uses [29], [28]	Similar ao *Definition-use pair coverage*, mas o uso da variável deve ser em um predicado.
All-c-uses [29], [28]	Similar ao *Definition-use pair coverage*, mas o uso da variável deve ser uma associação.
All-du-path [29], [28]	Similar ao *Definition-use pair coverage*, mas para cada definição de uma variável é suficiente encontrar todos os caminhos para o primeiro uso da variável.
All-input-bounds-approximate [17]	O conjunto de testes satisfaz o critério *all-input-bounds-approximate* se os endpoints ou valores próximos dos endpoints são exercitados por pelo menos um caso de teste. *Endpoints* são os valores que estão nas bordas dos intervalos delimitados pelas guardas de tempo, ou seja, considerando a e b constantes tal que $a < b$ o intervalo $[a, b[$ teria como *endpoint* inferior o valor a e como *endpoint* superior b.
All-timeout-actions [17]	O conjunto de testes satisfaz o critério *all-timeout-actions* se o *timeout* de cada ação de entrada é testado por pelo menos um caso de teste.
All-I/O-actions [17], [36]	Este critério é satisfeito se cada ação de entrada e cada ação de saída do modelo é exercitada por pelo menos um caso de teste.

Tabela 3.1: Critérios de Geração para Sistemas de Tempo Real.

En-Nouaary [23] apresenta um conjunto de critérios de geração de casos de teste organizados segundo uma relação de inclusão de forma que se um critério c_1 inclui um critério c_2 então qualquer conjunto de casos de teste que satisfaz c_1 também satisfaz c_2. c_1 inclui estritamente c_2 de forma que há casos de teste em c_2 que não satisfazem c_1. A relação de inclusão é transitiva de forma que se o critério c_1 inclui o critério c_2 e, por sua vez, c_2 inclui c_3 então o critério c_1 inclui, por transitividade, o critério c_3. Nesse trabalho, chamamos de hierarquia de critérios o conjunto de critérios encontrados na revisão sistemática e organizados segundo

a relação de inclusão de En-Nouaary [23].

A Figura 3.1 ilustra a hierarquia de critérios resultante desse trabalho, o critério mais alto na hierarquia é o *All Traces*. Ele inclui o critério *All Paths* e o *All States*, ou seja, o conjunto de casos de teste que satisfaz *All Traces* também satisfaz *All Paths* e o mesmo podemos dizer para *All States*. O critério *All One Loop Paths* está abaixo de *All Paths* na hierarquia e, desta forma, podemos dizer que *All Paths* inclui *All One Loop Paths*. Como a relação de inclusão é transitiva, também podemos afirmar que *All Traces* inclui *All One Loop Paths* dado que *All Traces* inclui *All Paths* que por sua vez inclui *All One Loop Paths*. A mesma conclusão podemos tirar dos critérios *All Clock Valuations*, *All States* e *All Traces* bem como para todos demais critérios abaixo da hierarquia.

A hierarquia de critérios presente em [23] foi melhorada por este trabalho adicionando-se a ela novos critérios encontrados na revisão sistemática. Os critérios encontrados, além dos presentes na hierarquia de En-Nouaary [23], foram classificados segundo a relação de inclusão e a nova hierarquia de critérios resultante pode ser vista na Figura 3.1. No entanto, alguns dos critérios identificados não incluem nem são inclusos por nenhum outro critério da hierarquia da Figura 3.1 e, portanto, não se adequam à hierarquia de critérios sendo os critérios: *Metric-based coverage*, *Random Testing*, *Search-Based Testing* , *Adaptive Random Testing* e *Stable Edge Set Criterion*.

O critério *Metric-based coverage* utiliza um *grid automaton* para representar de forma finita o domínio infinito de valores que um *clock* pode assumir. O *grid automaton* é um autômato que representa os valores de *clock* através de valores escolhidos de regiões de *clock*. Informalmente, uma região de *clock* é um conjunto de valores de *clock* que possuem o mesmo valor semântico na especificação do sistema de tempo real. Assim, o critério utiliza similaridade para selecionar um subconjunto ótimo de casos de teste a partir dos casos de teste gerados do *grid automaton* pelos critérios *All Transitions* e *All States*. Além disso, Zheng et al. [55] utiliza o modelo TROM (Timed Reactive Object Model) para descrever os STRs, cuja descrição dos estados do sistema se dá nos *locations* do modelo, divergindo da forma como a maioria dos outros critérios da hierarquia aborda e, portanto, não há como incluir o critério *Metric-based coverage* na relação de inclusão da hierarquia. *Random Testing*, *Search-Based Testing* e *Adaptive Random Testing* são critérios de geração de valores de *timeouts* de UML/MARTE. UML/MARTE modela a passagem do tempo através do evento

after(T) em que o parâmetro T determina o *timeout* a ser esperado para que a transição seja ativada. Os valores que podem ser alocados a T são alocados a um vetor no caso de teste com tamanho l. Toda vez que a transição é ativada é feita a escolha de um valor do vetor. Dando, por exemplo, o valor l=2, nós podemos ter um vetor contendo, por exemplo, {0.4,0.32}. A primeira vez que a transição é ativada, o valor 0.4 é usado. Na segunda vez, o valor 0.32 é usado. Na terceira vez, o valor 0.4 é usado novamente e assim sucessivamente. *Random Testing* gera valores de *timeouts* de forma totalmente aleatória, *Adaptive Random Testing* gera valores aleatórios diferentes, e *Search-Based Testing* utiliza algoritmos genéticos para gerar valores de *timeouts*. Assim, nenhum desses critérios gera casos de teste que possam satisfazer os critérios da hierarquia de critérios. O critério *Stable Edge Set Criterion* possui a mesma semântica que o critério *all clock zones*.

Conforme mencionado no Capítulo 2, critérios de geração podem ser classificados em uma das seguintes famílias:

- Critério de cobertura estrutural de modelo;
- Critério de cobertura de dados;
- Critério de modelo de falha;
- Critério baseado em requisitos;
- Especificação explícita de casos de teste;
- Métodos de geração de teste estatísticos.

Segundo [25], as falhas mais comumente encontradas em sistemas de tempo real podem ser classificadas em falhas devido ao não cumprimento de restrições de tempo e falhas devido à mudança indevida de estado do sistema. O não cumprimento das restrições de tempo ocorre quando algum evento acontece antes ou após o tempo especificado enquanto que as falhas decorrentes de mudança de estado do sistema ocorrem quando o sistema entra em um estado o qual não foi especificado. Considerando o TIOSTS, o sistema não cumpre a restrição de tempo quando um evento (ou ação no modelo) ocorre fora da zona de tempo delimitada pela guarda da transição em que essa ação está especificada. Considerando que no TIOSTS o estado do sistema corresponde ao *location* e ao valor das variáveis (incluindo os *clocks*) que

o sistema assume em um dado momento então uma falha decorrente da mudança de estado do sistema ocorre quando os valores das variáveis do sistema não estão de acordo com os requisitos especificados no modelo. Analisando os critérios identificados pode-se perceber que realmente todos os critérios de geração ou são de cobertura estrutural de modelo ou são de cobertura de dados podendo ser classificados conforme a Tabela 3.2.

Cobertura Estrutural de Modelo	**Cobertura de dados**
All Locations, All Transitions, State coverage, Metric-based coverage, All Paths, All one loop paths, All traces, All inputs, All outputs, All clock regions, All clock zones, All time constraints, All clock reset, Stable edge set criterion, Weighted fault model, Definition-use pair coverage, Definition context coverage, Ordered context coverage, All defs, All-p-uses, All-c-uses, All-du-path, All-timeout-actions, All-I/O-actions.	*All clock valuations, All clock guard bounds, All clock bounds, Random Testing, Adaptive random testing, Search-based testing, All-input-bounds-approximate.*

Tabela 3.2: Classificação dos critérios de geração.

Considerando que para esse trabalho foi escolhido o TIOSTS para descrever sistemas de tempo real, alguns dos critérios da Figura 3.1 não são passíveis de serem utilizados como diretrizes de geração de casos de teste utilizando TIOSTS. Por isso, na Figura 3.1 os critérios foram destacados em 3 cores. Os critérios de cor preta são critérios cujo conjunto de casos de teste gerado em TIOSTS teria tamanho infinito e, portanto, não podem ser incorporados em uma ferramenta de automação de geração de casos de teste. Os critérios na cor cinza são critérios que possuem conjuntos de casos de teste de tamanho finito e que podem ser geradas a partir do modelo TIOSTS enquanto que os critérios na cor branca ou são equivalentes a algum critério na cor cinza ou não são compatíveis com o TIOSTS. Assim foram considerados para esse trabalho apenas os 8 critérios na cor cinza. A justificativa de cada critério ausente do estudo está organizada na Tabela 3.3.

Critério Ausente	**Motivo**
All Traces	Em um modelo TIOSTS pode haver infinitos traces. Além disso, não faz sentido começar um caso de teste de um estado que não seja o estado inicial. Portanto, não existe caso de teste para traces que não partem do nó inicial.
All Paths	Em um modelo TIOSTS pode haver infinitos caminhos.
All States	Em qualquer modelo TIOSTS, por mais simples que seja, existem infinitos estados.
All Clock Valuations	Em qualquer modelo TIOSTS existem infinitos valores possíveis de *clock*.
All Clock Regions	TIOSTS não dá suporte à modelagem de regiões.
All input bounds approximate	TIOSTS trata os dados de maneira simbólica, incluindo os valores de *clock*. Este trabalho se detém à etapa de geração de casos de teste. No processo de teste utilizando o TIOSTS, os valores de *clock* são avaliados no momento da execução do teste e, portanto, fica fora do escopo deste trabalho.
All clock guard bounds	TIOSTS trata os dados de maneira simbólica, incluindo os valores de *clock*. Este trabalho se detém à etapa de geração de casos de teste. No processo de teste utilizando o TIOSTS, os valores de *clock* são instanciados no momento da execução do teste e, portanto, fica fora do escopo deste trabalho.
All clock bounds	TIOSTS trata os dados de maneira simbólica, incluindo os valores de *clock*. Este trabalho se detém à etapa de geração de casos de teste. No processo de teste utilizando o TIOSTS, os valores de *clock* são instanciados no momento da execução do teste e, portanto, fica fora do escopo deste trabalho.

All time constraints	Em TIOSTS restrições de tempo são modeladas nas guardas das transições. As guardas de transições que envolvem tempo delimitam zonas. Assim, o critério *All time constraints* é equivalente ao *All clock zones* em se tratando da modelagem de STRs em TIOSTS.
All timeout actions	Este critério requer que cada transição que tenha restrição de tempo seja ativada no último instante em que a guarda ainda continue válida testando assim o *timeout* da transição. No entanto, TIOSTS não considera sistemas bloqueantes, ou seja, não trata situações com restrições de tempo para ações de entrada. Assim, esse critério não pode ser utilizado em TIOSTS.
Stable Edge Set Criterion	Estados em TIOSTS são determinados pelos *locations* e valores das variáveis que o sistema pode assumir em um dado momento. Stable Edge Set Criterion particiona os estados que o sistema pode assumir em partições de equivalência em que em cada partição de equivalência os estados do sistema são equivalentes, ou seja, podem habilitar as mesmas transições do modelo. Além disso, o critério considera o estado do sistema como o *location* e os valores de *clocks* que o sistema assume em um dado momento [44]. Assim o critério acaba sendo equivalente ao *All clock zones*, pois cada zona particiona os estados da mesma forma que as partições de equivalência de Stable Edge Set Criterion.
All I/O Actions	As transições do TIOSTS podem possuir ações de entrada, de saída ou internas. No entanto, o processo de teste que utiliza o TIOSTS (Seção 2.3.2) não considera ações internas. Assim, o critério *All I/O Actions* fica equivalente ao critério *All transitions*.

All inputs	Sistemas de tempo real são, em sua maioria, sistemas reativos. Sistemas reativos reagem a estímulos de entrada e, portanto, seus modelos possuem sequências de entradas e saídas. Apesar de ser possível criar modelos que contenham caminhos só com ações de entrada, isso não faz sentido no contexto dos sistemas reativos.
All outputs	São muito raras as situações em que existem caminhos no modelo compostos de apenas ações de saída. Assim, quase sempre esse critério equivale a *All transitions*.
Definition Context Coverage	Considerando C como sendo um *clock*. Como o tempo não pode ser controlado, não faz sentido expressões do tipo $C := A \pm B$ seja A e B um *clock*, uma constante, ou uma variável qualquer. A única operação possível sobre um *clock* é reinicia-lo. Como este trabalho explora apenas restrições que envolvam tempo, ou seja, não interessa avaliar em especial guardas que não envolvam tempo. Então esse critério não se adequa ao TIOSTS.
Ordered Context Coverage	Considerando C como sendo um *clock*. Em TIOSTS não faz sentido expressões do tipo $C := A \pm B$ seja A e B um *clock*, uma constante, ou uma variável qualquer. Como este trabalho explora apenas restrições que envolvam tempo, ou seja, não interessa avaliar em especial guardas que não envolvam tempo. Então esse critério não se adequa ao TIOSTS.

All c uses	Considerando C como sendo um *clock*. Em TIOSTS não faz sentido expressões do tipo $C := A \pm B$ seja A e B um *clock*, uma constante, ou uma variável qualquer. Como este trabalho explora apenas restrições que envolvam tempo, ou seja, não interessa avaliar em especial guardas que não envolvam tempo. Então esse critério não se adequa ao TIOSTS.
All p uses	Como em TIOSTS *clocks* são utilizados apenas em expressões predicativas, então esse critério é equivalente ao *Definition-Use Pair*.

Tabela 3.3: Critérios Ausentes do Estudo.

3.2 Critérios de Geração Estudados

De todos os critérios analisados na Seção 3.1, 8 foram objeto de estudo do estudo experimental do Capítulo 5 e, portanto, merecem ser mais detalhadamente explicados e analisados. Nas subseções seguintes será explicada em mais detalhes a geração de casos de teste para cada um dos 8 critérios que se adequam ao escopo desse trabalho. Para cada critério foram gerados os casos de teste utilizando o mesmo modelo da Figura 3.2. Os casos de teste obtidos em cada critério foram organizados em tabelas e estão ilustrados da mesma forma que é ilustrado o modelo da Figura 3.2.

3.2.1 *All One Loop Paths*

O critério *All One Loop Paths* [10] é uma variação do critério *All Paths*. Esse critério percorre o modelo de forma a gerar casos de teste para cada caminho possível do modelo. Casos de teste são extraídos através do algoritmo de busca em profundidade Deep First Search. Entende-se como caminho o conjunto alternado de *locations* e transições partindo do *location* inicial, portanto, esse critério gera casos de teste para cada caminho possível do *location* inicial até todos os *locations* subsequentes do modelo. Se em um modelo houver

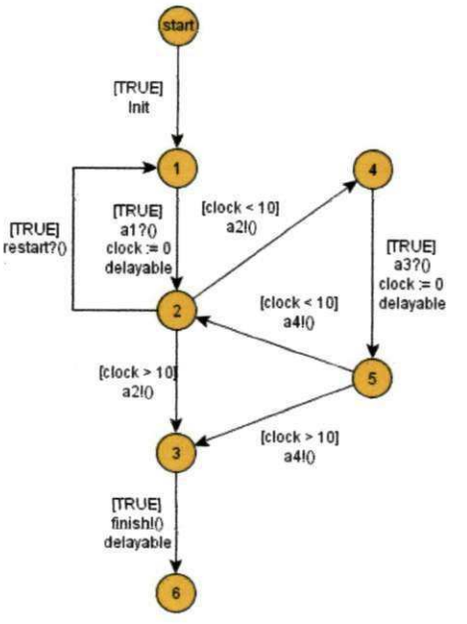
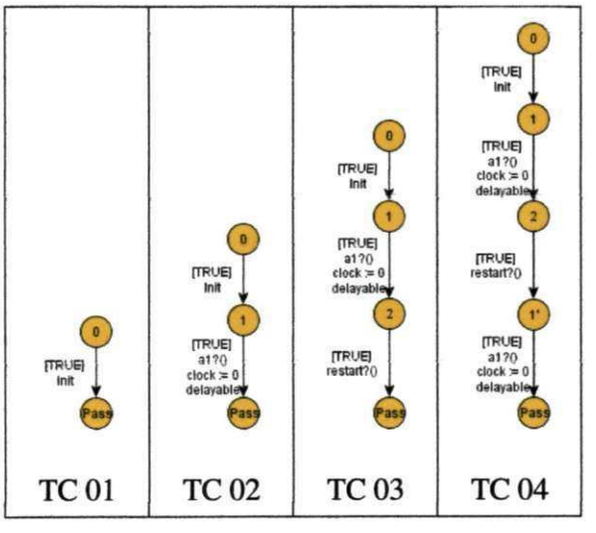
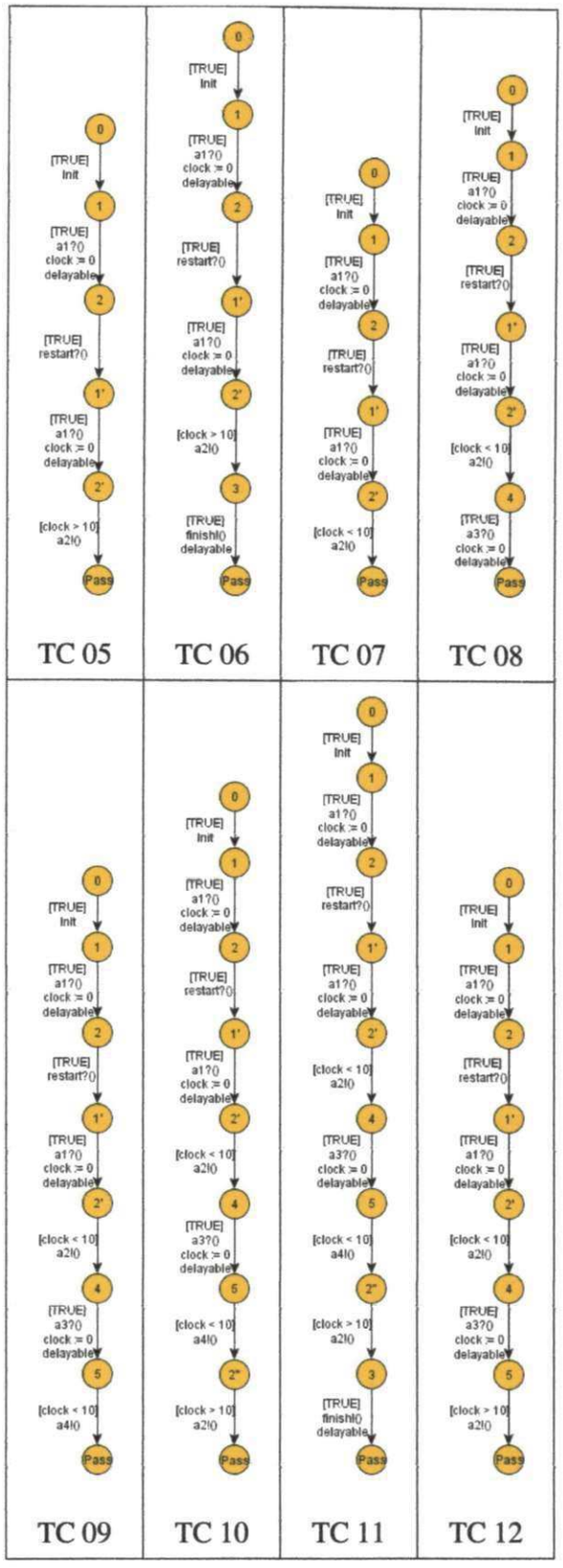
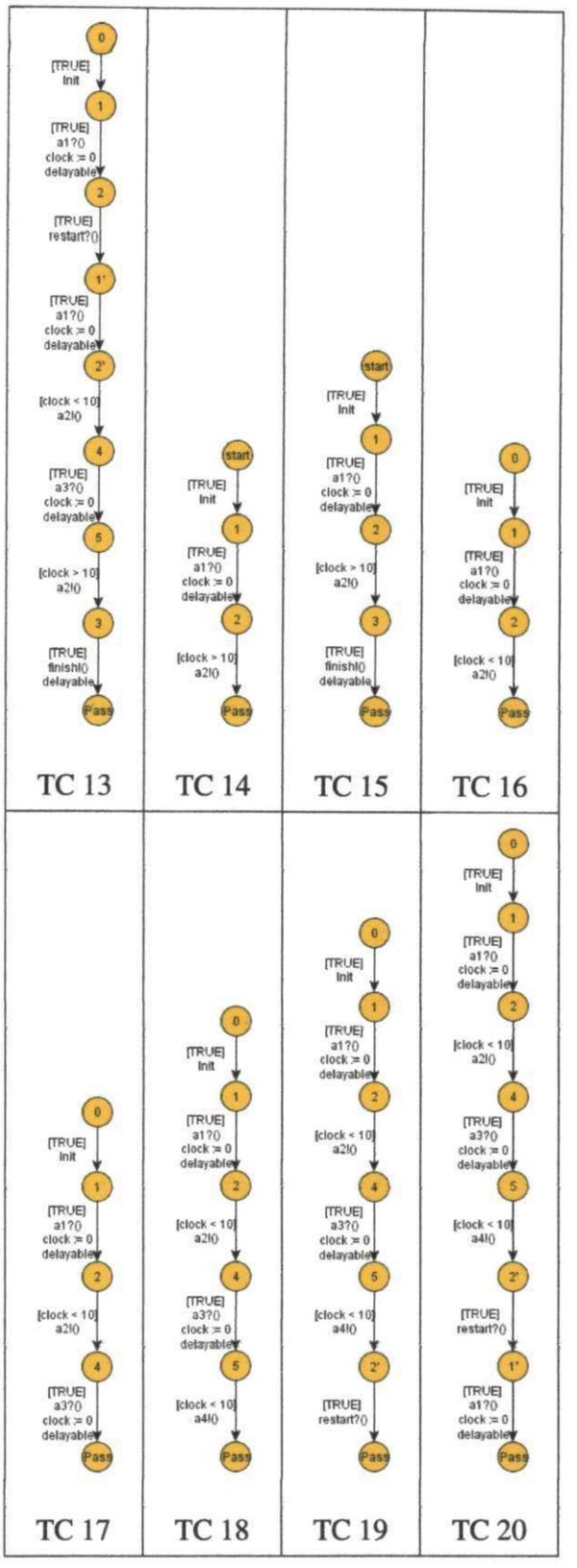


Figura 3.2: Exemplo Simples

algum loop então o conjunto de casos de teste desse modelo é infinito, pois podemos dar quantas voltas quisermos nesse loop. Assim, para evitar esse problema, o critério *All One Loop Paths* dá apenas uma volta em cada loop tornando finito o tamanho do conjunto de casos de teste gerado. Um possível conjunto de casos de teste gerado desse modelo seguindo o critério *All One Loop Paths* pode ser visto na Tabela 3.4.




TC 05
TC 06
TC 07
TC 08
TC 09
TC 10
TC 11
TC 12


0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[TRUE]
restart?()
1'
[TRUE]
a1?()
clock := 0
delayable
2'
[clock < 10]
a2!()
4
[TRUE]
a3?()
clock := 0
delayable
5
[clock > 10]
a2!()
3
[TRUE]
finish!()
delayable
Pass
TC 13
start
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock > 10]
a2!()
Pass
TC 14
start
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock > 10]
a2!()
3
[TRUE]
finish!()
delayable
Pass
TC 15
0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock < 10]
a2!()
Pass
TC 16
0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock < 10]
a2!()
4
[TRUE]
a3?()
clock := 0
delayable
5
[clock < 10]
a4!()
2'
[TRUE]
restart?()
1'
[TRUE]
a1?()
clock := 0
delayable
Pass
TC 17
0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock < 10]
a2!()
4
[TRUE]
a3?()
clock := 0
delayable
5
[clock < 10]
a4!()
Pass
TC 18
0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock < 10]
a2!()
4
[TRUE]
a3?()
clock := 0
delayable
5
[clock < 10]
a4!()
2'
[TRUE]
restart?()
Pass
TC 19
0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock < 10]
a2!()
4
[TRUE]
a3?()
clock := 0
delayable
5
[clock < 10]
a4!()
2'
[TRUE]
restart?()
1'
[TRUE]
a1?()
clock := 0
delayable
Pass
TC 20

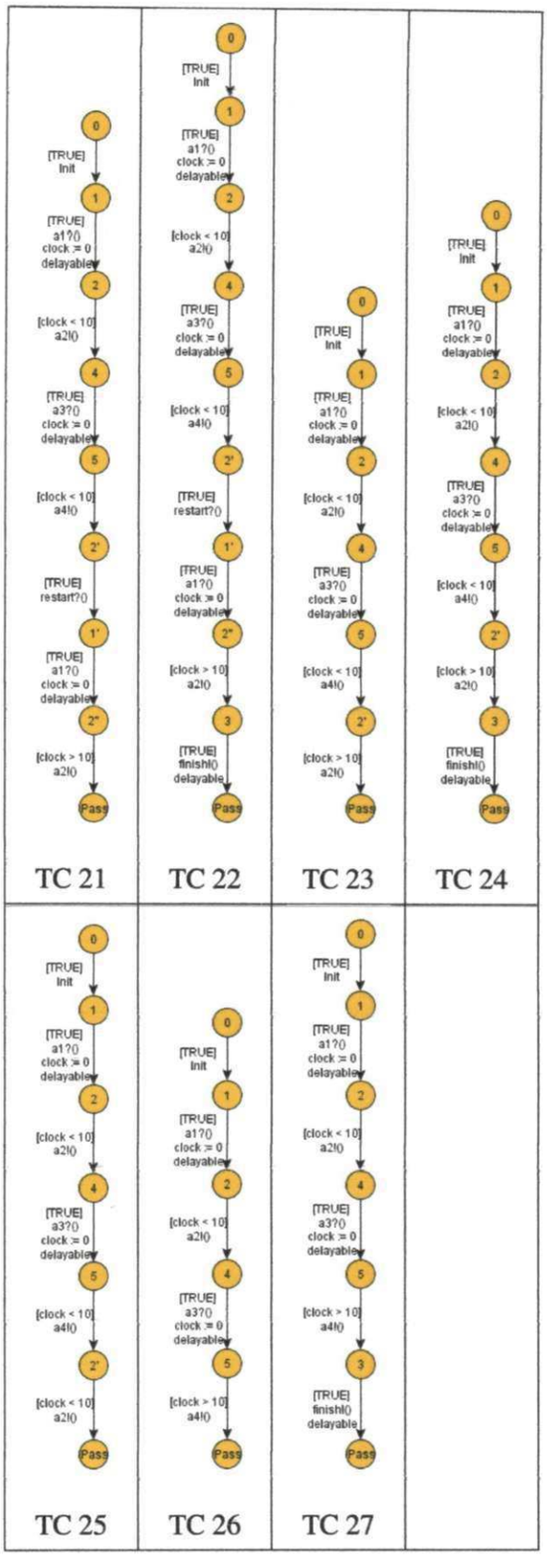

0
[TRUE]
Init
1
[TRUE]
a1?()
clock := 0
delayable
2
[clock < 10]
a2!()
4
[TRUE]
a3?()
clock := 0
delayable
5
[clock < 10]
a4!()
2'
[TRUE]
restart?()
1'
[TRUE]
a1?()
clock := 0
delayable
2"
[clock > 10]
a2!()
Pass
TC 21
TC 22
TC 23
TC 24
TC 25
TC 26
TC 27

Tabela 3.4: Conjunto de casos de Teste do Critério *All One Loop Paths*

3.2.2 *All Transitions*

O objetivo do critério *all transitions* é gerar casos de teste que cubram todas as transições do modelo, ou seja, cada transição do modelo deve estar presente em pelo menos um caso de teste do conjunto de casos de teste gerado. Para isso, o algoritmo que gera casos de teste seguindo o critério *all transitions* percorre o modelo seguindo o caminhamento de busca em profundidade e a cada caminho percorrido o modelo verifica se há alguma transição neste caminho que não tenha sido coberta por caminhos já gerados. Caso haja o novo caminho é incluído do conjunto de casos de teste. Caso contrário o novo caminho é descartado. Na Tabela 3.5 é ilustrada o conjunto de casos de teste gerado seguindo o critério *all transitions*.

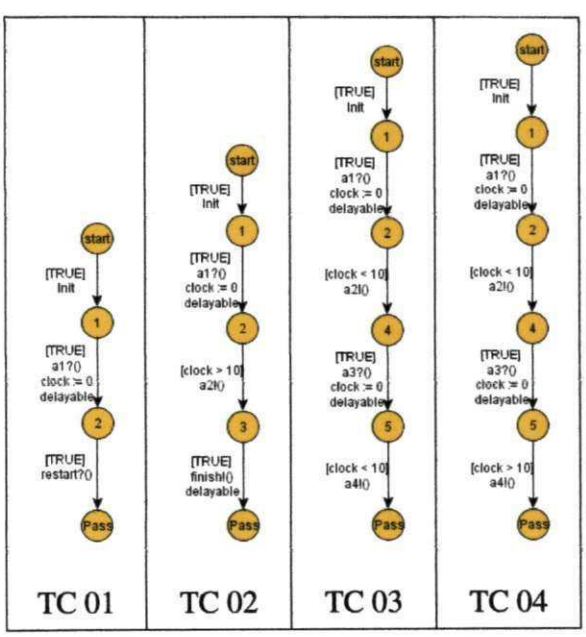


Tabela 3.5: Conjunto de casos de Teste do Critério *All Transitions*

3.2.3 *All Clock Zones*

A geração de casos de teste através do critério *all clock zones* segue basicamente a mesma metodologia do critério *all transitions* da subseção 3.2.2. O algoritmo de geração de casos de teste que segue esse critério também percorre o modelo através de busca em profundidade,

mas gera casos de teste para cada caminho do modelo de forma a cobrir todas as transições que possuam guardas com restrições de tempo gerando assim um subconjunto do conjunto de casos de teste de *all transitions* dado que o conjunto de transições com guardas que envolvem tempo é um subconjunto do conjunto de todas as transições presentes no modelo. A Tabela 3.6 ilustra o conjunto de casos de teste gerado a partir do critério *All Clock Zones*. Nela podemos ver claramente que o caso de teste TC 01 do conjunto de casos de teste de *All transitions* não está presente no conjunto de casos de teste de *all clock zones*, pois nele não há transições com restrições de tempo.

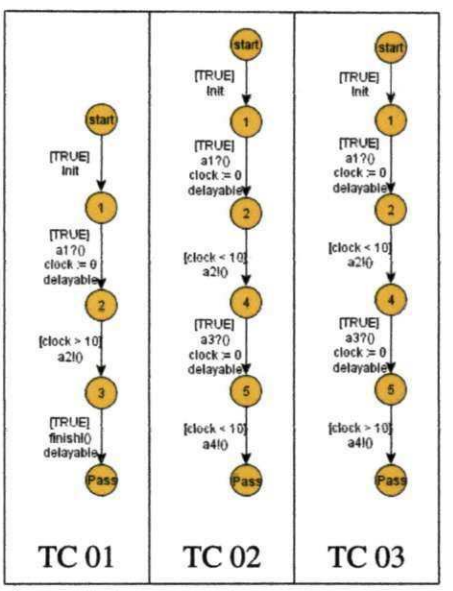


Tabela 3.6: Conjunto de casos de Teste do Critério *All Clock Zones*

3.2.4 *All Locations*

O critério de geração *all locations* também segue a mesma metodologia de *all transitions* e *all clock zones*, no entanto, o algoritmo de geração adiciona ao conjunto de casos de teste os casos de teste que cobrem de forma mínima os *locations* do modelo. Na Tabela 3.7 estão dispostos os casos de teste gerados do critério *all locations* do modelo da Figura 3.2. Podemos observar que nem todas as transições do modelo foram cobertas, mas todos os *locations* o foram. Além disso, podemos observar que os casos de teste realmente são um subconjunto dos casos de teste de *all transitions* especialmente o caso de teste TC 02 da Tabela 3.7 que é um caso de teste cujo caminho é uma parte do caminho do caso de teste TC

03 da Tabela 3.5.

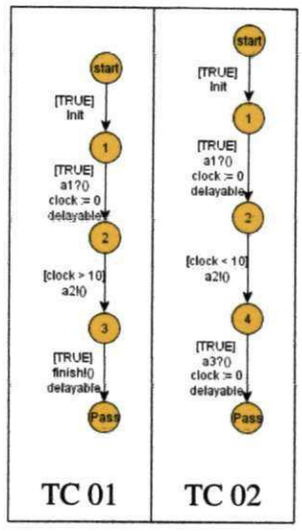


Tabela 3.7: Conjunto de casos de Teste do Critério *All Locations*

3.2.5 *Definition-Use Pair*

O critério *Definition-Use Pair* requer que um conjunto de casos de teste inclua todos os caminhos possíveis partindo de uma definição de uma variável até todas as transições alcançáveis em que essa variável seja usada. Para ficar mais claro considere o conjunto de casos de teste da Tabela 3.8 gerado por esse critério a partir do modelo da Figura 3.2. A variável *clock* recebe o valor zero na transição que parte do *location* 1 para o *location* 2 e é utilizada nas transições que partem do *location* 2 para o *location* 4, do *location* 5 para o *location* 2, do *location* 5 para o *location* 3 e do *location* 2 para o *location* 3. Portanto, para cada uma dessas transições que utilizam a variável *clock*, haverá um caso de teste cujo caminho passa pela definição até o uso dessa variável. No entanto, nesse exemplo também temos a definição da variável *clock* na transição que parte do *location* 4 para o *location* 5, portanto o critério também cobrirá os caminhos dessa definição até todas as transições com uso dessa variável.

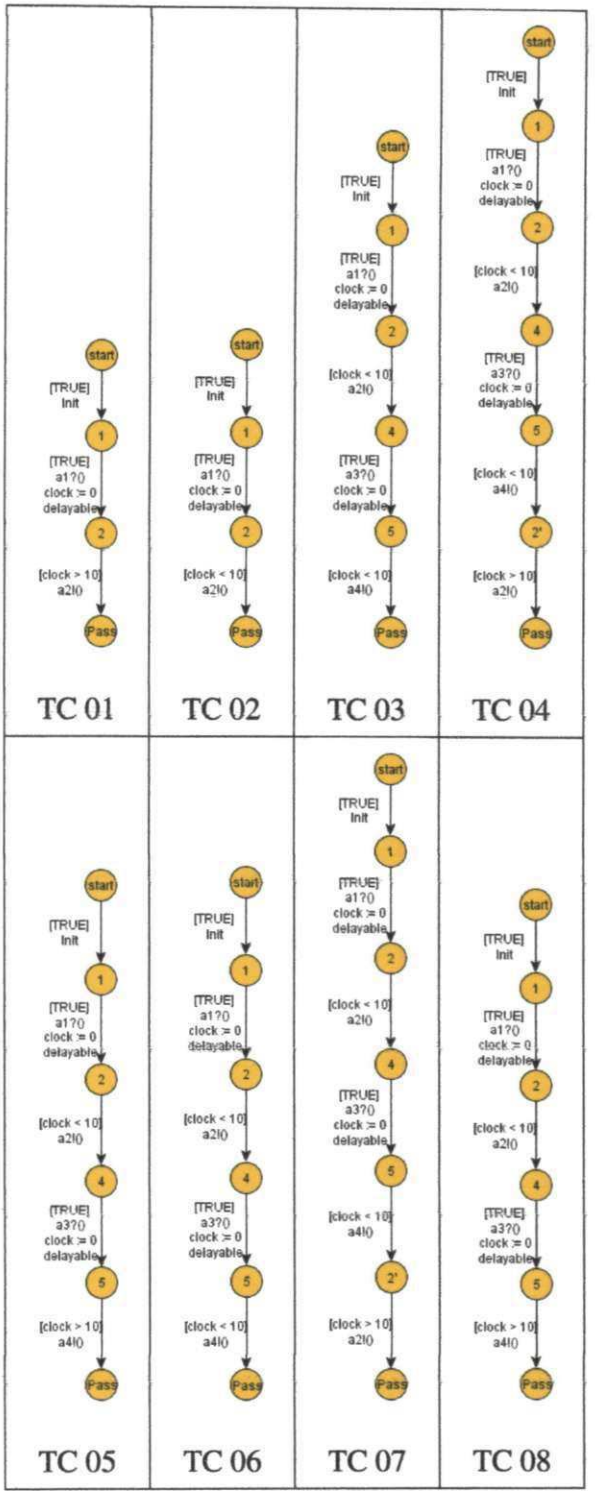


Tabela 3.8: Conjunto de Casos de Teste do Critério *Definition-Use Pair*

3.2.6 *All Du Paths*

O critério *all du paths* é um critério menos exaustivo do que o *definition-use pair*. Ele considera apenas os caminhos livres de definição, ou seja, apenas os caminhos de uma definição de um *clock* até o uso desse *clock* sem que no meio do caminho o *clock* seja definido novamente. Essa característica de cobrir apenas caminhos livres de definição elimina significativamente a redundância dos casos de teste como pode ser visto no conjunto de casos de teste da Tabela 3.9 gerada pelo critério *all du paths*.

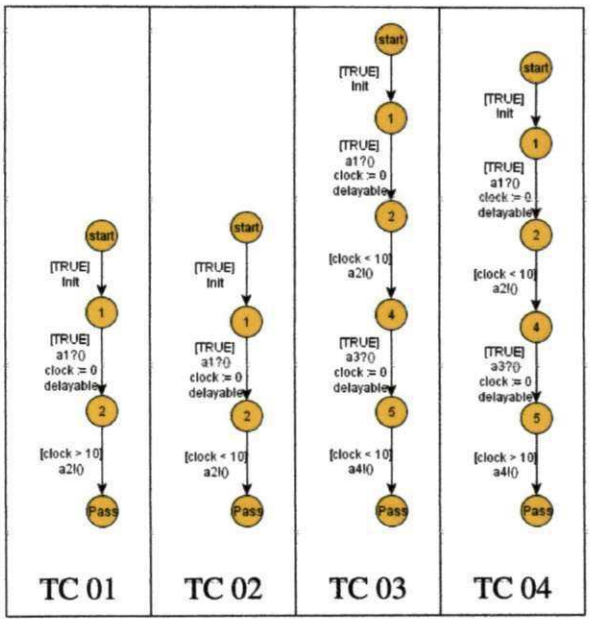


Tabela 3.9: Conjunto de Casos de Teste do Critério *All Du Paths*

3.2.7 *All Defs*

O critério *all defs* é ainda menos exaustivo que o critério *all du paths*. Nele é suficiente cobrir apenas um caminho da definição de cada variável para o seu uso. A Tabela 3.10 apresenta os casos de teste gerados do modelo da Figura 3.2 seguindo o critério *all defs*.

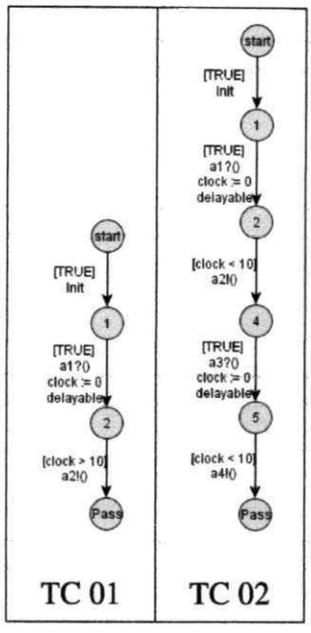


Tabela 3.10: Conjunto de Casos de Teste do Critério *All Defs*

3.2.8 *All Clock Resets*

All clock reset é o critério mais simples de todos os critérios implementados nesse trabalho. Nele, é suficiente incluir no conjunto de casos de teste os caminhos que cubram minimamente todas as transições que possuem reset de *clock*, ou seja, cada transição que possui reset de *clock* deve estar presente em pelo menos um caso de teste. Assim, os casos de teste do modelo da Figura 3.2 gerados seguindo o critério *All clock reset* podem ser vistos na Tabela 3.11.

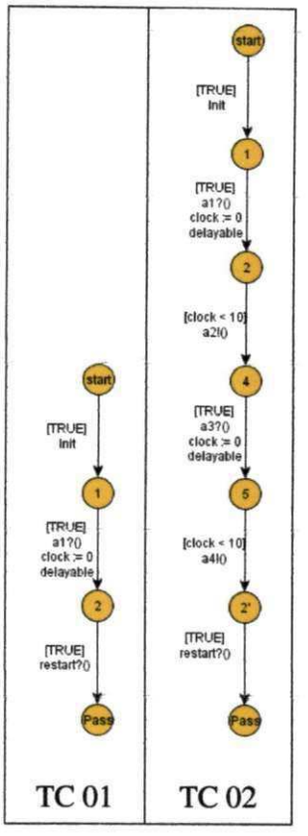


Tabela 3.11: Conjunto de Casos de Teste do Critério *All Clock Reset*

3.3 Considerações Finais

Este capítulo apresentou todos os critérios para sistemas de tempo real identificados através da revisão sistemática do Apêndice A. Todos os critérios foram organizados segundo uma hierarquia de inclusão de forma que o critério mais acima da hierarquia possui um conjunto de casos de teste que contém todos os casos de teste dos critérios abaixo da hierarquia. Também se pôde analisar que todos os critérios de geração para sistemas de tempo real ou são critérios de cobertura estrutural de modelo ou são critérios de cobertura de dados. Como o foco deste trabalho é no modelo TIOSTS então foram selecionados os critérios que se adequem a esse formalismo e então foram gerados os casos de teste em TIOSTS para cada critério selecionado.

Capítulo 4

Geração de Casos de Teste Baseada em Critérios de Geração

Este capítulo apresenta a extensão obtida no processo de teste de [4] e ilustra através de um exemplo como ocorre a geração de casos de teste baseada em critérios de geração. O capítulo também apresenta a extensão da ferramenta SYMBOLRT [5] para dar suporte ao novo processo de teste descrevendo sua nova estrutura e explica em mais detalhes através de um exemplo como os algoritmos do Apêndice B, implementados para cada um dos critérios estudados na Seção 3.2, caminham no modelo do SUT para com isso extrair os seus casos de teste.

4.1 Processo de Teste Usando os critérios

A Seção 2.3.2 do Capítulo 2 apresentou o processo de geração de casos de teste utilizando o TIOSTS para descrever tanto o sistema a ser testado quanto um propósito de teste. Nesta seção será apresentado o avanço realizado por este trabalho em relação ao processo de geração de teste da ferramenta SYMBOLRT [5].

Originalmente, SYMBOLRT dava suporte apenas à geração de casos de teste guiada por propósito de teste. A Figura 4.1 representa um panorama de como ocorre a geração de casos de teste na ferramenta SYMBOLRT. Nela, as atividades iniciais do processo de geração guiada por propósito de teste estão destacadas na cor cinza. A atividade 'Execução Simbólica' e 'Transformação de árvore de Teste' também são atividades previamente implementadas em

SYMBOLRT. O presente trabalho proporcionou um avanço em SYMBOLRT e agora, além da geração de casos de teste baseada em propósito de teste, é possível também a geração de casos de teste baseada em critérios de geração. Da atividade 'Extração de Sequências de Teste' são extraídas sequências de teste a partir da especificação do sistema de tempo real e do critério de geração escolhido. Essa atividade está destacada na cor preta na Figura 4.1 e foi implementada durante o presente trabalho. A atividade 'Execução Simbólica' e 'Transformação de árvore de Teste' são utilizadas para extrair os casos de teste a partir do resultado da atividade de extração de sequências de teste.

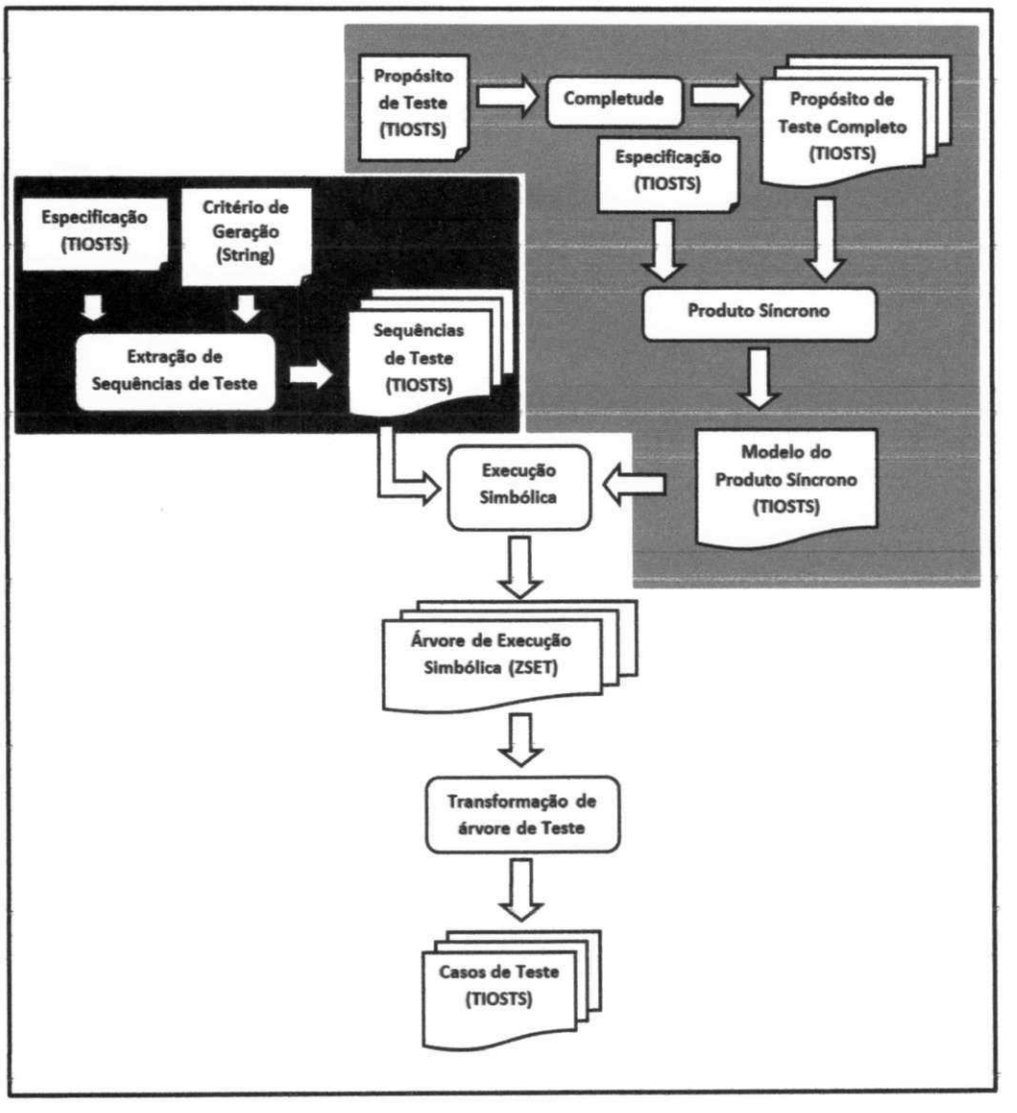


Figura 4.1: Atividades de Geração de Teste

A geração de casos de teste através de critérios de geração é realizada dando como en-

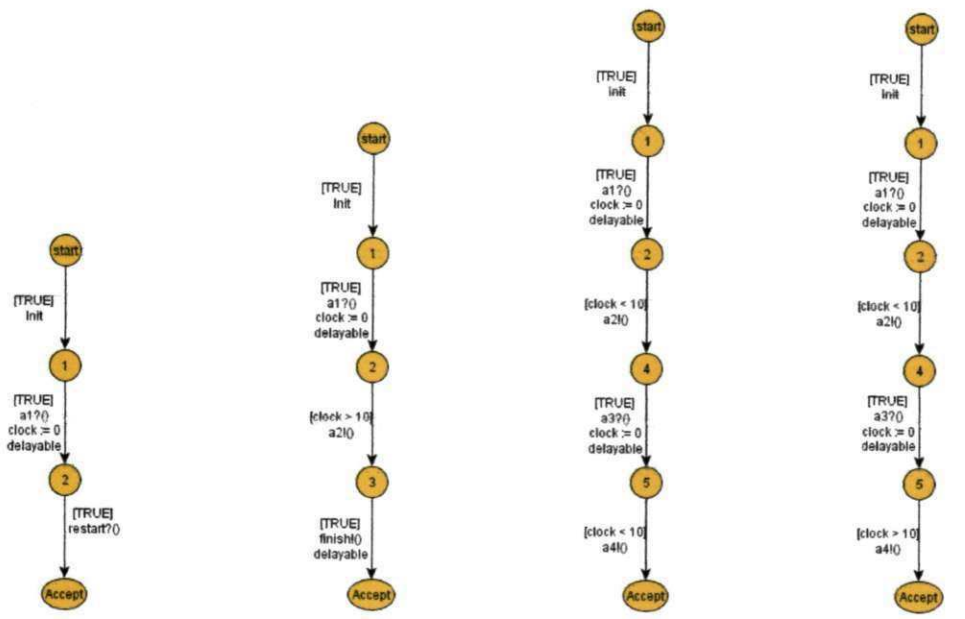
trada o modelo da aplicação a ser testada e o critério de geração que será utilizado para extração de casos de teste. A especificação é descrita em TIOSTS enquanto que o critério é descrito através de uma *string*, assim para gerar casos de teste segundo o critério *All Locations* é dado como entrada para o processo o modelo do SUT descrito em TIOSTS e a *string* *'All Locations'*. Existe um conjunto de palavras reservadas que determinam os oito critérios de geração de casos de teste sendo eles:

- *'All One Loop Paths'* para o critério *All One Loop Paths*;
- *'All Transitions'* para o critério *All Transitions*;
- *'All Locations'* para o critério *All Locations*;
- *'All Clock Zones'* para o critério *All Clock Zones*;
- *'All Clock Reset'* para o critério *All Clock Reset*;
- *'Definition Use Pair'* para o critério *Definition Use Pair*;
- *'All Du Paths'* para o critério *All Du Paths*;
- *'All Defs'* para o critério *All Defs*.

Definidas as entradas, a ferramenta percorre o modelo de acordo com a metodologia do critério de geração, ou seja, para o critério *all transitions* cobrirá todas as suas transições, para o critério *all locations* cobrirá todos os *locations*, para o critério *all one loop paths* cobrirá todos os caminhos possíveis dando no máximo uma volta em cada *loop*. A ferramenta extrai do modelo sequências de teste que representam os caminhos do modelo que satisfazem o critério de geração.

Para ilustrar melhor esta atividade considere o modelo da Figura 3.2 e o critério de teste *All Transitions*. O algoritmo de caminhamento no modelo do critério *all transitions* extrai caminhos novos do modelo enquanto houver caminhos com pelo menos uma transição do modelo não coberta. Assim, do modelo original da Figura 3.2, teremos as seguintes sequências de teste da Figura 4.2.

As sequências de teste são possivelmente os casos de teste do critério de geração utilizado. No entanto, podem haver estados inalcançáveis no modelo e sequências de teste com



(a) Sequência de Teste 1

(b) Sequência de Teste 2

(c) Sequência de Teste 3

(d) Sequência de Teste 4

Figura 4.2: Sequências de Teste

estados inalcançáveis não são passíveis de execução. A análise de quais sequências de teste podem vir a ser casos de teste é realizada através da execução simbólica. Então a execução simbólica é executada para cada sequência de teste. Como no exemplo da Figura 4.2 não há sequências de teste com estados inalcançáveis então estas sequências de teste formam os casos de teste do modelo da Figura 3.2 seguindo o critério *all transitions*. Porém, como o resultado da execução simbólica são os casos de teste modelados em ZSET então é necessária a execução da transformação de árvore de teste para cada sequência de teste em ZSET. Ao final desta operação, as sequências de teste resultantes serão os casos de teste finais.

4.2 Ferramenta

SYMBOLRT é uma ferramenta que automatiza todo o processo de geração de casos de teste para sistemas de tempo real descrito na Seção 2.3.2. A ferramenta foi estendida de forma a dar suporte à geração de casos de teste baseada em critérios de geração seguindo o processo

descrito na Seção 4.1 mantendo o processo de geração baseado em propósito de teste. As principais características da ferramenta são:

- Especificação, propósito de teste e critério de geração são especificados utilizando notação simples que esconde todos os detalhes teóricos do testador;
- Abstração de tempo e dados, evitando explosão de espaço de estados;
- Geração de casos de teste *off-line*;
- Modelagem e geração de casos de teste com interrupções;
- É um software de código aberto desenvolvido em Java e distribuído sobre a licença GNU GPL.

SYMBOLRT utiliza o modelo formal TIOSTS para descrever tanto a especificação do SUT quanto o propósito de teste. Assim, a ferramenta está limitada à capacidade de representação e à teoria fundada sobre TIOSTS.

São exemplos de algumas limitações do SYMBOLRT:

- Representar comportamentos concorrentes;
- Só considera modelos determinísticos;
- *Clocks*, variáveis e parâmetros só suportam tipos booleanos ou inteiros.

4.2.1 Arquitetura

A nova arquitetura da ferramenta é apresentada na Figura 4.3. Na nova arquitetura, há duas possibilidades de entrada para SYMBOLRT. Na primeira, a ferramenta recebe como entrada a especificação do SUT e o propósito de teste e assim gera casos de teste baseado em propósito seguindo o processo de geração da Seção 2.3.2. Na segunda, a ferramenta recebe como entrada a especificação e o critério de geração a ser utilizado. O formato de entrada adotado para a especificação e o propósito de teste é similar ao formato adotado pela ferramenta de geração automática de testes para sistemas reativos STG [16], porém estendida de forma a dar suporte a restrições de tempo. Já o critério de geração é determinado através de uma das palavras chaves descritas na Seção 4.1. O compilador é responsável por traduzir a entrada

para TIOSTS. Então, dependendo da entrada fornecida, o compilador executa a geração de casos de teste com propósito de teste ou com critérios de geração. Essa seção foca no avanço obtido na ferramenta SYMBOLRT e, portanto, se restringirá a discutir sobre a arquitetura referente à geração de casos de teste baseada em critérios de geração.

A próxima etapa do processo de teste descrito na Seção 4.1 é a extração de sequências de teste. Essa atividade é realizada pelo componente Decompositor. O Decompositor extrai do modelo sequências de teste que representam os caminhos do modelo que satisfazem o critério de geração. Por isso, estão implementados no Decompositor todos os critérios de geração de casos de teste investigados na Seção 3.2. Após a extração das sequências de teste, cada sequência de teste deve ser executada simbolicamente pelo componente de execução simbólica para determinar se os estados da sequência são alcançáveis. Como o resultado dessa operação é uma árvore de execução simbólica então o Seletor de Casos de Teste é executado para selecionar traces da árvore que levam ao cenário desejado. E, por fim, o componente de transformação de árvore de teste traduz a notação de árvore de execução simbólica para o TIOSTS. As três últimas atividades, execução simbólica, seleção de casos de teste e transformação da árvore de teste, são executadas para cada sequência de teste resultante do componente Decompositor, ou seja, se forem extraídas as três sequências de teste TS_1, TS_2 e TS_3 então o Decompositor passará primeiramente para o Executor Simbólico a sequência de teste TS_1, que por sua vez passará a árvore de execução simbólica resultante de TS_1 para o componente de transformação de árvore de teste que resultará em um caso de teste. Depois de obtido o caso de teste referente à sequência de teste TS_1, o Decompositor passará para o Executor Simbólico a sequência de teste TS_2 que fará o mesmo processo da sequência de teste anterior. Assim, o Decompositor só passará novamente para o Executor Simbólico a sequência de teste TS_3 quando o caso de teste referente à sequência de teste TS_2 estiver sido gerado.

4.2.2 Algoritmos

No Apêndice B é possível encontrar os códigos abstratos de todos os critérios de geração de casos de teste estudados na Seção 3.2. A exceção do critério *all clock zones* B.4, todos os critérios seguem o mesmo caminhamento no modelo diferenciando apenas a condição que determina se um dado caminho percorrido deve ou não fazer parte das sequências de teste

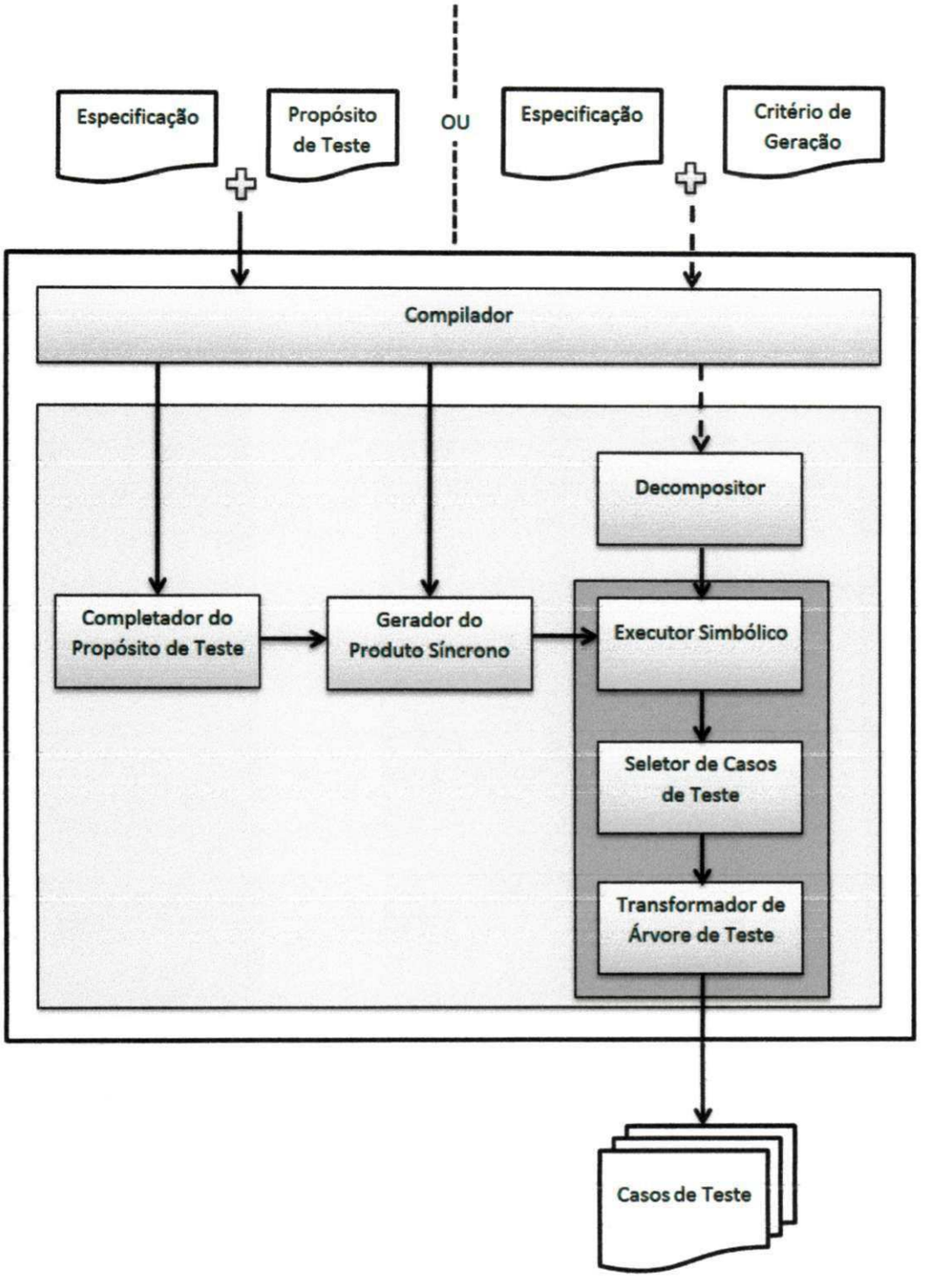


Figura 4.3: Nova Arquitetura da SYMBOLRT

resultantes do critério. No caso de *all clock zones*, a ferramenta executa primeiramente o critério *all transitions* B.2 e então exclui as sequências de teste que não possuem restrições de tempo.

Para entender melhor como são extraídas as sequências de teste do modelo, considere o código abstrato do critério *All One Loop Paths* 4.1. O algoritmo recebe como entrada o modelo da especificação e a partir dele extrai o *location* inicial (linha 3) que servirá como ponto de partida para o caminhamento no modelo. No SYMBOLRT é possível modelar interrupções em sistemas de tempo real utilizando a mesma técnica de Andrade et al. [7]. Interrupções são eventos assíncronos que podem ocorrer no sistema. Andrade et al. [7] modela o início de uma interrupção através de uma transição com ação Interrupt e parâmetro *intCode* cujo valor é único dentre todas as interrupções que o sistema pode ter e maior ou igual a zero. Após o tratamento de uma interrupção o sistema deve retomar ao mesmo ponto da execução normal do sistema quando foi interrompida. Essa característica é assegurada através da modelagem de uma transição que testa o valor da variável *choice* cujo valor deve ser o mesmo do valor do parâmetro *intCode* da interrupção ocorrida. Para manter a propriedade de geração de casos de teste para sistemas de tempo real com interrupção na ferramenta SYMBOLRT também foi necessário dar esse tratamento especial às variáveis *intCode* e *choice* nos algoritmos dos critérios de geração. Assim as variáveis *intCode* e *choice* (linhas 6 e 7) serão utilizadas para que, durante o caminhamento, o módulo Decompositor saiba quando estiver em um trace de uma interrupção bem como saber se o caminhamento já passou por uma interrupção. Assim, o método *allOneLoopPathsDecompose* do módulo Decomposer é chamado (linha 8) passando como parâmetros:

Código Fonte 4.1: Código Abstrato do Critério All One Loop Paths

```
1 Set<TIOSTS> getAllOneLoopPathsTestCases(TIOSTS model) {
2   Set<TIOSTS> testSequences = new Set<TIOSTS>();
3   Location initialLocation = model.getInitialLocation();
4   Set<Transition> path = new Set<Transition>();
5   Set<List<Transition>> paths = new Set<Set<Transition>>();
6   int intCode = -1;
7   int choice = 0;
8   Decomposer.allOneLoopPathsDecompose(testSequences,
9                                       initialLocation, path,
```

```
10                                          paths, intCode, choice);
11    return testSequences;
12  }
13
14  allOneLoopPathsDecompose(Set<TIOSTS> testSequences,
15                           Location location, Set<Transition> path,
16                           Set<Set<Transition>> paths,
17                           int intCode, int choice) {
18    if(!paths.contains(path)){
19      paths.add(path);
20      addPath(path, testSequences);
21    }
22    Set<Transition> outGoingTransitions = location.getOutTransitions();
23    for (Transition outGoingTransition : outGoingTransitions) {
24      if (!madeALoop(outGoingTransition, path, intCode)) {
25        if (isBeginInterruption(outGoingTransition)) {
26          if (choice == 0) {
27            intCode = getIntCode(outGoingTransition);
28            choice = intCode;
29          } else {
30            continue;
31          }
32        }
33        if (isEndInterruption(outGoingTransition)) {
34          if (intCode == getChoice(outGoingTransition)) {
35            intCode = -1;
36          } else {
37            continue;
38          }
39        }
40        path.add(outGoingTransition);
41        allOneLoopPathsDecompose(testSequences,
42                                 outGoingTransition.getTarget(),
43                                 path, paths, intCode, choice);
44      }
45    }
46  }
```

- O conjunto testSequences (até então vazio) de TIOSTS da linha 2 onde serão armazenadas as sequências de teste resultante do método *allOneLoopPathsDecompose*;
- O *location* inicial do modelo da especificação (linha 3) que servirá como ponto de partida para o caminhamento no modelo;
- O conjunto de transições path (linha 4) que servirá para o Decompositor armazenar o caminho percorrido durante a execução do algoritmo de caminhamento;
- O conjunto de conjuntos de transições paths (linha 5). Essa variável servirá para armazenar todos os caminhos percorridos no modelo pelo algoritmo de caminhamento do decompositor;
- A variável inteira *intCode* com valor inicial -1. O valor -1 indica que o nó em que o algoritmo de caminhamento está não pertence ao nó de um trace que faz parte de um tratamento de interrupção. Qualquer outro valor maior ou igual a zero indica que o *location* pertence a um trace que faz parte de um tratamento de interrupção;
- A variável inteira *choice* com valor inicial 0. O valor zero indica que o caminho percorrido no caminhamento não passou por um trace que faz parte do tratamento de uma interrupção. Qualquer outro valor maior que zero indica o contrário.

O método *allOneLoopPathsDecompose* percorre o modelo analisando todas as transições que saem do *location* (linhas 22 e 23) recebido como parâmetro na linha 15. Para cada transição que sai do *location* é verificado se ela já forma um loop (linha 24). Caso não forme o algoritmo segue adiante. Para toda nova transição avaliada é verificado se faz parte do começo de uma interrupção. Caso seja as variáveis *intCode* e *choice* recebem o valor do código da interrupção da transição. Se não fizer parte do começo de uma interrupção, é possível que faça parte do final de uma interrupção. Então é verificado se a transição faz parte do final de uma interrupção na (linha 33). Caso seja a variável *intCode* recebe o valor -1 (linha 35) sinalizando que o caminho saiu do tratamento da interrupção. Antes da chamada recursiva passando como novo *location* o *location* destino da transição avaliada (linha 42), a transição avaliada é armazenada no caminho percorrido. No início de cada chama recursiva do método *allOneLoopPathsDecompose*, é observado se o caminho até então percorrido já

faz parte dos caminhos percorridos. Caso não faça, adiciona-se esse caminho ao conjunto de caminhos e cria-se uma sequência de teste a partir desse caminho percorrido.

Os outros algoritmos percorrem o modelo da mesma forma que o *allOneLoopPaths*, divergindo apenas na condição que determina se um dado caminho percorrido deve ou não fazer parte das sequências de teste resultantes do critério. O critério *All One Loop Paths* é o critério mais exaustivo, ele percorre todos os possíveis caminhos que o modelo pode ter dando apenas uma volta em cada *loop*. Apesar de dar apenas uma volta em cada *loop*, o critério tem complexidade exponencial dado que cada *loop* aumenta a quantidade de casos de teste gerados em uma razão exponencial de base 2. Os critérios *Definition Use Pair* e *All Du Paths* também seguem a mesma complexidade de *All One Loop Paths*, no entanto, como não percorrem necessariamente todos os caminhos existentes então são menos exaustivos que o *All One Loop Paths*. Os critérios *All Transitions*, *All Locations*, *All Clock Zones*, *All Clock Reset* e *All Defs* por não percorrerem o modelo em busca de caminhos, então possuem complexidade menor que *All One Loop Paths*, *All Du Paths* e *Definition Use Pair* de forma que a quantidade de loops não influencia demasiadamente a quantidade de casos de teste gerados.

4.3 Considerações Finais

Neste capítulo foi apresentado o processo de geração de casos de teste baseado em critérios de geração. O processo de geração definido por Wilkerson de Andrade [4] foi estendido bem como a ferramenta SYMBOLRT de forma a permitir não somente a geração de casos de teste com propósito de teste, mas também a geração de casos de teste baseada em critérios de geração. Agora a ferramenta dá suporte a nove maneiras de geração de casos de teste, uma baseada em propósito de teste e oito baseada em critérios implementados cujos códigos abstratos podem ser encontrados no Apêndice B. A geração de casos de teste baseada em propósito e a geração de casos de teste baseada em critérios de geração são maneiras complementares de extrair casos de teste. Por um lado, a geração de casos de teste baseada em propósito é uma abordagem que fica muito dependente da experiência do testador. Assim, um testador com pouca experiência pode escolher cenários dos quais deseja extrair casos de teste que revelam poucas falhas. Além disso, a geração baseada por propósito de

teste tende a ser uma abordagem menos automatizada do que a geração baseada em critérios sendo, portanto, mais cara a geração de casos de teste baseada em propósito do que baseada por critérios de geração pois requer a análise de um testador, o qual vai escolher de quais cenários serão gerados casos de teste. A geração baseada em critérios de geração por ser uma abordagem mais automatizada requer um custo menor para geração. No entanto, por existir uma análise inicial de quais cenários são potencialmente mais defeituosos, a geração de casos de teste baseada por propósito, quando realizada por um testador experiente, tende a ser mais efetiva do que a geração de casos de teste baseada por critérios de geração. Por isso a importância de se manter em SYMBOLRT as duas formas de gerar casos de teste. A escolha de qual maneira utilizar vai depender da experiência do testador e do custo da geração que a ser tolerado.

Capítulo 5

Experimento

A Seção 3.2 apresentou oito critérios de geração de casos de teste para sistemas de tempo real resultantes da revisão sistemática do Apêndice A. No intuito de observar diferença entre os critérios em relação à capacidade de revelar falhas em sistemas de tempo real foi realizado um estudo experimental.

A prática de teste baseado em modelos já está inserida atualmente na indústria o que torna a busca por formas de gerar casos de teste que possam revelar mais falhas um problema de grande importância. Assim sendo, o experimento realizado retrata um problema real e que vem a contribuir significativamente na área de testes, mais precisamente em Testes Baseado em Modelos.

Este capítulo apresenta o experimento e os resultados obtidos de sua execução. Foi utilizado o *framework* proposto por Wohlin et al. [54]. A definição, planejamento, operação, análise e interpretação dos resultados do experimento são apresentados nas seções seguintes.

5.1 Definição

O primeiro passo é definir o experimento. Portanto, as principais questões propostas por Wohlin et al. [54] foram respondidas:

1. **O que está sendo estudado?** Critérios de geração de casos de teste para sistemas de tempo real;

2. **Qual é a intenção?** Investigar;

3. **Quais efeitos são estudados?** Capacidade de revelar falhas;

4. **Sob o ponto de vista de quem?** Testador;

5. **Qual é o contexto do estudo a ser conduzido?** Teste Baseado em Modelos (TBM).

A partir dessas respostas, a definição do objetivo pode ser escrita segundo o template GQM (*Goal Question Metric*) [52], [13] da Tabela 5.1.

Analisar critérios de geração de casos de teste para sistemas de tempo real
com a intenção de comparar conjuntos de casos de teste de STR
com respeito à quantidade de falhas descobertas e tamanho do conjunto gerado
do ponto do testador
no contexto de TBM

Tabela 5.1: Objetivo do experimento em GQM.

5.2 Planejamento

Uma vez que os elementos do experimento estão definidos, os passos seguintes do estudo devem ser planejados. Seguindo o *framework*, a seleção do contexto, as variáveis (dependentes e independentes), hipóteses, design e instrumentação foram definidas.

5.2.1 Seleção do Contexto

O experimento foi realizado em um laboratório situado na Universidade Federal de Campina Grande e não esteve envolvido com a indústria. Além disso, foi conduzido por um aluno de mestrado e supervisionado por dois professores doutores na área de teste.

Por se tratar de critérios de geração de casos de teste para sistemas de tempo real utilizando modelos baseados em TIOSTS esse experimento não é geral para qualquer modelo ou sistemas que não envolvam restrições de tempo.

5.2.2 Variáveis

Para caracterizar o experimento, as variáveis devem ser definidas. Nesse experimento são consideradas as seguintes variáveis dependentes e independentes:

Variáveis Independentes:

- Conjunto de critérios de geração selecionados na Seção 3.2;
- Modelos dos sistemas de tempo real do Apêndice C.

Variável Dependentes:

- Quantidade de falhas capturadas e o tamanho do conjunto de casos de teste.

5.2.3 Hipóteses

A intenção do experimento é analisar a capacidade de revelar falhas dos critérios de geração relacionando essa capacidade com o tamanho do conjunto de casos de teste gerado por cada critério de geração. A relação entre a capacidade de revelar falhas e o tamanho do conjunto de casos de teste gerado é expressa através da densidade. A densidade é a razão entre a quantidade de falhas reveladas e o tamanho do conjunto de casos de teste gerado. Acredita-se que os critérios de geração possuem densidades diferentes e, portanto, os critérios com maior densidade são mais efetivos em relação à capacidade de revelar falhas. Assim, foram formuladas a seguinte hipótese nula e alternativa deste experimento:

- H_0: Os critérios geram conjuntos de casos de teste que, em média, possuem a mesma densidade.
- H_1: Os critérios geram conjuntos de casos de teste que, em média, possuem densidades diferentes.

A hipótese nula descreve a situação a qual desejamos refutar neste experimento. Nela acredita-se que não há diferença entre as densidades dos critérios, ou seja, qualquer que seja o critério escolhido o mesmo será tão efetivo quanto qualquer outro. A hipótese alternativa é a negação da nula, ou seja, acredita-se que há diferença entre as densidades dos critérios e, portanto, há critérios mais efetivos que outros em relação à capacidade de revelar falhas.

5.2.4 Projeto do Experimento

Esse experimento consiste de um fator com oito níveis e seis repetições onde os oito níveis correspondem aos critérios investigados da Seção 3.2 e as seis repetições são os modelos elaborados a partir das três especificações de sistemas de tempo real utilizados na execução do experimento (Seção 5.3).

STRs, quanto ao grau de confiabilidade, podem ser classificados em *hard* ou em *soft*. STRs *hard* requerem um alto grau de confiabilidade enquanto que os *soft* aceitam um grau de confiabilidade menor. Esse experimento será conduzido para STRs *soft*. Por isso, será utilizado o nível de confiança de 95% e, portanto, nível de significância de 0,05. Além disso, segundo Jain [32] 0,05 é um nível de significância aceitável para comparações.

5.2.5 Instrumentação

O próximo passo do planejamento é especificar a instrumentação do experimento. Nesse passo, há três tipos de instrumentos [54]:

- **Objetos:** Para a execução do experimento foram utilizados três especificações de sistemas de tempo real:
  - **Sistema de Alarme:** O Sistema de Alarme é um sistema de monitoramento capaz de detectar a invasão de um intruso através de sensores de portas, janelas e movimento. O Sistema de Alarme possui um sistema de alimentação reserva permitindo operar mesmo quando há queda de energia elétrica. A Seção C.1 do Apêndice C apresenta uma explicação mais detalhada do sistema;
  - **Sistema de Ataque de um Caça:** O Sistema de Ataque de um Caça é um sistema que permite ao piloto de um caça realizar ataques a alvos terrestres. O sistema também permite a detecção de ameaças eminentes como ser alvo de um míssil ou ser alvo de um outro caça. A Seção C.2 do Apêndice C apresenta uma explicação mais detalhada do sistema;
  - **Protocolo de Áudio da Philips:** O Protocolo de Áudio da Philips [9] é um protocolo dedicado para trocas de mensagens de controle de dispositivos de áudio e

vídeo. Nele, os dados são codificados em Manchester e transferidos em um barramento único compartilhado. O protocolo também é capaz de detectar colisões no barramento. Além disso, o protocolo pode reiniciar o processo de envio de bits caso haja algum problema de envio. A Seção C.3 do Apêndice C apresenta uma explicação mais detalhada do sistema.

Destas especificações foram gerados seis modelos dos quais os casos de teste foram gerados:

- **Sistema de Alarme:** Foi modelado o Sistema de Alarme da Seção C.1 do Apêndice C desconsiderando o tratamento de queda de energia. Assim, o modelo ficou sem *loops* conforme pode ser visto na Figura 5.1;
- **Sistema de Alarme com Tratamento a Queda de Energia:** O Sistema de Alarme da Seção C.1 do Apêndice C foi modelado considerando o tratamento de queda de energia. A Figura 5.2 ilustra o modelo da especificação da Seção C.1. Com a intenção de diminuir o tamanho do modelo omitiu-se as guardas e as associações das transições que não pertencem ao tratamento da queda de energia pois são as mesmas do modelo da Figura 5.1;
- **Sistema de Ataque de um Caça:** Foi modelado o Sistema de Ataque de um Caça da Seção C.2 do Apêndice C desconsiderando o tratamento de ameaça de um ataque inimigo. A Figura 5.3 ilustra o modelo em questão;
- **Sistema de Ataque de um Caça com Tratamento a Ataque Inimigo:** O Sistema de Alarme de um Caça da Seção C.2 do Apêndice C foi modelado considerando o tratamento a ataque inimigo. A Figura 5.4 ilustra o modelo da especificação da Seção C.2. Assim como o a Figura 5.2, as guardas e associações das transições que não pertencem ao tratamento de ataque inimigo foram omitidas para simplificar o modelo;
- **Protocolo de Áudio da Philips:** Foi modelado o Protocolo de Áudio da Philips da Seção C.3 do Apêndice C desconsiderando a reinicialização do protocolo no caso de haver algum problema no envio de bits. A Figura 5.5 ilustra o modelo em questão;

– **Protocolo de Áudio da Philips com Tratamento a Erros:** O Protocolo de Áudio da Philips da Seção C.3 do Apêndice C foi modelado considerando o tratamento a erros de envio de bits. A Figura 5.6 ilustra o modelo da especificação da Seção C.3.

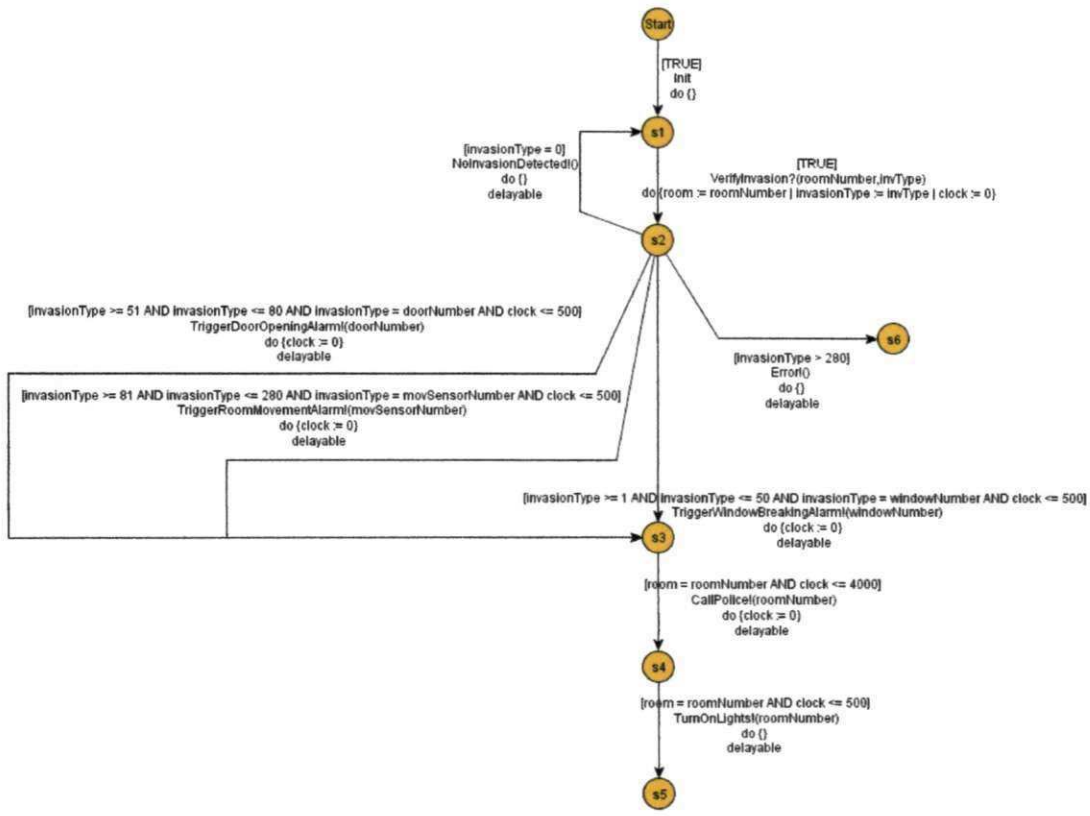


Figura 5.1: Modelo do Sistema de Alarme

- **Diretrizes:** Não haverá diretrizes já que não há sujeitos.
- **Instrumentos de medição:** a medição será realizada através da observação de falhas identificadas nos conjuntos de casos de teste gerados pelos critérios de geração bem como o tamanho do conjunto de casos de teste gerado.

5.2.6 Avaliação de Validade

Com a finalidade de identificar ameaças à validade do experimento e diminuir suas influências, algumas ameaças foram elencadas e classificadas em ameaças à validade de conclusão,

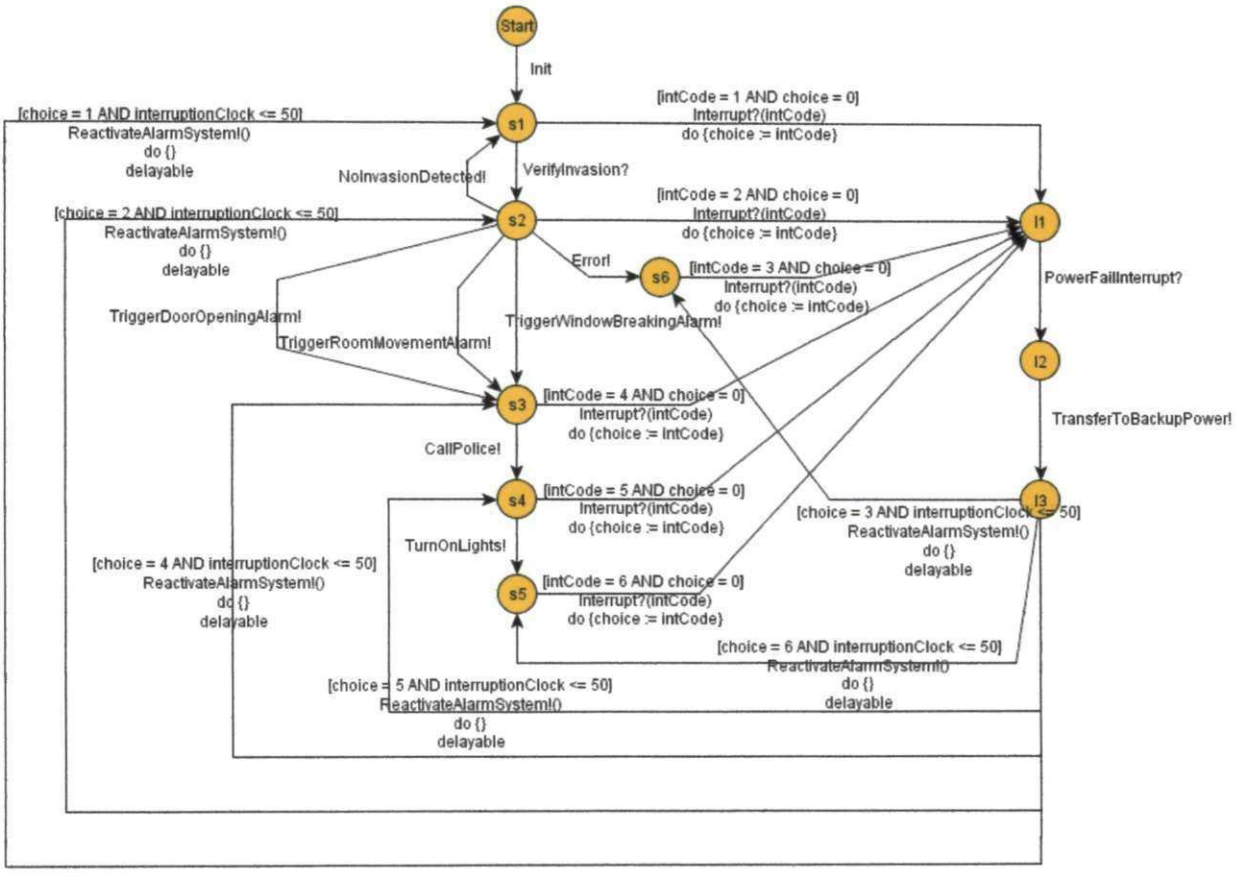


Figura 5.2: Modelo do Sistema de Alarme com Tratamento a Queda de Energia

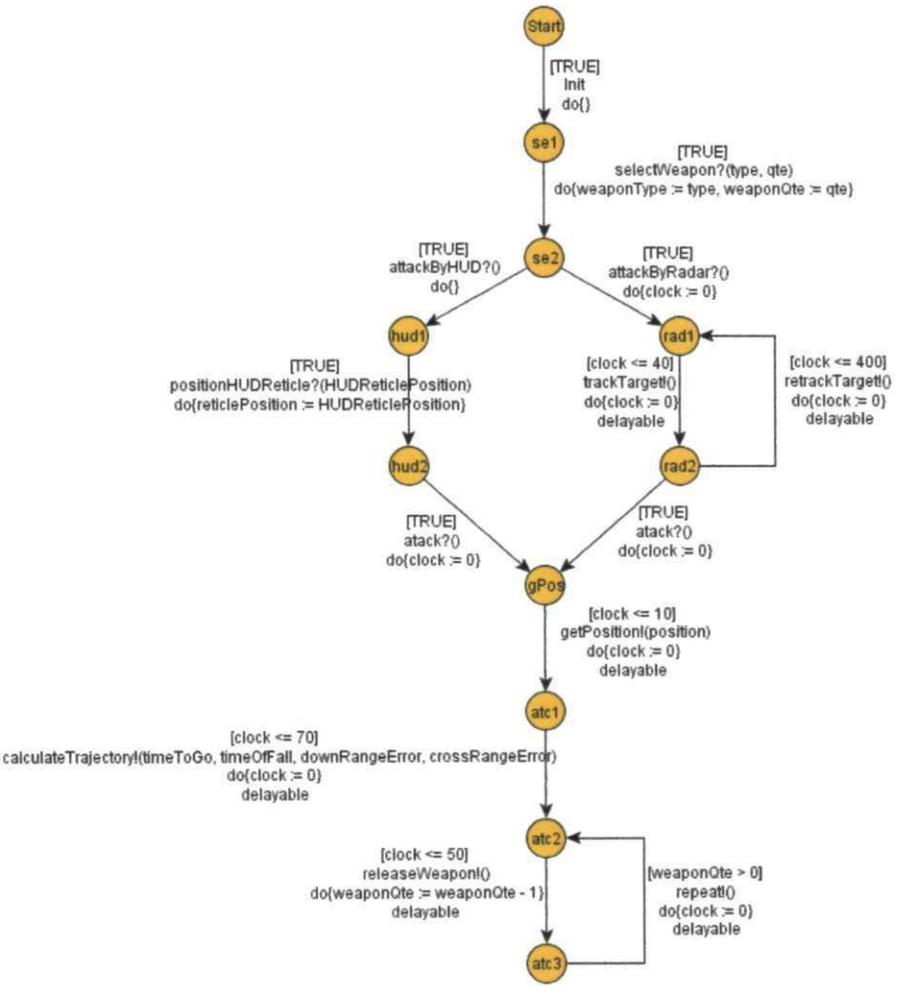


Figura 5.3: Modelo do Sistema de Ataque de um Caça

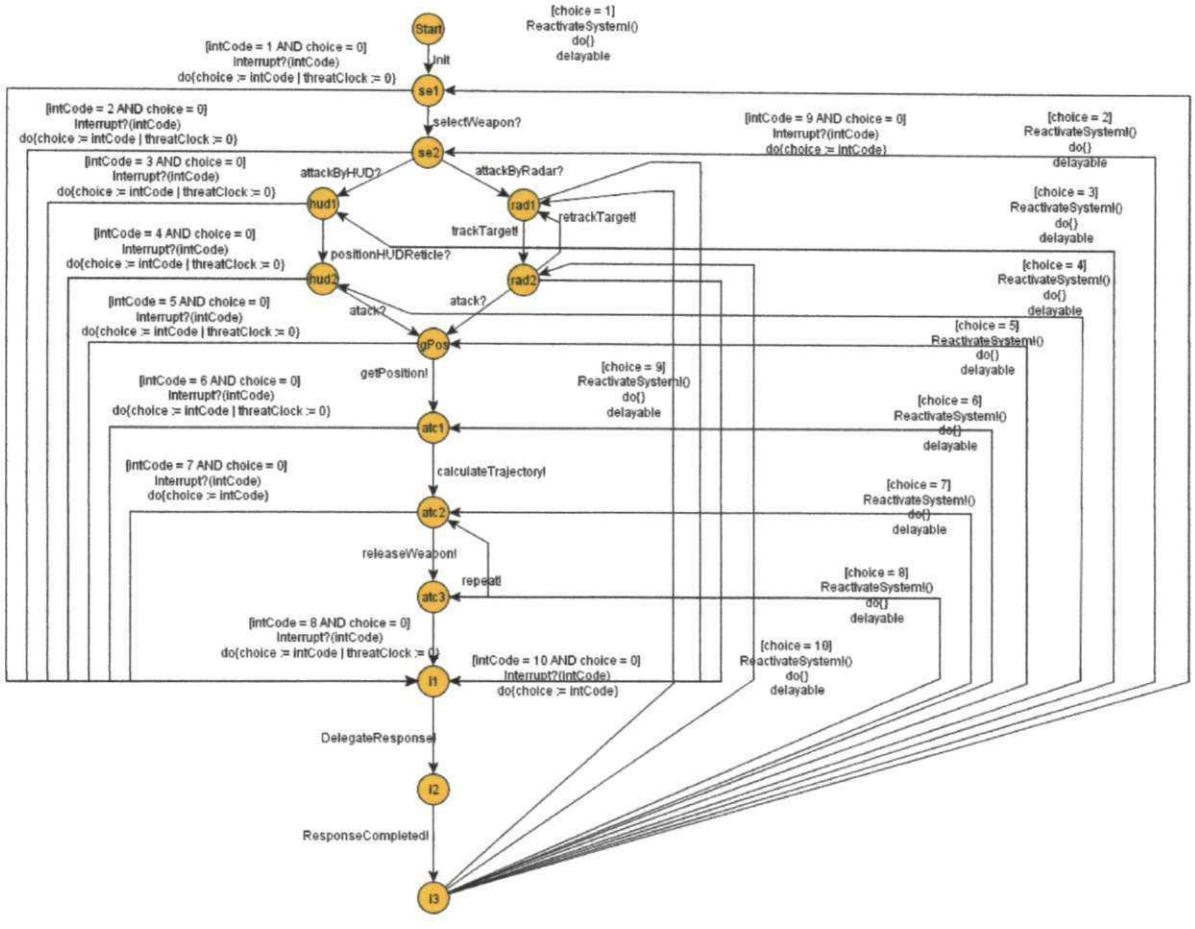


Figura 5.4: Modelo do Sistema de Ataque de um Caça com Tratamento a Ataque Inimigo

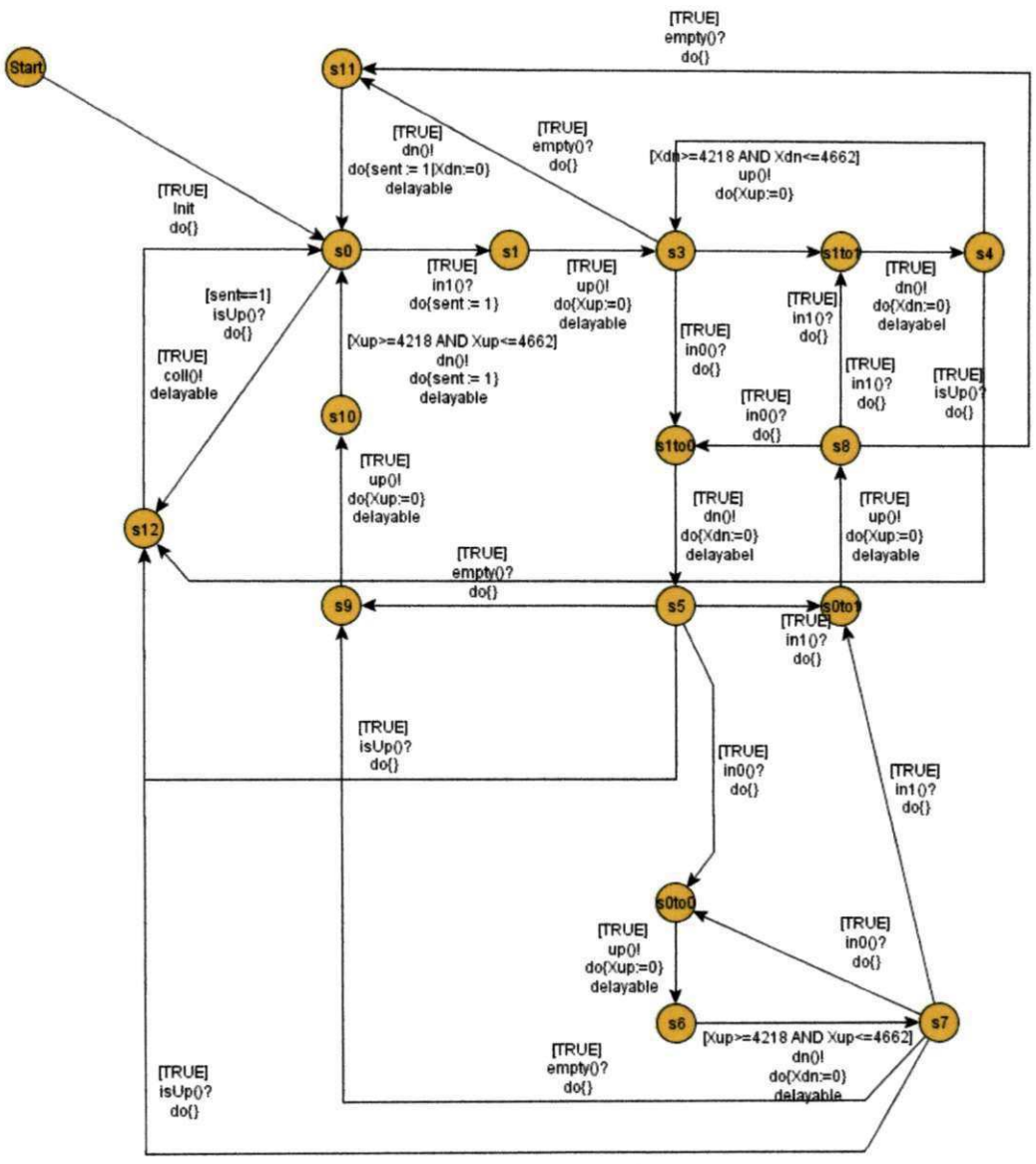


Figura 5.5: Modelo do Protocolo de Áudio da Philips

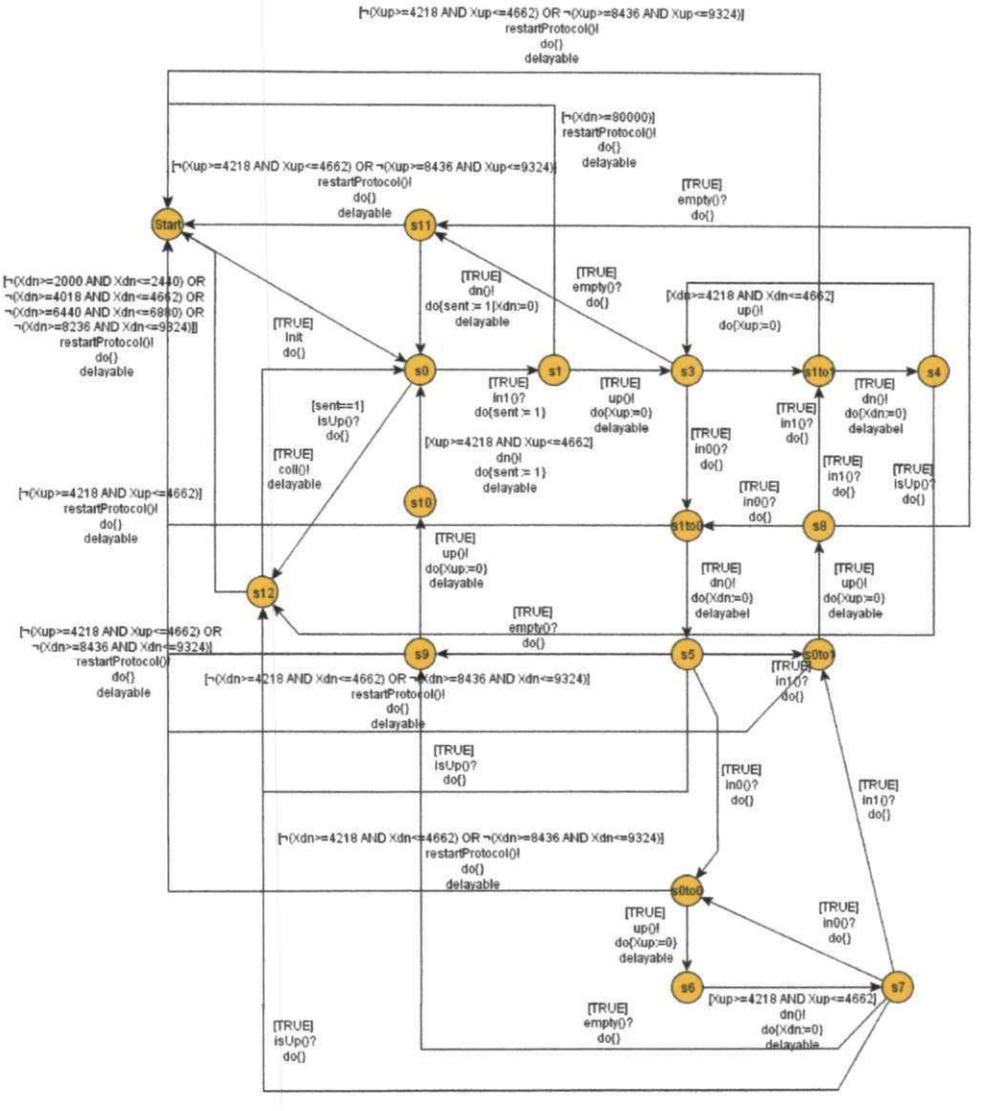


Figura 5.6: Modelo do Protocolo de Áudio da Philips com Tratamento a Erros

validade interna, validade de construção e validade externa.

Validade de Conclusão

- **Ameaça:** Como o experimento foi conduzido dentro do ambiente de indústria, é possível que o valor estimado para o nível de confiança seja menor do que o devidamente usado.
- **Impacto nos resultados do experimento:** Os resultados podem não ser estatisticamente significativos.
- **Ação tomada:** Foi utilizado o nível de confiança de 95%, pois segundo Jain [32] 0,05 é um nível de significância aceitável para comparações.

Validade Interna

- **Ameaça 1:** É possível que seja escolhida uma especificação de um STR que seja tendenciosa e conduza o resultado a um resultado desejado influenciando, dessa forma, o resultado do experimento.
- **Impacto nos resultados do experimento:** Os resultados podem ser tendenciados a um resultado desejado.
- **Ação tomada:** Para diminuir esse risco foram utilizadas especificações de outros autores como [48], [31] e [9].
- **Ameaça 2:** É possível que a estrutura do modelo tais como percentual de transições com restrições de tempo, quantidade de loops e quantidade de transições influencie no tamanho do conjunto de casos de teste gerado.
- **Impacto nos resultados do experimento:** A densidade de cobertura de falhas de cada critério não condiz com o resultado esperado.
- **Ação tomada:** Foram escolhidos modelos com estruturas bem diferentes a fim de representar melhor a diversidade dos possíveis modelos de STRs.

Validade de Construção

- **Ameaça 1:** É possível que os algoritmos utilizados na geração de casos de teste não estejam corretamente implementados.
- **Impacto nos resultados do experimento:** Os casos de teste gerados não satisfazem os critérios de geração.
- **Ação tomada:** A implementação foi supervisionada pelos orientadores desse trabalho e os casos de teste gerados por cada critério de geração e em cada modelo foram analisados para garantir que satisfazem o critério de geração.
- **Ameaça 2:** Para um mesmo modelo e o mesmo critério de geração é possível extrair casos de teste diferentes dependendo de como se caminha no modelo.
- **Impacto nos resultados do experimento:** Dificuldade em avaliar qual critério revela mais falhas se não houver uma uniformidade nos algoritmos.
- **Ação tomada:** Todos os algoritmos foram implementados seguindo o a mesma forma de caminhamento no modelo. Todos seguem o caminhamento por busca em profundidade (Depth First Search).

Validade Externa

- **Ameaça 1:** É possível que os modelos utilizados na execução do experimento (Seção 5.3) não sejam representativos para a maioria dos STRs.
- **Impacto nos resultados do experimento:** Os resultados obtidos podem não ser representativos para a maioria dos STRs.
- **Ação tomada:** Os modelos foram baseados em especificações reais de STRs.
- **Ameaça 2:** Capacidade reduzida de generalizar os resultados do experimento pela pequena quantidade de aplicações utilizadas para geração de casos de teste baseada nos critérios de geração.
- **Impacto nos resultados do experimento:** Os resultados obtidos podem não ser representativos para a maioria dos STRs.

- **Ação tomada:** Foram selecionados e modelados sistemas de tempo real cujos modelos apresentassem estruturas diferentes tais como quantidade de transições com restrições de tempo, caminhos diferentes no modelo (i.e. quantidade de *loops*, bifurcações, etc.), e quantidade de *clocks* distintos.

5.3 Operação

Para executar esse experimento foi necessário implementar os oito critérios de geração de casos de teste da Seção 3.2. Os critérios foram implementados na ferramenta SYMBOLRT e seus códigos abstratos encontram-se no Apêndice B. Assim, para executar o código de cada critério é necessário termos modelos de sistemas de tempo real a partir dos quais serão extraídos casos de teste. Por isso, foram elaborados seis modelos que serviram de entrada para execução de cada caso de teste (ver Apêndice C). Porém, para determinar a capacidade de revelar falhas é necessário conhecer as possíveis falhas que os sistemas a serem testados podem apresentar. Assim, foram elaborados modelos de falhas dos seis sistemas modelados. As falhas foram baseadas no trabalho de [25] e o modelo de falhas pode ser encontrado no Apêndice C.4.

5.4 Análise e Interpretação

Após executado o experimento observou-se quais falhas foram cobertas pelos casos de teste de cada modelo utilizado. Os casos de teste gerados por cada critério de geração em um dos seis modelos utilizados como objeto de instrumentação desse experimento pode ser contrado em https://sites.google.com/a/computacao.ufcg.edu.br/rtscoverage. As Tabelas 5.2, 5.3, 5.4, 5.5, 5.6, 5.7 indicam quais as falhas que foram cobertas por cada critério de geração por cada modelo utilizado.

	All One Loop Paths	***All Transitions***	***All Locations***	***All Clock Zones***	***All Clock Reset***	***Definition Use Pair***	***All Du Paths***	***All Defs***
Sistema de Alarme								

Falha 1	X	X			X			
Falha 2	X	X			X			
Falha 3	X	X			X			
Total de Falhas Capturadas	3	3	0	0	3	0	0	0
Tamanho do Conjunto de Casos de Teste	24	5	2	3	4	18	9	5
Densidade	0,125	0,6	0	0	0,75	0	0	0

Tabela 5.2: Falhas capturadas do Sistema de Alarme.

	All One Loop Paths	***All Transitions***	***All Locations***	***All Clock Zones***	***All Clock Reset***	***Definition Use Pair***	***All Du Paths***	***All Defs***
Sistema de Alarme com Tratamento a Queda de Energia								
Falha 4	X					X	X	
Falha 5	X	X		X		X	X	
Falha 6	X	X		X		X	X	
Falha 7	X					X	X	
Falha 8	X					X	X	
Falha 9	X					X	X	
Falha 10	X							
Total de Falhas Capturadas	6	2	0	2	0	6	6	0

Tamanho do Conjunto de Casos de Teste	204	8	2	8	4	91	49	6
Densidade	0,023	0,25	0	0,25	0	0,066	0,122	0

Tabela 5.3: Falhas capturadas do Sistema de Alarme com Tratamento a Queda de Energia.

	All One Loop Paths	***All Transitions***	***All Locations***	***All Clock Zones***	***All Clock Reset***	***Definition Use Pair***	***All Du Paths***	***All Defs***
Sistema de Ataque de um Caça								
Falha 11	X	X	X	X	X	X	X	X
Falha 12	X	X	X	X	X	X	X	X
Falha 13	X	X	X	X	X	X	X	X
Total de Falhas Capturadas	3	3	3	3	3	3	3	3
Tamanho do Conjunto de Casos de Teste	25	3	2	3	3	21	9	6
Densidade	0,12	1	1,5	1	1	0,143	0,333	0,5

Tabela 5.4: Falhas capturadas do Sistema de Ataque de um Caça.

	All One Loop Paths	***All Transi-tions***	***All Lo-cati-ons***	***All Clock Zones***	***All Clock Reset***	***Definition Use Pair***	***All Du Paths***	***All Defs***
Sistema de Ataque de um Caça com Tratamento a Ataque Inimigo								
Falha 14	X					X	X	
Falha 15	X				X	X	X	
Falha 16	X	X		X	X			
Falha 17	X	X		X	X	X	X	
Falha 18	X				X	X	X	
Falha 19	X				X	X	X	
Falha 20	X					X	X	
Total de Falhas Capturadas	7	2	0	2	5	6	6	0
Tamanho do Conjunto de Casos de Teste	263	11	2	11	11	195	91	17
Densidade	0,027	0,182	0	0,182	0,454	0,031	0,066	0

Tabela 5.5: Falhas capturadas do Sistema de Ataque de um Caça com Tratamento a Ataque Inimigo.

	All One Loop Paths	***All Transi-tions***	***All Lo-cati-ons***	***All Clock Zones***	***All Clock Reset***	***Definition Use Pair***	***All Du Paths***	***All Defs***
Protocolo de Áudio da Philips								

Falha 21	X	X	X					
Falha 22	X	X	X	X	X	X	X	X
Falha 23	X	X	X	X	X	X	X	X
Falha 24	X	X	X	X	X	X	X	X
Falha 25	X	X		X	X	X	X	X
Falha 26	X	X	X	X	X	X	X	X
Falha 27	X	X		X		X	X	
Total de Falhas Capturadas	7	7	5	6	5	6	6	5
Tamanho do Conjunto de Casos de Teste	409	11	7	10	4	46	36	7
Densidade	0,017	0,636	0,714	0,6	1,250	0,130	0,167	0,714

Tabela 5.6: Falhas capturadas do Protocolo de Áudio da Philips.

	All One Loop Paths	***All Transitions***	***All Locations***	***All Clock Zones***	***All Clock Reset***	***Definition Use Pair***	***All Du Paths***	***All Defs***
Protocolo de Áudio da Philips com Tratamento de Erro de Protocolo								
Falha 21	X	X	X			X	X	
Falha 22	X	X	X	X	X	X	X	X
Falha 23	X	X	X	X	X	X	X	X
Falha 24	X	X	X	X	X	X	X	X
Falha 25	X	X		X	X	X	X	X
Falha 26	X	X	X	X	X	X	X	X

Falha 27	X	X		X		X	X	
Total de Falhas Capturadas	7	7	5	6	5	7	7	5
Tamanho do Conjunto de Casos de Teste	649	16	7	15	4	300	202	10
Densidade	0,011	0,437	0,714	0,4	1,250	0,023	0,035	0,5

Tabela 5.7: Falhas capturadas do Protocolo de Áudio da Philips com Tratamento de Erro de Protocolo.

Organizando os dados das Tabelas 5.2, 5.3, 5.4, 5.5, 5.6, 5.7 em *box-plots* de cada critério na Figura 5.7, podemos perceber melhor a diferença na quantidade de falhas reveladas por critérios de geração. O critério *All One Loop Paths* é o critério que, em média, mais consegue revelar falhas enquanto que o critério *All Locations* e o *All Defs* são os que, em média, menos conseguem revelar falhas. Porém, Analisar apenas a quantidade de falhas reveladas não é prudente. É necessário relacionar quantidade de falhas e tamanho do conjunto de casos de teste gerado por cada critério. Assim, chamando de densidade de falhas a razão entre a quantidade de falhas descobertas pelo tamanho do conjunto de casos de teste de cada critério, podemos ver, através da Figura 5.8, que também há diferença em relação a densidade de falhas. No entanto, para garantir estatisticamente que realmente há diferença na densidade de falhas é necessário realizar um teste de hipótese e verificar se os dados obtidos permitem rejeitar a hipótese nula da Seção 5.2.3.

Esse experimento envolve um único fator com oito níveis, então o teste de hipótese adequado é ANOVA [32], [54]. Porém, trata-se de um teste paramétrico e, desta forma, esse teste parte do princípio que os dados seguem a distribuição normal. Para verificarmos se a densidade de falhas seguem essa distribuição plotamos o gráfico quantil-quantil da densidade de falhas de cada critério.

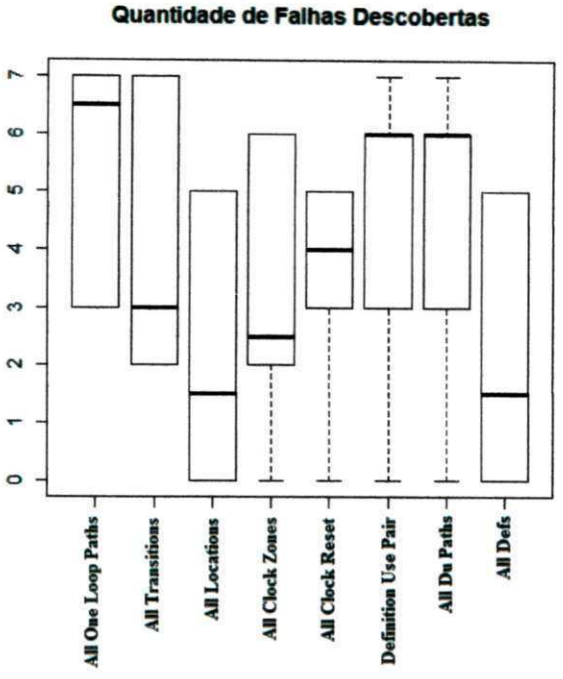


Figura 5.7: Box-plot da quantidade de falhas capturadas por critério.

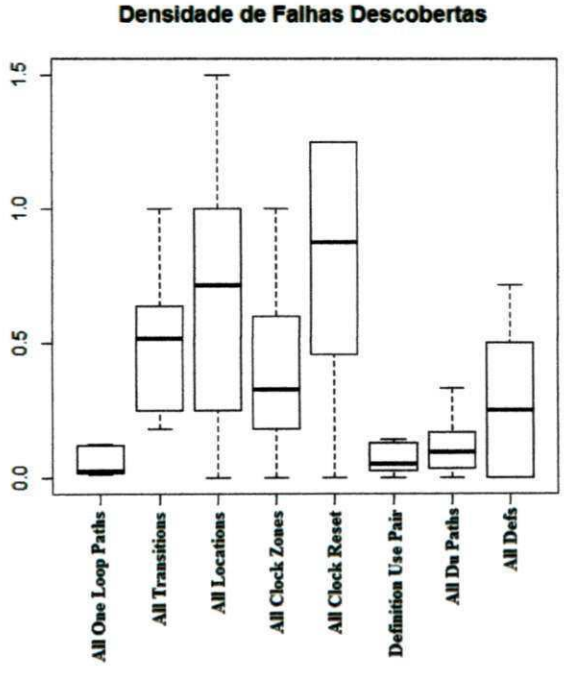
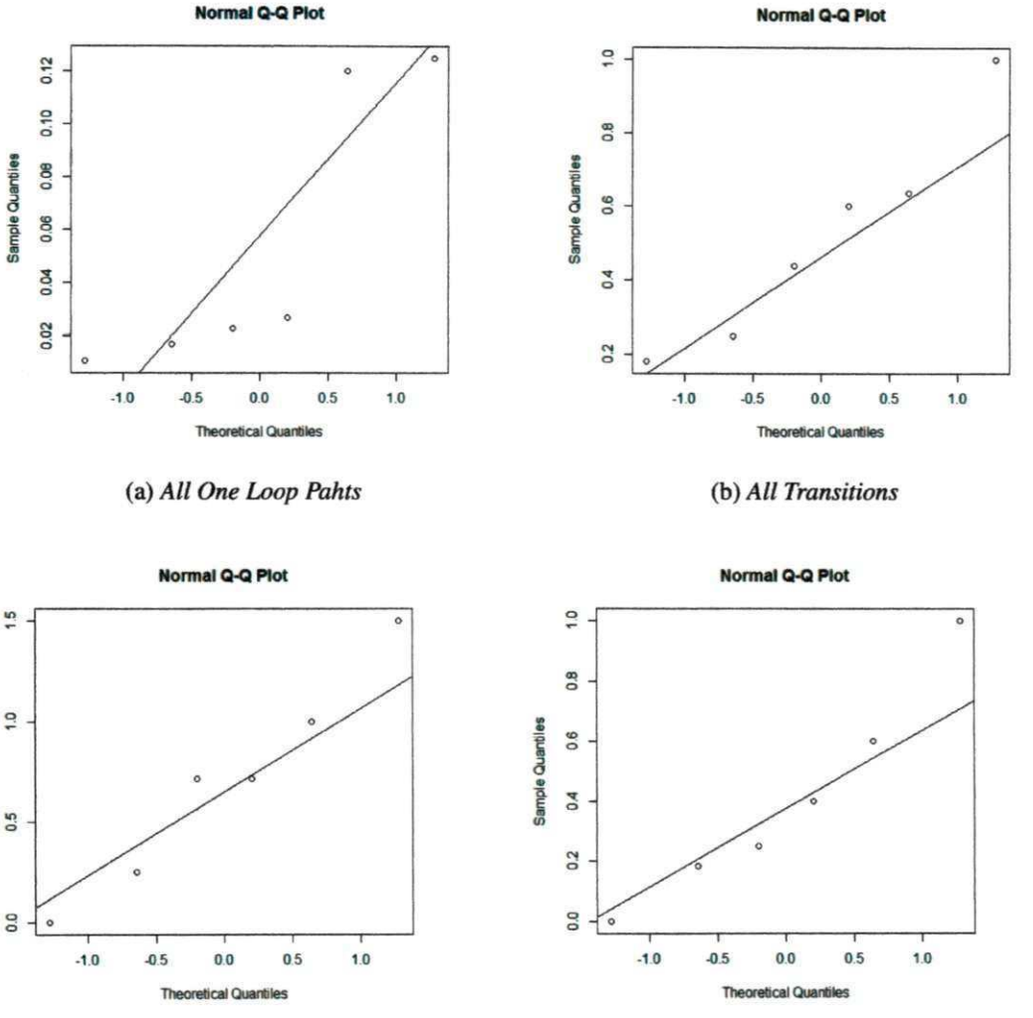


Figura 5.8: Box-plot da densidade de falhas capturadas por critério.

Na Figura 5.9 notamos que nem todas as amostras se situam próximas da linha que representa os quantis da distribuição normal, o que significa que nem todas as amostras advém de população que segue distribuição normal. Assim, o teste de hipótese utilizado não pode ser o ANOVA ou nenhum outro teste paramétrico.



(a) *All One Loop Pahts*

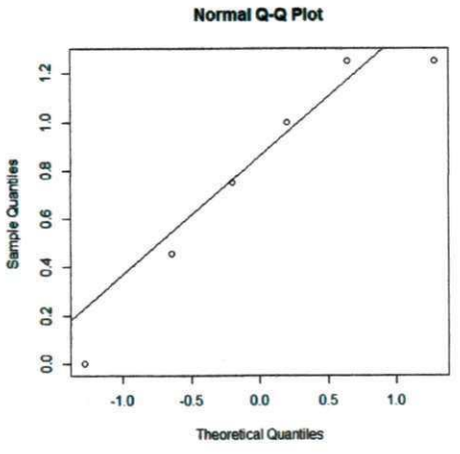
(b) *All Transitions*

(c) *All Locations*

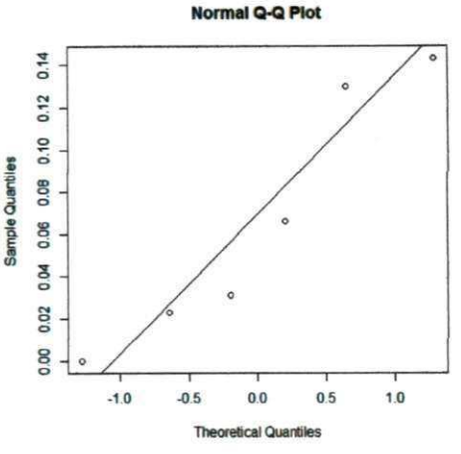
(d) *All Clock Zones*

Figura 5.9: Gráfico quantil-quantil da densidade de falhas reveladas de cada critério de geração.

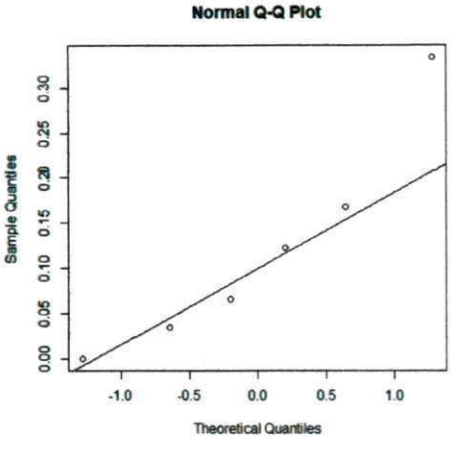
No resultado do teste, é mostrado o valor da estatística de teste, os graus de liberdade e, o mais importante, o p-valor do teste. A Tabela 5.8 indica o resultado do teste de Kruskal-Wallis para a densidade de falhas descobertas pelos critérios de geração.



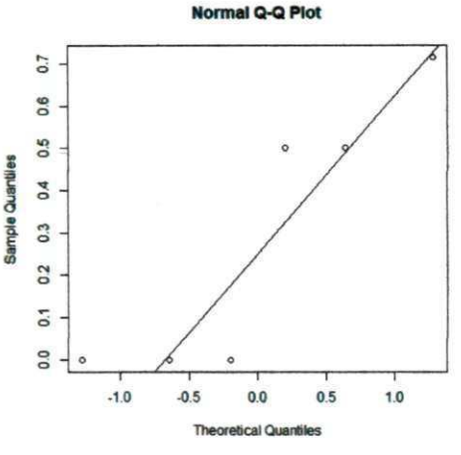
(e) *All Clock Reset*



(f) *Definition Use Pair*



(g) *All Du Paths*



(h) *All Defs*

Figura 5.9: Gráfico quantil-quantil da densidade de falhas reveladas de cada critério de geração.

Tabela 5.8: Resultados do Teste de Kruskal-Wallis.

Valor da Estatística de Teste	**Graus de Liberdade**	**p-valor**
Densidade de Falhas		
17,3928	7	0,01503

Como o p-valor é menor que o nível de significância de 0,05 podemos refutar com 95% de confiança a hipótese nula.

5.5 Observações Subjetivas

O critério *All One Loop Paths* é o critério que, em média, mais consegue revelar falhas. Neste experimento, todas as falhas do modelo de falhas foram capturadas pelo critério *All One Loop Paths*. No entanto, esse critério segue a metodologia de teste exaustivo e o conjunto de casos de teste gerado segundo esse critério é muito grande, sendo, portanto, esse critério aconselhado apenas quando o sistema de tempo real é crítico e/ou o custo da atividade de teste não é um fator decisivo para o sucesso do desenvolvimento do sistema.

Analisando os modelos, pode-se notar que quanto mais transições com restrições de tempo tiver o modelo torna-se mais interessante optar pelo critério *All Transition* pois ele consegue capturar mais falhas com uma quantidade de casos de teste baixa em relação aos outros critérios. No entanto, o critério *All Transitions* tem uma deficiência em capturar falhas provenientes de uma sequência de eventos. Por exemplo, a falha 6 'O sensor de movimento não detecta a invasão de um intruso no momento em que há uma queda de energia' consiste de dois eventos: 1 - primeiramente há uma queda de energia e; 2 - após essa queda de energia há uma invasão de um intruso.

O critério *All Clock Reset*, por ser um critério muito simples (simples por apenas cobrir as transições em que *clocks* são zerados), acreditava-se inicialmente que não seria capaz de revelar muitas falhas. No entanto, ele pode ser um critério muito efetivo tendo densidade entre 0 e 1,25. É um critério que gera poucos casos de teste e tem densidade alta podendo ser executado em cenários onde o custo de execução de casos de teste é elevado. Porém, a sua densidade varia muito e em alguns casos, como foi a densidade no modelo do 'Sistema de Alarme com Tratamento a Queda de Energia', pode ter densidade muito baixa.

O critério *Definition Use Pair* consegue capturar todas as falhas do modelo de falhas que são provenientes de não cumprimento de restrições de tempo e conseguiu, nesse experimento, capturar quase todas as falhas do modelo de falhas. No entanto, gera conjuntos de casos de teste muito grandes e, por isso, possui densidade muito baixa (densidade média de 0,065). Conforme o resultado do experimento, o critério *All Du Paths* consegue capturar

exatamente as mesmas falhas que o *Definition Use Pair*, no entanto possui densidade de 0.12 que representa quase o dobro da densidade do critério *Definition Use Pair* e, com isso, torna-se um critério mais adequado a se utilizar quando deseja-se dar mais importância nas falhas de restrições de tempo dos sistemas de tempo real.

O critério *All Defs* não se mostrou ser um critério com boa capacidade de revelar falhas. De um total de 34 falhas (27 falhas do modelo de falhas repetindo-se as 7 últimas no modelo 'Protocolo de Áudio da Philips com Tratamento de Erro de Protocolo') apenas 13 foram capturadas e das 19 falhas (considerando as repetidas) decorrentes de não cumprimento de restrições de tempo, ele conseguiu capturar apenas 11 falhas. O critério *All Locations* foi o menos promissor. Das 34 falhas, o critério capturou apenas 13. Além disso, nos modelos Sistema de Alarme, Sistema de Alarme com Tratamento a Queda de Energia e Sistema de Ataque de um Caça com Tratamento a Ataque Inimigo, o critério não conseguiu capturar nenhuma falha.

5.6 Considerações Finais

Este capítulo relatou o estudo experimental realizado com a finalidade de verificar diferença na densidade de falhas reveladas pelos critérios de geração de casos de teste descritos na Seção 3.2. Verificou-se com 95% de confiança que é possível refutar a hipótese nula e assim afirmar que realmente há diferença na densidade dos critérios de geração. Uma análise subjetiva foi realizada sobre os critérios estudados de forma que possa auxiliar o testador na hora de escolher qual critério de geração adotar.

Capítulo 6

Trabalhos Relacionados

Com o intuito de identificar o estado da arte no contexto de critérios de geração de casos de teste pra sistemas de tempo real ao nível de modelo foi realizada uma revisão sistemática (ver Apêndice A). O objetivo deste capítulo é descrever brevemente os principais trabalhos que abordam essa problemática [23], [44], [8], [12], [17], [24], [29], [28], [37], [49]. Entretanto, nenhum dos trabalhos realizam algum estudo sobre a capacidade de revelar falhas em sistemas de tempo real limitando-se apenas a descrever um critério ou um conjunto de critérios e, em alguns trabalhos, implementá-lo em uma ferramenta.

En-Nouaary [23] é o trabalho que mais se aproxima do trabalho proposto nesse mestrado, assim este capítulo foi dividido da seguinte maneira: a primeira seção apresenta o trabalho de En-Nouaary [23], a segunda seção apresenta os trabalhos que implementam critérios de geração em alguma ferramenta de automação de geração de casos de teste, e, por fim, a terceira seção apresenta os trabalhos que apenas descrevem ou analisam critérios de geração, mas não os implementa em uma ferramenta de automação de geração de casos de teste.

6.1 En-Nouaary

En-Nouaary [23] é o trabalho que mais se aproxima do trabalho proposto nesse mestrado. O *paper* apresenta um conjunto de critérios de geração de casos de teste ordenados segundo uma relação de inclusão conforme a Figura 6.1. A relação de inclusão implica que se um critério c_1 inclui um critério c_2 então qualquer conjunto de casos de teste que satisfaz c_1 também satisfaz c_2. c_1 inclui estritamente c_2 de forma que há casos de teste em c_2 que não

satisfazem c_1.

Figura 6.1: Critérios de Geração Ordenados Segundo a Relação de Inclusão.

Os critérios de geração apresentados por En-Nouaary são descritos da seguinte maneira:

Seja *A* um TIOA (*Timed Input Output Automata*), conforme a Definição 6.1, e *TS* um conjunto de casos de teste gerado a partir de *A*: Os critérios de geração são:

- *All-traces*: *TS* satisfaz o critério *all-traces* se todo *trace* de *A* está incluído na *TS*;
- *All-paths*: *TS* satisfaz o critério *all-paths* se cada caminho de *A* é coberto por pelo menos um caso de teste de *TS*;
- *All-states*: *TS* satisfaz o critério *all-states* se cada estado de *A* é visitado por pelo menos um caso de teste de *TS*;
- *All-transitions*: *TS* satisfaz o critério *all-transitions* se cada transição de *A* é coberta por pelo menos um caso de teste de *TS*;
- *All-clock-valuations*: *TS* satisfaz o critério *all-clock-valuations* se cada valor de *clock* é alcançado por pelo menos um caso de teste de *TS*;

- *All-inputs*: *TS* satisfaz o critério *all-inputs* se cada entrada de *A* é exercida por pelo menos um caso de teste de *TS*;
- *All-outputs*: *TS* satisfaz o critério *all-output* se cada saída de *A* é exercida por pelo menos um caso de teste de *TS*;
- *All-locations*: *TS* satisfaz o critério *all-locations* se cada *location* de *A* é visitada por pelo menos um caso de teste de *TS*;
- *All-clock-regions*: *TS* satisfaz o critério *all-clock-regions* se cada região de *clock* de *A* for visitado por pelo menos um caso de teste de *TS*;
- *All-clock-zones*: *TS* satisfaz o critério *all-clock-zones* se cada zona de *clock* de *A* for visitado por pelo menos um caso de teste de *TS*;
- *All-clock-guards* ou *All-time-constraints*: *TS* satisfaz o critério *all-clock-guards* (ou *all-time-constraints*) se cada guarda de *A* for exercitada por pelo menos um caso de teste de *TS*;
- *All-clock-guard-bounds*: *TS* satisfaz o critério *all-clock-guard-bounds* se cada limite de guarda de *clock* em *A* é exercitada por pelo menos uma suíte de teste de *TS*;
- *All-clock-bounds*: *TS* satisfaz o critério *all-clock-bounds* se cada limite de *clock* em *A* é exercitado por pelo menos um caso de teste de *TS*;
- *All-clock-reset*: *TS* satisfaz o critério *all-clock-reset* se cada reset de *clock* é exercitado por pelo menos um caso de teste de *TS*.

Definição 6.1 (TIOA). *UM TIOA é uma tupla (I, O, L, l0, C, T), onde:*

- *I é um conjunto finito de mensagens de entradas. En-Nouaary [23] denota mensagens de entrada por ?*
- *O é um conjunto finito de mensagens de saídas. En-Nouaary [23] denota mensagens de entrada por !*
- *L é um conjunto finito de locations. Um location representa a posição do sistema após a execução de uma transição.*

- *l0 ∈ L é o location inicial do TIOA.*
- *C é um conjunto finito de clocks, que são zerados em l0. Um clock é uma variável de tempo que conta quanto tempo se passou desde que o clock foi zerado. Em En-Nouaary [23], clocks recebem valores reais positivos.*
- *T é um conjunto finito de transições.*

◇

En-Nouaary não apresenta nenhum estudo mais aprofundado como um experimento ou estudo de caso de nenhum dos critérios de geração apresentados. O estudo se limita a discutir brevemente sobre cada critério e a organizá-los segundo a relação de inclusão. Apesar de não haver um estudo mais aprofundado sobre os critérios o *paper* trás como contribuição um *survey* sobre os critérios de geração de casos de teste para sistemas de tempo real modelados por *timed automata* servindo como base para o desenvolvimento de métodos de teste para *timed automata*.

6.2 Propostas de critérios de geração sem implementações

Esta seção apresenta os trabalhos que implementam critérios de geração de casos de teste para sistemas de tempo real em alguma ferramenta de geração automática de casos de teste.

6.2.1 Nielsen et al.

Nielsen et al. [44] apresenta uma ferramenta baseada em métodos formais para geração automática de casos de teste de conformidade para sistemas críticos. A geração de testes é baseada na seleção por critério de cobertura da especificação descrito em ERA (*Event Recording Automata*) proposto por Alur et al. [3].

Os casos de teste são gerados sistematicamente a partir de um critério de cobertura da especificação. O espaço de estados da especificação é particionado em classes de equivalência que preservam características temporais e sincronizam informações. A geração de casos de teste garante que cada classe de equivalência alcançável vai ser coberto pelo conjunto de casos de teste. Portanto, é empregada a técnica de análise simbólica de alcançabilidade

baseada em *constraint solving* que tem sido recentemente desenvolvido por *model checking* de *timed automata*.

A ênfase do *paper* é na ferramenta de geração de testes e uma aplicação dela e, portanto, não tem foco na capacidade de revelar falhas do critério adotado na geração de casos de teste. Nielsen et al. denomina de *stable edge set criterion* o critério de geração que particiona o espaço de estados da especificação em classes de equivalência. Os estados (par consistindo de *locations* e valores de *clock*) do autômato são particionados de forma que dois valores de *clock* pertencem à mesma classe de equivalência se, e somente se, eles habilitam precisamente as mesmas arestas do conjunto de estados que o autômato pode estar ocupando. Por isso, é aplicada análise simbólica empregando-se o conceito de zonas que são representadas por DBM (*difference bound matrix*) [21].

6.2.2 Krichen et al.

Krichen et al. [37] propõe um novo framework para teste de conformidade caixa preta para sistemas de tempo real considerando dois tipos de teste: teste de *clock* analógico e teste de *clock* digital onde *clocks* analógicos são representados como *timed automata* determinísticos. O trabalho também provê algoritmos para geração estática ou *on-the-fly* de testes de *clock* digital além de propor técnicas para cobertura de *locations*, arestas e estados da especificação, reduzindo o problema para cobertura de um grafo simbólico de alcançabilidade (*Observable Graph*). O trabalho trás a implementação do protótipo de uma ferramenta chamada TTG e dois estudos de caso: um dispositivo de iluminação e do Protocolo de Retransmissão Limitado.

O trabalho apresenta três critérios de cobertura:

- *Edge coverage*: cada aresta alcançável do modelo da especificação é coberto por pelo menos um caso de teste do conjunto de casos de teste.
- *Location coverage*: cada *location* alcançável do modelo da especificação é coberto por pelo menos um caso de teste do conjunto de casos de teste.
- *Action coverage*: cada ação alcançável do modelo da da especificação é coberto por pelo menos um caso de teste do conjunto de casos de teste.

6.3 Trabalhos que não Implementam Critérios de Geração em uma Ferramenta de Geração Automática

Esta seção apresenta os trabalhos que estudam critérios de geração de casos de teste para sistemas de tempo real, mas sem implementá-los em alguma ferramenta de geração automática de casos de teste.

6.3.1 Arcuri et al.

Arcuri et al. [8] adota uma abordagem caixa preta e modela o ambiente RTES (*Real Time Embedded Systems*) usando UML/MARTE. O *paper* foca a seleção de casos de teste e investiga três estratégias de geração usando entradas a partir de modelos do ambiente UML/-MARTE: *Random Testing* (*baseline*), *Adaptive Random Testing*, e *Search-Based Testing* (usando algoritmos genéticos).

Arcuri et al. segue a abordagem TBM e adaptou os princípios de *Adaptive Random Testing* (ART) e *Search-Based Testing* (SBT) para o problema e contexto explorados pelo trabalho. *Random Testing* (RT) é usado como *baseline* para avaliação empírica.

A Figura 6.2 ilustra um exemplo de uma máquina de estado UML/MARTE. A escolha não determinística C -> D recebe o parâmetro $T \in [0, 1]$ que determina o *timeout* a ser esperado para que a transição seja ativada. Os valores que podem ser alocados a T são alocados a um vetor no caso de teste com tamanho l. Toda vez que a transição é ativada é escolhido um valor do vetor. Considerando, por exemplo, o valor $l = 2$, nós podemos ter um vetor contendo, por exemplo, 0.4,0.32. A primeira vez que a transição C -> D é ativada, o valor 0.4 é usado. Na segunda vez, o valor 0.32 é usado. Na terceira vez, o valor 0.4 é usado novamente e assim sucessivamente.

A abordagem do *paper* consiste em determinar a melhor forma de escolher os valores de T. A forma mais simples de escolher é a *Random Testing* (RT). Para cada variável V, um valor do seu domínio é escolhido com a mesma probabilidade que qualquer outro valor desse domínio.

Outra técnica investigada pelo *paper* é *Adaptive* RT (ART), que foi proposta como uma extensão de RT. A ideia de ART consiste na diversidade dos casos de teste, dado que casos

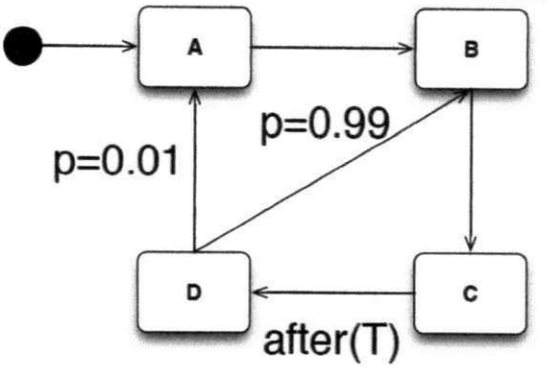


Figura 6.2: Exemplo de uma máquina de estados UML/MARTE.

de teste com falhas tendem a estar presentes em regiões próximas e contínuas do domínio de entrada. ART pode ser automatizada se for definida uma função de similaridade útil para casos de teste.

Arcuri et al. também investiga o uso de algoritmos de busca para enfrentar o teste de RTES. Em particular é considerado o de Algoritmos Genéticos (GAs para *Generic Algorithms*), que são os algoritmos de busca mais usados na literatura em *Search-Based Testing* (SBT). Para usar algoritmos de busca para atacar o problema específico, uma função de aptidão precisa ser adaptada para resolver o problema. A função de aptidão é usada para, heuristicamente, avaliar quão bom é um caso de teste. Nesse caso, a função de aptidão é usada para estimar a proximidade do caso de teste de disparar uma falha no RTES, isto é, quando pelo menos um componente do ambiente entra em um estado de erro.

6.3.2 Clarke et al.

Clarke et al. [17] apresenta um *framework* para testar restrições de tempo de sistemas de tempo real. Os testes são derivados automaticamente da especificação de *deadlines* mínimos e máximos entre eventos de saída e entrada na execução do sistema. O *paper* utiliza ACSR (*Algebra of Communicating Shared Resources*) para descrever os testes e os modelos do sistema. ACSR é um processo formal algébrico adaptado para o domínio de sistemas de tempo real concorrentes, com notação explícita de tempo, recursos reusáveis em série, e prioridade.

Os testes são aplicados para uma fórmula algébrica de uma implementação proposta do

sistema escrito em ACSR. A técnica para derivar e descrever testes apresentada por Clarke et al. são suportados por ferramentas de software para (1) construir uma representação gráfica de restrições de tempo; (2) derivar casos de teste otimizados a partir das restrições de tempo; e (3) aplicar testes para os modelos de processo ACSR e reportar os resultados.

O Paradigma ACSR é baseado na visão que um sistema de tempo real consiste de um conjunto de processos comunicantes que executam em um conjunto finito de recursos reusáveis em série que se sincronizam entre si através de canais de comunicação.

Clarke et al. discute que testar todo o domínio de valores de tempo é uma atividade impossível. Portanto, uma aproximação razoável seria testar o que o *paper* chama de *endpoints* de cada domínio, mas discute que testar os *endpoints* do domínio não é uma atividade realizável se o domínio contiver *endpoints* abertos (intervalos de valores abertos). *Endpoints* são os valores que estão nas bordas dos intervalos delimitados pelas guardas de tempo, ou seja, considerando a e b constantes tal que $a < b$ o intervalo $[a, b[$ teria como *endpoint* inferior o valor a e como *endpoint* superior b.

Para tratar dessas situações, é definido um critério apropriado que leva em conta aproximações dos endpoints ao invés de considerar exatamente os valores mínimos e máximos.

Clarke et al. propõe os seguintes critérios de cobertura:

- *All-input-bounds-approximate*: O conjunto de casos de teste satisfaz o critério *all-input-bounds-approximate* se os *endpoints* ou valores próximos dos *endpoints* são exercitados por pelo menos um caso de teste.

- *All-input-actions criterion*: O conjunto de casos de teste satisfaz o critério *all-input-actions* se cada ação de entrada é exercitada por pelo menos um caso de teste.

- *All-timeout-actions*: O conjunto de casos de teste satisfaz o critério *all-timeout-actions* se o *timeout* de cada ação de entrada é testado por pelo menos um caso de teste.

- *All-output-actions*: O conjunto de casos de teste satisfaz o critério *all-output-actions* se cada ação de saída é exercitada por pelo menos um caso de teste.

- *All-I/O-actions*: O conjunto de casos de teste satisfaz o critério *all-I/O-actions* se o conjunto de casos de teste também satisfizer os critérios *all-input-actions* e *all-output-actions*.

6.3.3 En-Nouaary et al.

En-Nouaary et al. [24] propõe um método para teste de sistemas de tempo real, formalmente modelado como TIOA (*Timed Input Output Automata*), que visa gerar um conjunto de casos de teste que permitiria verificar cada transição do TIOA o mais breve possível, o mais tarde possível, e no momento entre estas duas execuções, ou seja, verifica-se nas bordas da região de *clock* delimitada pela guarda da transição bem como o momento entre as bordas (ponto médio da região). Como para cada transição são gerados três casos de teste então o conjunto de casos de teste terá tamanho de três vezes a quantidade de transições do modelo tornando a abordagem escalável para sistemas grandes. Os momentos da execução de cada transição são determinados com base nos *deadlines* mínimos e máximos entre o estado fonte da transição e sua guarda de *clock*.

6.3.4 Hessel et al.

Hessel et al. [29], [28] ataca dois problemas: como especificar formalmente critério de cobertura e como gerar um conjunto de casos de teste um modelo formal de um sistema tal que o conjunto de casos de teste satisfaça um determinado critério de cobertura. A abordagem dada pelo teste é converter o problema de geração de casos de teste em problema de alcançabilidade. Hessel et al. denomina itens de cobertura o conjunto de itens cobertos por um critério de cobertura. O problema de gerar um caso de teste para cada item de cobertura pode ser tratado como problema de alcançabilidade em separado. A informação de cobertura é utilizada para selecionar um conjunto de casos de teste que juntos satisfazem todos os itens de cobertura e, portanto, o critério de cobertura completo.

Hessel et al. apresenta um algoritmo de geração de casos de teste que de uma máquina de um modelo EFSM (*Extended Finity State Machine*) e um critério de cobertura gera um conjunto de casos de teste que satisfaz o critério de cobertura. O algoritmo proposto pode então ser usado para qualquer critério de cobertura. Hessel et al. não comenta sobre capacidade dos critérios em revelar falhas.

Hessel et al. descreve os seguintes critérios de cobertura:

- *Reach coverage* ou *definition-use pair (du-pair)*: Requer que um conjunto de casos de testeinclua todos os caminhos da definição de uma variável x para todas as transições

alcançáveis onde x é usada.

- *Context coverage* ou *definition context coverage*: Um contexto de uma definição de variável é as arestas em que as variáveis usadas para a definição são definidas. Por exemplo, para a associação $x := y + z$ o contexto é (e_y, e_z) se y foi definido em e_y e z foi definido em e_z. O critério requer que um conjunto de teste inclua todos os caminhos tal que para toda definição de uma variável x, cada contexto diferente da definição é representada.

- *Ordered context coverage*: Similar a *context coverage*, no entanto, as arestas no contexto são listadas na ordem de suas definições.

- *All-paths*: Inclui todos os caminhos possíveis do modelo.

- *All-defs*: Segundo a tese é o critério de fluxo de dados mais fácil de se alcançar. Nesse critério é suficiente cobrir para cada definição apenas um caminho que leve de um uso.

- *All-p-uses*: Similar ao *reach coverage*, mas o uso da variável deve ser em um predicado.

- *All-c-uses*: Similar ao *reach coverage*, mas o uso da variável deve ser uma associação.

- *All-du-path*: Similar ao *reach coverage*, mas para cada definição de uma variável é suficiente encontrar todos os caminhos para o primeiro uso da variável.

6.4 Considerações Finais

Este capítulo apresentou os principais trabalhos relacionados a critérios de geração de casos de teste para sistemas de tempo real encontrados na literatura através de uma revisão sistemática realizada nesse trabalho. Como podemos observar no resumo de cada trabalho, nenhum deles estuda a capacidade de revelar falhas dos critérios de geração. En-Nouaary [23] é o trabalho que mais se aproxima do trabalho proposto nesse mestrado, no entanto En-Nouaary apenas apresenta um conjunto de critérios que podem ser utilizados na geração de casos de teste para sistemas de tempo real e não faz qualquer comparação entre capacidade de revelar falhas nem tamanho do conjunto de casos de teste que satisfazem cada critério. Arcuri et al.

[8] realiza um estudo comparativo de três critérios de geração, porém os critérios comparados são critérios de cobertura de dados e como esse trabalho trata de critérios de cobertura estrutural de modelo então não se enquadra no escopo desse trabalho de mestrado. Os demais trabalhos apenas utilizam algum critério de geração para gerar casos de teste, mas sem realizar um estudo ou comparação sobre sua capacidade de revelar falhas. A Tabela 6.1 sumariza os trabalhos relacionados apresentados nesse capítulo, nela podemos encontrar quais foram os critérios investigados por cada trabalho, qual ferramenta implementa os critérios (se houver) e qual tipo de avaliação o trabalho utiliza para investigar os critérios (i.e. survey, estudo de caso, experimentação).

Trabalho	**Criterios Investigados**	**Ferramenta**	**Tipo de Avaliação**
En-Nouaary [23]	*All Locations*, *All Transitions*, *All States*, *All paths*, *All traces*, *All inputs*, *All outputs*, *All clock valuations*, *All clock regions*, *All clock zones*, *All clock guard bounds*, *All time constraints*, *All clock bounds*, *All clock reset*	Não possui	Survey
Nielsen et al. [44]	*Stable edge set criterion*	RTCAT	Estudo de Caso
Krichen et al. [37]	*Edge coverage*, *Location coverage*, *Action coverage*	TTG	Estudo de Caso
Arcuri et al. [8]	*Random Testing*, *Adaptive random testing*, *Search-based testing*	Não Possui	Experimento

Clarke et al. [17]	*All inputs*, *All outputs*, *All-I/O-actions*, *All-input-bounds-approximate*, *All-timeout-actions*	Não Possui	Estudo de Caso
En-Nouaary et al. [24]	*All clock guard bounds*	Não Possui	Estudo de Caso
Hessel et al. [29], [28]	*All paths*, *Definition-use pair coverage*, *Context coverage*, *Ordered context coverage*, *All Defs*, *All-p-uses*, *All-c-uses*, *All-du-path*	Não Possui	Estudo de Caso

Tabela 6.1: Resumo dos Trabalhos Relacionados.

Capítulo 7

Conclusões

O trabalho apresentado neste documento descreve o estudo realizado sobre critérios de geração de casos de teste para sistemas de tempo real em TBM. Inicialmente, foi conduzida uma revisão sistemática com a finalidade de encontrar trabalhos na literatura que abordem questões sobre critérios de geração de casos de teste para sistemas de tempo real em TBM. Concluída a revisão sistemática, foram organizados hierarquicamente, segundo a relação de inclusão de En-Nouaary [23], os trinta critérios de geração encontrados. Dos trinta critérios de geração da hierarquia, foram selecionados oito critérios que se adequam ao escopo desse mestrado e estes foram objeto do estudo experimental. Os critérios foram comparados observando sua capacidade de revelar falhas e tamanho do conjunto de casos de teste gerado. A comparação se deu dentro de um estudo experimental.

Para realizar o estudo experimental foi necessário:

- **Elaborar modelos de sistemas de tempo real**: Foram modelados seis sistemas de tempo real para que deles fossem gerados casos de teste segundo cada critério;
- **Elaborar um modelo de falhas**: Foi elaborado um modelo de falhas que descrevesse as possíveis falhas que os seis sistemas modelados podem apresentar;
- **Implementar os critérios de geração**: Era necessário um processo de geração de casos de teste para sistemas de tempo real em TBM baseado em critérios de geração. Os critérios de geração foram implementados e incorporados à ferramenta SYMBOLRT de forma a aproveitar o processo de geração de casos de teste previamente implementado em SYMBOLRT.

O estudo experimental permitiu concluir com 95% de confiança que realmente os critérios estudados apresentam diferentes capacidades de revelar falhas. Com isso, puderam-se tirar algumas conclusões subjetivas sobre em quais cenários é mais apropriado utilizar determinados critérios de geração. Como principais resultados podemos citar:

- Revisão sistemática: através da revisão sistemática foi possível a produção de vários artefatos como: documentos explicando as fontes de pesquisa, termos (palavras-chave) consultados e todas as etapas de seleção, lista integral de todos os artigos retornados pela pesquisa. Assim, o resultado da revisão sistemática pode ser utilizado para propiciar a possível replicação do estudo por parte de outro pesquisador além de, no futuro, poder encontrar possíveis novos critérios de geração que possam ser criados.

- Hierarquia de critérios: a hierarquia dos trinta critérios identificados na revisão sistemática auxilia o testador a avaliar qual critério de geração utilizar. A relação de inclusão da hierarquia permite que o testador conheça quais critérios de geração estão implicitamente sendo gerados a partir de um critério de geração mais alto na hierarquia de critérios.

- Extensão da ferramenta SYMBOLRT: para dar suporte à geração de casos de teste de sistemas de tempo real baseada em critérios de geração a ferramenta SYMBOLRT foi estendida. A implementação dos critérios permite não somente a execução do experimento conduzido nesse trabalho, mas também possibilita que outros sistemas de tempo real sejam testados através da ferramenta SYMBOLRT e os casos de teste possam ser gerados tanto através de propósito de teste quanto segundo algum dos critérios estudados nesse trabalho.

- Algoritmos de geração de casos de teste para cada critério: o presente trabalho também produziu como resultado os códigos abstratos da extração de casos de teste do modelo do sistema de tempo real para cada critério de geração. Assim, é possível implementar a geração de casos de teste em qualquer outra ferramenta que utilize modelo simbólico como formalismo para descrever STRs.

- Adaptação de critérios para STR: os critérios Definition Use Pair, All Du Paths e All Defs são originalmente utilizados para cobrir caminhos de definição e uso de variáveis,

mas não tratam a variável clock de forma diferente. Assim, esses critérios foram modificados de forma a tratar de maneira especial a variável clock e, por isso, se mostraram critérios muito eficientes para cobrir falhas devido a não cumprimento de restrições de tempo.

7.1 Trabalhos Futuros

Com a conclusão deste trabalho, surgem um conjunto de possíveis trabalhos futuros para continuação do mesmo. Dentre eles é possível destacar:

- **Relacionar tamanho de conjunto de casos de teste com elementos do modelo**: uma limitação desse trabalho se dá na interpretação do tamanho do conjunto de casos de teste. O tamanho do conjunto de casos de teste é investigado de forma subjetiva. Como trabalho futuro, poder-se-ia investigar qual a relação do tamanho do conjunto de casos de teste por critério de geração em função de elementos do modelo como, por exemplo, a quantidade de transições no modelo, quantidade de caminhos livres de *loop*, quantidade de *locations*, quantidade de guardas com restrição de tempo, etc;

- **Perfil dos casos de teste gerados pelos critérios de geração**: outro fator que o trabalho não aborda é o perfil dos casos de teste. O presente trabalho admite como fator negativo apenas o tamanho do conjunto de casos de teste que um critério de geração gera. No entanto, outras métricas podem ser avaliadas como redundância dos casos de teste, custo de execução de cada caso de teste, tempo de geração dos casos de teste, etc;

- **Repetir o experimento com aplicações reais implementadas**: o experimento conduzido nesse trabalho utilizou modelos de falhas como meio de observar quais critérios de geração possuem mais capacidade de revelar falhas. Como trabalho futuro, poder-se-ia repetir o experimento desse trabalho com aplicações de sistemas de tempo real implementadas e utilizar mutantes para introduzir faltas no código das aplicações com o intuito de verificar se as conclusões propostas nesse trabalho se repetem.

Bibliografia

[1] V. S. Alagar, O. Ormandjieva, and M. Zheng. Specification-based testing for real-time reactive systems. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, TOOLS '00, pages 25–, Washington, DC, USA, 2000. IEEE Computer Society.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, January 1999.

[4] Wilkerson L. Andrade. *Symbolic Model-Based Testing for Real-Time Systems*. PhD thesis, Federal University of Campina Grande, Apr 2011. `http://docs.computacao.ufcg.edu.br/posgraduacao/teses/2011/Tese_WilkersondeLucenaAndrade.pdf`.

[5] Wilkerson L. Andrade, Diego R. Almeida, Jeanderson B. Cândido, and Patrícia D. L. Machado. SYMBOLRT: A Tool for Symbolic Model-Based Test Case Generation for Real-Time Systems. In *19th Tools Session of the 3nd Brazilian Conference on Software: Theory and Practice (CBSoft 2012)*, pages 1–6, 2012. To Appear.

[6] Wilkerson L. Andrade and Patrícia D. L. Machado. Modeling and testing interruptions in reactive systems using symbolic models. In *SAST'08: Proc. of the 2nd Brazilian Work. on Systematic and Automated Software Testing*, pages 34–43, Porto Alegre, 2008. SBC.

[7] Wilkerson L. Andrade, Patrícia D. L. Machado, Everton L. G. Alves, and Diego R. Almeida. Test case generation of embedded real-time systems with interruptions for FreeRTOS. In *Formal Methods: Foundations and Applications*, volume 5902 of *LNCS*, pages 54–69. Springer, 2009.

[8] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 95–110, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 244–256, London, UK, UK, 1996. Springer-Verlag.

[10] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 25–, Washington, DC, USA, 1997. IEEE Computer Society.

[12] Laura Brandan Briones. *Theories for model-based testing: real-time and coverage*. PhD thesis, University of Twente, Enschede, March 2007. CTIT number: 07-97.

[13] Lionel C. Briand, Christiane M. Differding, and H. Dieter Rombach. Practical guidelines for measurement-based process improvement, 1996.

[14] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. O. Neto, and Patrícia D. L. Machado. LTSBT: A tool to generate and select functional test cases for embedded systems. In *SAC'08: Proc. of the 2008 ACM symposium on Applied computing*, volume 2, pages 1540–1544, New York, NY, USA, 2008. ACM Press.

[15] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based coverage-driven test suite generation for software product lines. In *Proceedings of*

the 14th international conference on Model driven engineering languages and systems, MODELS'11, pages 425–439, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In *TACAS'02*, volume 2280 of *LNCS*, pages 151–173. Springer, 2002.

[17] Duncan Clarke and Insup Lee. Automatic test generation for the analysis of a real-time system: Case study. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, pages 112–, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Trans. Softw. Eng.*, 33(8):558–574, August 2007.

[19] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 349–355, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *STTT*, pages 382–393, 2000.

[21] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[22] Ibrahim K. El-Far and James A. Whittaker. Model-based software testing. In *Encyclopedia on Software Engineering*. Wiley-Interscience, 2001.

[23] Abdeslam En-Nouaary. Test selection criteria for real-time systems modeled as timed input-output automata. *International Journal of Web Information Systems*, 3(4):279–292, 2007.

[24] Abdeslam En-Nouaary and Abdelwahab Hamou-Lhadj. A boundary checking technique for testing real-time systems modeled as timed input output automata (short paper). In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, pages 209–215, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Abdeslam En-Nouaary, Ferhat Khendek, and Rachida Dssouli. Fault coverage in testing real-time systems. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, pages 150–, Washington, DC, USA, 1999. IEEE Computer Society.

[26] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.

[27] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, June 1994.

[28] Anders Hessel. Model-based test case selection and generation for real-time systems, 2006.

[29] Anders Hessel. Model-based test case generation for real-time systems, 2007.

[30] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage estimation for symbolic model checking. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 300–305, New York, NY, USA, 1999. ACM.

[31] Carnegie-Mellon University. Software Engineering Institute, C.D. Locke, D.R. Vogel, L. Lucas, and J.B. Goodenough. *Generic Avionics Software Specification.* Technical report. Carnegie Mellon University, Software Engineering Institute, 1990.

[32] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[33] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, August 2005.

[34] Bertrand Jeannet, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Symbolic test selection based on approximate analysis. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 349–364, Berlin, Heidelberg, 2005. Springer-Verlag.

[35] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition*. AUERBACH, 3 edition, 2008.

[36] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2004.

[37] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.

[38] Phillip A. Laplante. *Real-Time System Design and Analysis*. John Wiley & Sons, 2004.

[39] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, pages 3–, Washington, DC, USA, 2002. IEEE Computer Society.

[40] William E. Lewis. *Software Testing and Continuous Quality Improvement, Third Edition*. Auerbach Publications, Boston, MA, USA, 2nd edition, 2008.

[41] L.Y. Liu and R.K. Shyamasundar. Static analysis of real-time distributed systems. *IEEE Transactions on Software Engineering*, 16:373–388, 1990.

[42] John D. McGregor and David A. Sykes. *A practical guide to testing object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[43] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 396–397, Washington, DC, USA, 2004. IEEE Computer Society.

[44] Brian Nielsen and Arne Skou. Test generation for time critical systems: Tool and case study. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ECRTS '01, pages 155–, Washington, DC, USA, 2001. IEEE Computer Society.

[45] David Owen, Dejan Desovski, and Bojan Cukic. Random testing of formal software models and induced coverage. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 20–27, New York, NY, USA, 2006. ACM.

[46] Ajitha Rajan. Coverage metrics to measure adequacy of black-box test suites. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 335–338, Washington, DC, USA, 2006. IEEE Computer Society.

[47] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *Proceedings of the Second International Conference on Integrated Formal Methods*, IFM '00, pages 338–357, London, UK, UK, 2000. Springer-Verlag.

[48] Ian Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[49] Mohammad Saeed Abou Trab, Bachar Alrouh, Steve Counsell, Rob M. Hierons, and George Ghinea. A multi-criteria decision making framework for real time model-based testing. In *Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques*, TAIC PART'10, pages 194–197, Berlin, Heidelberg, 2010. Springer-Verlag.

[50] Jan Tretmans. Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, December 1996.

[51] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[52] R. Van Solingen and E. Berghout. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.

[53] Reinhard Wilhelm, Sebastian Altmeyer, Claire BurguiÃ¨re, Daniel Grund, JÃ¶rg Herter, Jan Reineke, BjÃ¶rn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In Gilles Barthe and Manuel Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin / Heidelberg, 2010.

[54] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[55] Mao Zheng, Vasu Alagar, and Olga Ormandjieva. Automated generation of test suites from formal specifications of real-time reactive systems. *J. Syst. Softw.*, 81(2):286–304, February 2008.

Apêndice A

Revisão Sistemática

1. **Formalização da Questão**

   (a) **Foco da Questão:** Identificar trabalhos na literatura que abordem critérios de geração de casos de teste para sistemas de tempo real ao nível de modelo.

   (b) **Qualidade da Questão e Amplitude**

   - **Problema:** Critérios de geração são determinantes na capacidade de revelar falhas dos casos de teste gerados e consequentemente também são responsáveis pela qualidade dos testes. Sistemas de tempo real são sistemas que muitas vezes são críticos e, portanto, necessitam de testes mais efetivos. Teste exaustivo é um teste que é capaz de revelar um grande número de faltas, no entanto é um método muito caro e muitas vezes impraticável. É necessária uma investigação mais detalhada de quais critérios de geração são mais apropriados para sistemas de tempo real mantendo viável o tamanho do conjunto de casos de teste.
   - **Questão:** Quais trabalhos abordam critérios de geração de casos de teste para sistemas de tempo real ao nível de modelo.
   - **Palavras chave e Sinônimos:** Test selection criteria, test generation criteria, coverage criteria, metrics of test cases selection, stopping condition, real time systems, interruptions, model based testing.
   - **Intervenção:** Critérios de geração de casos de teste para sistemas de tempo real.

- **Controle:** Nenhum.
- **Efeito:** Identificação dos trabalhos relacionados a critérios de geração de casos de teste.
- **Medida de Saída:** Número de trabalhos identificados.
- **População:** Publicações sobre critérios de geração de casos de teste para sistemas de tempo real.
- **Aplicação:** Ferramentas de geração de casos de teste para sistemas de tempo real.
- **Design Experimental:** Nenhum método estatístico será aplicado.

2. **Seleção de Fontes**

   (a) **Definição de Critério de Seleção de Fontes:** Disponibilidade para consultar artigos científicos na web; presença de mecanismos de pesquisa usando palavras chave.

   (b) **Idiomas:** Inglês.

   (c) **Identificação de Fontes**

   - **Métodos de Pesquisa de Fontes:** Pesquisa através de engenhos de pesquisa web.
   - **String de Pesquisa:** (Test selection criteria OR test generation criteria OR coverage criteria OR metrics of test cases selection OR stopping condition) AND Real time systems AND model based testing.
   - **Lista de Fontes**
     - http://scholar.google.com.br/
     - http://citeseer.ist.psu.edu/
     - http://portal.acm.org/
     - http://www.springerlink.com
     - http://ieeexplore.ieee.org

   (d) **Seleção de Fontes Após Avaliação:** A princípio, todas as fontes listadas satisfazem o critério de qualidade.

(e) **Checagem de Referências:** Todas as fontes foram aprovadas.

3. **Seleção de Estudos**

   (a) **Definição de Estudo**

   - **Inclusão de Estudo e Definição de Critério de Exclusão:** Os trabalhos devem abordar questões referentes a critérios de geração de casos de teste para sistemas de tempo real ao nível de modelo podendo abordar questões como descrições, análises ou comparações entre critérios.
   - **Definição de Tipos de Estudo:** Todos os tipos de estudos relacionados ao tópico da pesquisa serão selecionados.
   - **Procedimentos para Seleção de Estudos:** A string de pesquisa deverá ser executada nas fontes selecionadas. Para obter um conjunto inicial de estudos, o título e o abstract de cada estudo obtido da pesquisa web será lido e avaliado de acordo com o critério de inclusão e exclusão. Para refinar o conjunto inicial, seus textos completos serão lidos.

   (b) **Execução da Seleção**

   - **Seleção Inicial de Estudos:** O conjunto de estudos iniciais pode ser encontrado no Anexo 1.
   - **Avaliação de Qualidade de Estudos:** Foram encontrados 11 estudos que se adequam com os critérios de inclusão e exclusão previamente definidos.
   - **Revisão de Seleção:** O resultado foi aceito.

4. **Extração de Informação**

   (a) **Inclusão de Informação e Definição de Critério de Exclusão:** A informação extraída dos estudos deve ser relacionada a critérios de geração de casos de teste para sistemas de tempo real ao nível de modelo.

   (b) **Extração de dados de formulários:** Não foram utilizados formulários.

   (c) **Execução de Extração**

   - **Extração de resultados objetivos**

i. **Identificação dos estudos:** [44], [8], [17], [23], [24], [29], [28], [36], [1], [49], [55].

ii. **Metodologia de estudo:** Os estudos identificados abordam questões de critérios de geração de casos de teste de sistemas de tempo real. Alguns deles fazem comparações entre os critérios enquanto outros não fazem.

iii. **Resultados de estudo:** Identificação de critérios de geração de casos de teste para sistemas de tempo real.

iv. **Problemas de estudo:** Nenhum dos estudos faz um estudo experimental do critério que aborda.

- **Extração de resultados subjetivos**

i. **Informação através de autores:** Não foi necessário.

ii. **Impressões gerais e abstrações:** Os estudos abordam critérios de geração e seleção de casos de teste para sistemas de tempo real. Alguns deles comparam os critérios, mas nenhum os compara com um estudo experimental utilizando apenas algum estudo de caso ou analisando critério de inclusão. Nenhum dos estudos levam em consideração eventos assíncronos (interrupções).

(d) **Resolução de Divergências Entre Revisores:** Não houve divergências.

5. **Sumarização de Resultados**

(a) **Resultados de cálculos estatísticos:** Cálculos estatísticos não foram usados.

(b) **Resultados apresentados em tabela:**

Tabela A.1: Tabela de Critérios.

Critério	Trabalhos
All Locations	[23], [36]
All Transitions	[23], [36]
All States	[55], [1], [23]
Metric-based coverage	[55], [1]
All paths	[23], [29], [28]

All traces	[23]
All inputs	[17], [23]
All outputs	[17], [23]
All-I/O-actions	[17], [36]
All clock valuations	[23]
All clock regions	[23]
All clock zones	[23], [49]
All clock guard bounds	[23], [24]
All time constraints	[23]
All clock bounds	[23]
All clock reset	[23]
Random Testing	[8]
Adaptive random testing	[8]
Search-based testing	[8]
Stable edge set criterion	[44]
Definition-use pair coverage	[29], [28]
Context coverage	[29], [28]
Ordered context coverage	[29], [28]
All Defs	[29], [28]
All-p-uses	[29], [28]
All-c-uses	[29], [28]
All-du-path	[29], [28]
All-input-bounds-approximate	[17]
All-timeout-actions	[17]

6. **Análise de sensibilidade:** Não foi aplicado.

7. **Plotting:** Não foi aplicado.

8. **Comentários Finais**

   - **Número de estudos:** Estudos encontrados na busca: 1.766; Estudos seleciona-

dos por título e abstract: 46; Estudos selecionados: 11.

- **Viés de busca, seleção e extração:** Não foi definido nenhum.
- **Viés publicados:** Não foi definido nenhum.
- **Validação entre revisores:** Não houve validação.
- **Aplicação dos resultados:** Os resultados mostram que além de haver poucos trabalhos que comparam critérios de geração de casos de teste não há comparações através de experimentos. Nota-se também que as comparações existentes não se preocupam em levantar critérios do estado da arte. Limitando-se apenas a comparar critérios do interesse dos autores.
- **Recomendações:** Nenhuma.

Apêndice B

Códigos Abstratos Dos Critérios de Geração

Código Fonte B.1: Código Abstrato do Critério All One Loop Paths

```
1  Set<TIOSTS> getAllOneLoopPathsTestCases(TIOSTS model) {
2    Set<TIOSTS> testSequences = new Set<TIOSTS>();
3    Location initialLocation = model.getInitialLocation();
4    Set<Transition> path = new Set<Transition>();
5    Set<List<Transition>> paths = new Set<Set<Transition>>();
6    int intCode = -1;
7    int choice = 0;
8    Decomposer.allOneLoopPathsDecompose(testSequences,
9                                        initialLocation, path,
10                                       paths, intCode, choice);
11   return testSequences;
12 }
13
14 allOneLoopPathsDecompose(Set<TIOSTS> testSequences,
15                          Location location, Set<Transition> path,
16                          Set<Set<Transition>> paths,
17                          int intCode, int choice) {
18   if(!paths.contains(path)){
19     paths.add(path);
20     addPath(path, testSequences);
21   }
```

```
22    Set<Transition> outGoingTransitions = location.getOutTransitions();
23    for (Transition outGoingTransition : outGoingTransitions) {
24      if (!madeALoop(outGoingTransition, path, intCode)) {
25        if (isBeginInterruption(outGoingTransition)) {
26          if (choice == 0) {
27            intCode = getIntCode(outGoingTransition);
28            choice = intCode;
29          } else {
30            continue;
31          }
32        }
33        if (isEndInterruption(outGoingTransition)) {
34          if (intCode == getChoice(outGoingTransition)) {
35            intCode = -1;
36          } else {
37            continue;
38          }
39        }
40        path.add(outGoingTransition);
41        allOneLoopPathsDecompose(testSequences,
42                                 outGoingTransition.getTarget(),
43                                 path, paths, intCode, choice);
44      }
45    }
46  }
```

Código Fonte B.2: Código Abstrato do Critério All Transitions

```
1  Set<TIOSTS> getAllTransitionsTestCases(TIOSTS model) {
2    Set<TIOSTS> testSequences = new Set<TIOSTS>();
3    Location initialLocation = model.getInitialLocation();
4    Set<Transition> path = new Set<Transition>();
5    Set<Transition> visitedTransitions = new Set<Transition>();
6    int intCode = -1;
7    int choice = 0;
8    Decomposer.allTransitionsDecompose(testSequences, initialLocation,
9                                       path, visitedTransitions,
10                                      intCode, choice);
```

```
11    return testSequences;
12  }
13
14  allTransitionsDecompose(Set<TIOSTS> testSequences,
15                          Location location, Set<Transition> path,
16                          Set<Transition> visitedTransitions,
17                          int intCode, int choice) {
18   if (finalLocation(location, visitedTransitions, intCode, choice)
19       AND path.size() != 0) {
20    addPath(path, testSequences);
21    return;
22   }
23   Set<Transition> outGoingTransitions = location.getOutTransitions();
24   for (Transition outGoingTransition : outGoingTransitions) {
25    if (!alreadyVisited(outGoingTransition, visitedTransitions, intCode)) {
26     if (isBeginInterruption(outGoingTransition)) {
27      if (choice == 0) {
28       intCode = getIntCode(outGoingTransition);
29       choice = intCode;
30      } else {
31       continue;
32      }
33     }
34     if (isEndInterruption(outGoingTransition)) {
35      if (intCode == getChoice(outGoingTransition)) {
36       intCode = -1;
37      } else {
38       continue;
39      }
40     }
41     path.add(outGoingTransition);
42     visitedTransitions.add(outGoingTransition);
43     allTransitionsDecompose(testSequences, outGoingTransition.getTarget(),
44                             path, visitedTransitions,
45                             intCode, choice);
46    }
47   }
```

```
48 }
```

Código Fonte B.3: Código Abstrato do Critério All Locations

```
1  Set<TIOSTS> getAllLocationsTestPurpose(TIOSTS model) {
2   Set<TIOSTS> testSequences = new ArrayList<TIOSTS>();
3   Location initialLocation = model.getInitialLocation();
4   Set<Transition> path = new Set<Transition>();
5   Set<Location> visitedLocations = new Set<Location>();
6   int intCode = -1;
7   int choice = 0;
8   Decomposer.allLocationsDecompose(testSequences, initialLocation,
9                                    path, visitedLocations,
10                                   intCode, choice);
11  return testSequences;
12 }
13
14 allLocationsDecompose(Set<TIOSTS> testSequences, Location location,
15                       Set<Transition> path,
16                       Set<Location> visitedLocations,
17                       int intCode, int choice) {
18  if (finalLocation(location, visitedLocations) AND path.size() != 0) {
19   addPath(path, testSequences);
20   return;
21  }
22  Set<Transition> outGoingTransitions = location.getOutTransitions();
23  for (Transition outGoingTransition : outGoingTransitions) {
24   if (!visitedLocations.contains(outGoingTransition.getTarget())) {
25    if (isBeginInterruption(outGoingTransition)) {
26     if (choice == 0) {
27      intCode = getIntCode(outGoingTransition);
28      choice = intCode;
29     } else {
30      continue;
31     }
32   }
33   if (isEndInterruption(outGoingTransition)) {
34    if (intCode == getChoice(outGoingTransition)) {
```

```
35        intCode = -1;
36      } else {
37        continue;
38      }
39    }
40    path.add(outGoingTransition);
41    visitedLocations.add(location);
42    allLocationsDecompose(testSequences,
43                          outGoingTransition.getTarget(),
44                          path,    visitedLocations,
45                          intCode, choice);
46    }
47  }
48 }
```

Código Fonte B.4: Código Abstrato do Critério All Clock Zones

```
1  Set<TIOSTS> getAllClockZonesTestCases(TIOSTS model) {
2   Set<TIOSTS> allTransitions = getAllTransitionsTestCases(model);
3   Set<TIOSTS> result = Decomposer.allClockZonesDecompose(allTransitions);
4   return result;
5  }
6
7  Set<TIOSTS> allClockZonesDecompose(List<TIOSTS> allTransitions) {
8   Set<TIOSTS> result = new Set<TIOSTS>();
9   for (TIOSTS testSequences : allTransitions) {
10   if(thereIsClockGuard(testSequences)){
11    result.add(testSequences);
12   }
13  }
14  return result;
15 }
```

Código Fonte B.5: Código Abstrato do Critério All Clock Reset

```
1  Set<TIOSTS> getAllClockResetTestCases(TIOSTS model) {
2   Set<TIOSTS> testSequences = new Set<TIOSTS>();
3   Set<Transition> resetingTransitions =
4                   getResetingClockTransitions(model);
```

```
5   Location initialLocation = model.getInitialLocation();
6   Set<Transition> path = new Set<Transition>();
7   int intCode = -1;
8   int choice = 0;
9   Decomposer.allClockResetDecompose(testSequences, resetingTransitions,
10                                    initialLocation, path,
11                                    intCode, choice);
12  return testSequences;
13 }
14
15 allClockResetDecompose(Set<TIOSTS> testSequences,
16                         Set<Transition> resetingTransitions,
17                         Location location, Set<Transition> path,
18                         int intCode, int choice) {
19  if (finalLocation(location, path, intCode, choice) AND
20      path.size() != 0) {
21   if(containsTransitions(resetingTransitions, path)){
22    addPath(path, testSequences, false);
23   }
24   return;
25  }
26  Set<Transition> outGoingTransitions = location.getOutTransitions();
27  for (Transition outGoingTransition : outGoingTransitions) {
28   if (!alreadyVisited(outGoingTransition, path, intCode)) {
29    if (isBeginInterruption(outGoingTransition)) {
30     if (choice == 0) {
31      intCode = getIntCode(outGoingTransition);
32      choice = intCode;
33     } else {
34      continue;
35     }
36    }
37    if (isEndInterruption(outGoingTransition)) {
38     if (intCode == getChoice(outGoingTransition)) {
39      intCode = -1;
40       } else {
41        continue;
```

```
42          }
43        }
44        path.add(outGoingTransition);
45        allClockResetDecompose(testSequences, resetingTransitions,
46                               outGoingTransition.getTarget(),
47                               path, intCode, choice);
48    }
49  }
50 }
```

Código Fonte B.6: Código Abstrato do Procedimento comum aos critérios Definition-Use Pair, All Du Paths e All Defs

```
1  defUseDecompose(Set<TIOSTS> testSequences,
2                  Transition resettingTransition,
3                  Location location, Set<Transition> path,
4                  Set<Location> visitedLocations, String clock,
5                  boolean usedClock, boolean afterResettingTrasition,
6                  boolean allDuPaths, boolean allDefs,
7                  int intCode, int choice) {
8   if(finishIt){
9    return;
10  }
11  if (afterResettingTrasition AND usedClock) {
12   addPath(path, testSequences);
13   if(allDuPaths){
14    return;
15   }
16   if(allDefs){
17    finishIt = true;
18    return;
19   }
20   usedClock = false;
21  }
22  if (finalLocation(location, path, intCode, choice)) {
23   return;
24  }
25  Set<Transition> outGoingTransitions = location.getOutTransitions();
```

```
26  for (Transition outGoingTransition : outGoingTransitions) {
27   if (!alreadyVisited(outGoingTransition, path, intCode)) {
28    if (isBeginInterruption(outGoingTransition)) {
29     if (choice == 0) {
30      intCode = getIntCode(outGoingTransition);
31      choice = intCode;
32     } else {
33      continue;
34     }
35    }
36    if (isEndInterruption(outGoingTransition)) {
37     if (intCode == getChoice(outGoingTransition)) {
38      intCode = -1;
39     } else {
40      continue;
41     }
42    }
43    boolean usedCl = isClockUseTransition(clock, outGoingTransition) AND
44                                          !outGoingTransition.equals(
45                                           resettingTransition);
46    path.add(outGoingTransition);
47    boolean afterResetTrasition = afterResettingTrasition  OR
48                                  outGoingTransition.equals(
49                                  resettingTransition);
50    visitedLocations.add(location);
51    defUseDecompose(testSequences, resettingTransition,
52                    outGoingTransition.getTarget(), path,
53                    visitedLocations, clock, usedCl,
54                    afterResetTrasition, allDuPaths,
55                    allDefs, intCode, choice);
56   }
57  }
58 }
```

Código Fonte B.7: Código Abstrato do Critério Definition-Use Pair

```
1 Set<TIOSTS> getDefinitionUsePairTestCases(TIOSTS model) {
2  Set<TIOSTS> testSequences = new Set<TIOSTS>();
```

```
3   Set<Transition> resetingTransitions =
4                   getResetingClockTransitions(model);
5   Decomposer.definitionUsePairDecompose(model,
6                                         testSequences,
7                                         resetingTransitions);
8   return testSequences;
9  }
10
11 definitionUsePairDecompose(TIOSTS model, Set<TIOSTS> testSequences,
12                              Set<Transition> resettingTransitions){
13  Location initialLocation = model.getInitialLocation();
14  Set<Transition> path = new Set<Transition>();
15  Set<Location> visitedLocations = new Set<Location>();
16  for (Transition resettingTransition : resettingTransitions) {
17   Set<String> clocks = resettingTransition.getResetedClocks();
18   for (String clock : clocks) {
19    finishIt = false;
20    defUseDecompose(testSequences, resettingTransition,
21                    initialLocation, path, visitedLocations,
22                    clock, false, false, false, false, -1, 0);
23   }
24  }
25 }
```

Código Fonte B.8: Código Abstrato do Critério All Du Paths

```
1  Set<TIOSTS> getAllDuPathsTestCases(TIOSTS model) {
2   Set<TIOSTS> testSequences = new Set<TIOSTS>();
3   Set<Transition> resetingTransitions =
4                   getResetingClockTransitions(model);
5   Decomposer.allDuPathsDecompose(model, testSequences,
6                                  resetingTransitions);
7   return testSequences;
8  }
9
10 allDuPathsDecompose(TIOSTS model, Set<TIOSTS> testSequences,
11                     Set<Transition> resettingTransitions) {
12  Location initialLocation = model.getInitialLocation();
```

```
13   Set<Transition> path = new Set<Transition>();
14   Set<Location> visitedLocations = new Set<Location>();
15   for (Transition resettingTransition : resettingTransitions) {
16    Set<String> clocks = resettingTransition.getResetedClocks();
17    for (String clock : clocks) {
18     finishIt = false;
19     defUseDecompose(testSequences, resettingTransition,
20                     initialLocation, path, visitedLocations,
21                     clock, false, false, true, false, -1, 0);
22    }
23   }
24 }
```

Código Fonte B.9: Código Abstrato do Critério All Defs

```
1  Set<TIOSTS> getAllDefsTestCases(TIOSTS model) {
2   Set<TIOSTS> testSequences = new Set<TIOSTS>();
3   Set<Transition> resetingTransitions =
4                   getResetingClockTransitions(model);
5   Decomposer.allDefsDecompose(model, testSequences,
6                               resetingTransitions);
7   return testSequences;
8  }
9
10 allDefsDecompose(TIOSTS model, Set<TIOSTS> testSequences,
11                  Set<Transition> resettingTransitions) {
12  Location initialLocation = model.getInitialLocation();
13  Set<Transition> path = new Set<Transition>();
14  Set<Location> visitedLocations = new Set<Location>();
15  for (Transition resettingTransition : resettingTransitions) {
16   Set<String> clocks = resettingTransition.getResetedClocks();
17   for (String clock : clocks) {
18    finishIt = false;
19    defUseDecompose(testSequences, resettingTransition,
20                    initialLocation, path,
21                    visitedLocations, clock,
22                    false, false, false, true, -1, 0);
23   }
```

```
24    }
25 }
```

Apêndice C

Modelos de Sistemas de Tempo Real

C.1 Sistema de Alarme

O Sistema de Alarme é um sistema de monitoramento de tempo real cujo objetivo é monitorar sensores para detectar a presença de intrusos em um prédio.

Este sistema usa diferentes tipos de sensores incluindo detectores de movimentos em salas individuais, sensores de janela, que detectam a quebra de uma janela e detectores de porta, que detecta a abertura de portas. Há 50 sensores de janela, 30 sensores de porta, e 200 sensores de movimento. Quando um sensor indica a presença de um intruso, o sistema automaticamente chama a polícia e, com um sintetizador de voz, reporta a posição do alarme. Adicionalmente, o sistema liga as luzes ao redor da área ativada pelo sensor e aciona um alarme audível. O sistema é normalmente alimentado por um sistema central de energia, mas é equipado com uma bateria de reserva. A perda de energia é detectada por um monitor de circuito que monitora a tensão central. O sistema aciona automaticamente a energia reserva (bateria) quando a queda de voltagem é detectada. As restrições de tempo do sistema de alarme são descritas na Tabela C.1.

Tabela C.1: Restrições de Tempo.

Estimulo/Resposta	**Restrições de Tempo**
Interrupção de falha de energia	A mudança para a bateria de reserva deve ser executada em até 50 ms.

Alarme audível	O alarme audível deve ser acionado em 0,5 segundo após a detecção de um intruso.
Sintetizador de voz	Uma mensagem sintetizada deve estar disponível em 3 segundos após um alarme audível ter sido acionado.
Comunicações	A chamada à polícia deve ser iniciada em 1 segundo após a mensagem ser sintetizada.
Luzes	As luzes devem ser ligadas em 0,5 segundo após a chamada à polícia.

Considerando a arquitetura do sistema, cada funcionalidade é alocada a um processo concorrente assim como cada tipo de sensor é alocado a um processo. Há um sistema dirigido a interrupção para tratar a falha e mudança de fonte de alimentação, um sistema de comunicação, um sintetizador de voz, um sistema de alarme audível, e um sistema de acionamento de iluminação para ligar as luzes ao redor do sensor. A arquitetura do sistema é descrito na Figura C.1. As setas rotuladas indicam o fluxo de dados entre processos e as notas associadas com os processos indicam qual processo ou ator causa a interrupção.

A Figura C.2 ilustra o modelo TIOSTS do sistema de alarme. Para efeitos de visualização, os rótulos das transições foram mapeados e seus significados podem ser encontrados na Tabela C.2.

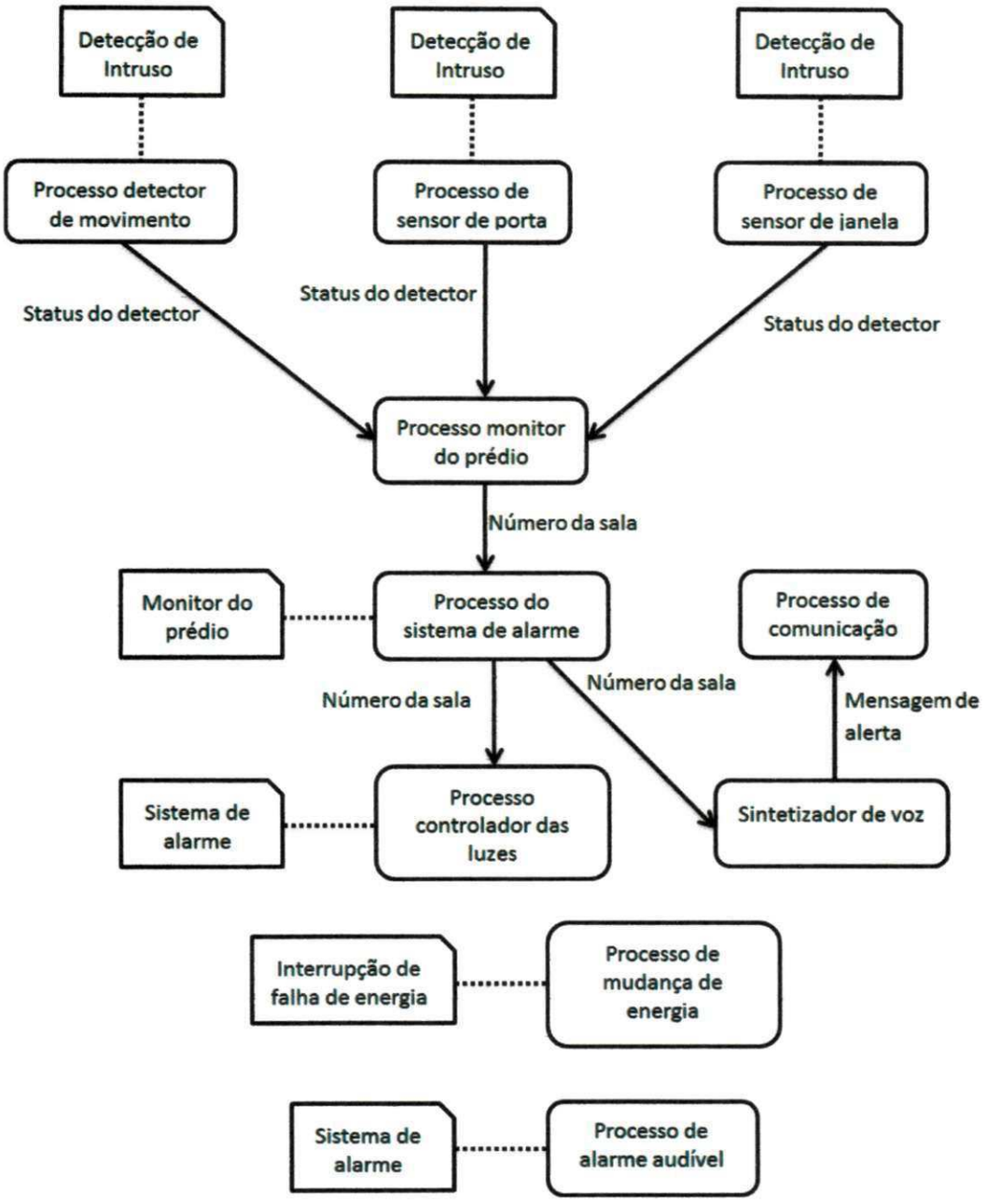


Figura C.1: Arquitetura do sistema de Alarme

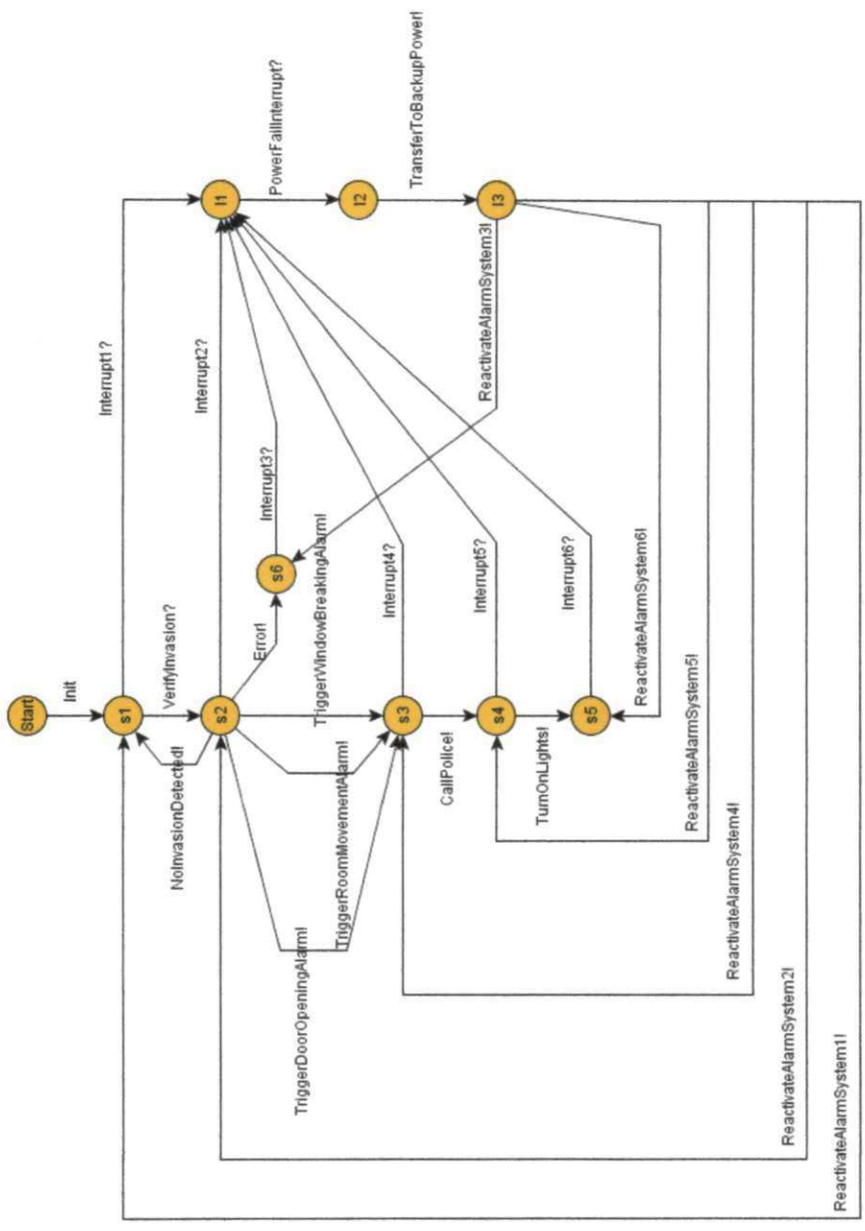


Figura C.2: Modelo TIOSTS do Sistema de Alarme

Tabela C.2: Mapa de transições do modelo do sistema de alarme.

Código Init	**Transição** Init
Interrupt?1	intCode = 1 AND choice = 0 Interrupt?(intCode) choice := intCode
ReactivateAlarmSystem!1	choice = 1 AND interruptionClock <= 50 ReactivateAlarmSystem!() delayable
Interrupt?2	intCode = 2 AND choice = 0 Interrupt?(intCode) choice := intCode
ReactivateAlarmSystem!2	choice = 2 AND interruptionClock <= 50 ReactivateAlarmSystem!() delayable
Interrupt?3	intCode = 3 AND choice = 0 Interrupt?(intCode) choice := intCode
ReactivateAlarmSystem!3	choice = 3 AND interruptionClock <= 50 ReactivateAlarmSystem!() delayable
Interrupt?4	intCode = 4 AND choice = 0 Interrupt?(intCode) choice := intCode
ReactivateAlarmSystem!4	choice = 4 AND interruptionClock <= 50 ReactivateAlarmSystem!() delayable
Interrupt?5	intCode = 5 AND choice = 0 Interrupt?(intCode) choice := intCode

ReactivateAlarmSystem!5	choice = 5 AND interruptionClock <= 50 ReactivateAlarmSystem!() delayable
Interrupt?6	intCode = 6 AND choice = 0 Interrupt?(intCode) choice := intCode
ReactivateAlarmSystem!6	choice = 6 AND interruptionClock <= 50 ReactivateAlarmSystem!() delayable
PowerFailInterrupt?	PowerFailInterrupt?() interruptionClock := 0
TransferToBackupPower!	TransferToBackupPower!()
VerifyInvasion?	VerifyInvasion?(roomNumber,invType) room := roomNumber | invasionType := invType | clock := 0
NoInvasionDetected!	invasionType = 0 NoInvasionDetected!() delayable
TriggerWindowBreakingAlarm!	invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500 TriggerWindowBreakingAlarm!(windowNumber) clock := 0 delayable

TriggerRoomMovementAlarm!	invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500 TriggerRoomMovementAlarm!(movSensorNumber) clock := 0 delayable
TriggerDoorOpeningAlarm!	invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500 TriggerDoorOpeningAlarm!(doorNumber) clock := 0 delayable
Error!	invasionType > 280 Error!() delayable
CallPolice!	room = roomNumber AND clock <= 4000 CallPolice!(roomNumber) clock := 0 delayable
TurnOnLights!	room = roomNumber AND clock <= 500 TurnOnLights!(roomNumber) delayable

C.2 Sistema de Ataque de um Caça

Aircraft Attack System é um sistema de tempo real hipotético baseado nas especificações do Generic Avionics Software Specification [31] cujo objetivo é executar ataques a alvos no chão.

O sistema é composto de:

- **Controles e Displays:** controles e displays incluem o head-up display (HUD), display

multe propósito (MPD *Multi-Purpose Display*), *keyset* do piloto, e o *hands-on throttle and stick* (HOTAS). Esses controles e displays constituem a interface homem máquina entre o piloto e a aeronave. Pelo menos dois displays simultâneos são normalmente disponíveis. O display HUD, que é uma display transparente com imagens geradas por computador sobrepondo a visão fora da janela, mostra dados de voo (velocidade do ar, altitude, horizonte, etc.) bem como a mira da arma. O display MPD geralmente mostra situações táticas, incluindo dados de ameaça, superpostas ao mapa digital de movimento. A Figura C.3 mostra os controles e displays de aeronaves similares a descrita nessa especificação.

- **Sensores:** Aeronaves de ataque geralmente carregam três tipos de sensores:
  - **Sensores de navegação:** incluem Computador de Dados Aéreos (ADC *Air Data Computer*) que provê altímetro barométrico e dados sobre pressão, Sistema de Navegação Inercial (INS *Inertial Navigation System*) que provê posição e velocidade da aeronave, Radar de Altitude (RALT *Radar Altimeter*) que provê medida de altitude acima do solo.
  - **Radar de Recepção de Ameaças (RWR *Radar Warning Reciver*):** é um sensor de alerta de ameaças que adverte ao piloto sobre energia de radar hostil, tal como de míssil guiado por calor.
  - **Sensor de alvo:** tal como radar, provê dados de alcance e alvo (incluindo taxas) de precisão suficiente para rastrear alvos móveis.

É possível realizar ataques via HUD ou por radar. Independente da forma de ataque é necessário que o piloto escolha o tipo de arma que será utilizada. Caso seja míssil, é necessário que o piloto informe a quantidade de mísseis a serem lançados. Caso o ataque seja por HUD, o HUD mostrará ao piloto os possíveis alvos visíveis então o piloto deverá posicionar o alvo desejado dentro de um retângulo que aparecerá no HUD. Após isso o sensor de alvo calculará a posição do alvo (distância, angulação, etc.). Em seguida, calculará a trajetória incluindo o tempo para atingir o alvo e a margem de erro. Essa trajetória será usada para cada munição que for lançada. Alternativamente, é possível realizar ataque a alvos no solo também por radar. O procedimento é idêntico à exceção que o piloto não

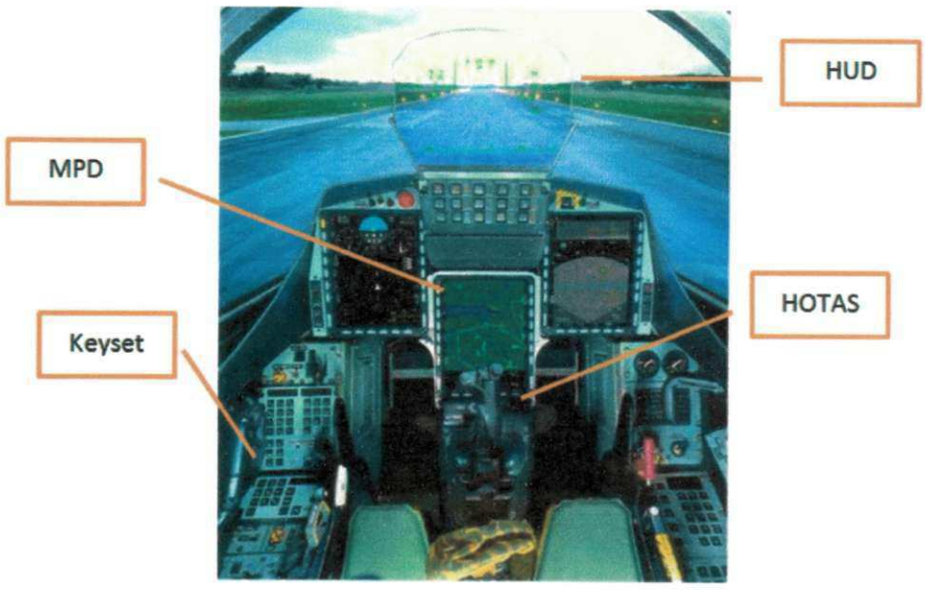


Figura C.3: Interface homem máquina entre piloto e aeronave

seleciona o alvo. Este é escolhido de forma automática e a cada 4 segundos a aeronave recalcula a posição do alvo no HUD. A Figura C.4 mostra um exemplo de ataque por uma aeronave a um alvo terrestre.

A qualquer momento a aeronave pode ser vítima de um ataque. Caso isso ocorra a aeronave deve descobrir qual é o tipo de ameaça para que possa delegar a responsabilidade de resposta. Como exemplo podemos citar que a aeronave pode estar sendo alvo de uma outra aeronave, então é responsabilidade do piloto realizar manobras para que possa sair da mira da aeronave inimiga ou a aeronave pode estar sendo vítima de um ataque de míssil, então a aeronave ativa armas de contra ataque ao míssil.

As restrições de tempo do sistema de alarme são descritas na Tabela C.3.

Tabela C.3: Restrições de Tempo.

Estimulo/Resposta	**Restrições de Tempo**
Calcular posição	O calculo da posição do alvo deve ser realizado em até 10 ms.
Calcular trajetória	O calculo da trajetória do alvo deve ser realizado em até 70 ms.
Lançar arma	O lançamento da munição deve ocorrer em até 50 ms.

Posicionamento do alvo no HUD pelo radar	O radar deve posicionar o alvo no HUD em até 40 ms.
Reposicionamento do alvo no HUD pelo radar	Enquanto o piloto não disparar a arma o radar deve reposicionar o alvo no HUD em intervalos de tempo de até 400 ms.
Delegar responsabilidade de ataque	O sistema tem até 1 ms para delegar a responsabilidade de resposta a uma ameaça.
Confirmação de responsabilidade	O sistema tem até 3 ms para receber a confirmação da delegação de responsabilidade.

A Figura C.5 ilustra o modelo TIOSTS do sistema de ataque de um caça. Para efeitos de visualização, os rótulos das transições foram mapeados e seus significados podem ser encontrados na Tabela C.4.

Tabela C.4: Mapa de transições do modelo do ataque de um caça.

Código Init	**Transição** Init
Interrupt?1	intCode = 1 AND choice = 0 Interrupt?(intCode) choice := intCode ‖ threatClock := 0
ReactivateSystem!1	choice = 1 ReactivateSystem!() delayable
Interrupt?2	intCode = 2 AND choice = 0 Interrupt?(intCode) choice := intCode ‖ threatClock := 0
ReactivateSystem!2	choice = 2 ReactivateSystem!() delayable

Interrupt?3	intCode = 3 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!3	choice = 3 ReactivateSystem!() delayable
Interrupt?4	intCode = 4 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!4	choice = 4 ReactivateSystem!() delayable
Interrupt?5	intCode = 5 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!5	choice = 5 ReactivateSystem!() delayable
Interrupt?6	intCode = 6 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!6	choice = 6 ReactivateSystem!() delayable
Interrupt?7	intCode = 7 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!7	choice = 7 ReactivateSystem!() delayable

Interrupt?8	intCode = 8 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!8	choice = 8 ReactivateSystem!() delayable
Interrupt?9	intCode = 9 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!9	choice = 9 ReactivateSystem!() delayable
Interrupt?10	intCode = 10 AND choice = 0 Interrupt?(intCode) choice := intCode | threatClock := 0
ReactivateSystem!10	choice = 10 ReactivateSystem!() delayable
DelegateResponse!	threat != 0 AND threatClock <= 1 DelegateResponse!(threat) threatClock := 0 delayable
ResponseCompleted!	threatClock <= 3 ResponseCompleted!() delayable
Init	Init

selectWeapon?	selectWeapon?(type, qte) weaponType := type, weaponQte := qte
attackByHUD?	attackByHUD?()
positionHUDReticle?	positionHUDReticle?(HUDReticlePosition) reticlePosition := HUDReticlePosition
atack?	atack?() clock := 0
attackByRadar?	attackByRadar?() clock := 0
trackTarget!	clock <= 40 trackTarget!() clock := 0 delayable
retrackTarget!	clock <= 400 retrackTarget!() clock := 0 delayable
getPosition!	clock <= 10 getPosition!(position) clock := 0 delayable

calculateTrajectory!	clock <= 70 calculateTrajectory!(timeToGo, timeOfFall, downRangeError, crossRangeError) clock := 0 delayable
releaseWeapon!	clock <= 50 releaseWeapon!() weaponQte := weaponQte - 1 delayable
repeat!	weaponQte > 0 repeat!() clock := 0 delayable

C.3 Protocolo de Áudio da Philips

O Protocolo de Áudio da Philips [9] é um protocolo dedicado para trocas de mensagens de controle de dispositivos de áudio e vídeo. Consequentemente, o protocolo deve ser simples e barato de implementar. Os dados são codificados em Manchester, e transferidos em um barramento único compartilhado. O protocolo admite uma tolerância de $\pm$ nas restrições de tempo sem que haja prejuízo na decodificação do sinal de transmissão. Outra característica do protocolo é a detecção de colisão no envio de bits por dois ou mais *senders*. O protocolo considera que um ou mais *senders* podem enviar bits pelo barramento e há um ou mais *receivers* que podem receber os bits através desse mesmo barramento. Um *sender* é equipado com um módulo capaz de codificar e transmitir os dados pelo barramento e um *receiver* é equipado com um módulo capaz de receber e decodificar os dados. Um overview do protocolo pode ser visto na Figura C.6.

Figura C.4: Ataque a um alvo terrestre

O *sender* obtem o stream de bits para ser transmitido via três ações: in0, in1, e empty, respectivamente representando um bit zero, um bit um, e um delimitador de fim de mensagem. O *sender* codifica estes bits, e usa as ações up e dn para conduzir a voltagem do barramento para alta e baixa respectivamente.

O barramento trabalha como uma lógica OU, portanto, sempre que um *sender* envia um bit 1, o arramento vai ser 1 mesmo que outros *senders* enviem um bit zero. Um *sender* pode detectar a colisão verificando se a voltagem do barramento está baixa quando o *sender* tiver enviado um bit zero. A ação isUp é usada para esse propósito. Se uma colisão é detectada a camada mais alta do protocolo será informada pela ação *coll*. O *receiver* captura os bits pela ação VUP. Para decodificar o sinal usando apenas uma ação, as mensagens devem começar com o valor 1 e ter tamanho ímpar.

Usando codificação Manchester, como ilustrado na Figura C.7, o eixo do tempo é dividido em bit slots de tamanhos iguais. Em cada bit slot um bit pode ser enviado. Um bit slot é dividido por sua vez em dois intervalos de tamanhos iguais. O valor lógico zero é representado por uma voltagem baixa no barramento no primeiro intervalo do bit slot e uma voltagem alta no segundo intervalo. O valor lógico um é representado por uma voltagem alta no barramento no primeiro intervalo do bit slot e uma voltagem baixa no segundo intervalo.

Um bit slot no protocolo da Philips é $888\mu s$. No modelo é utilizado quartis de bit slots

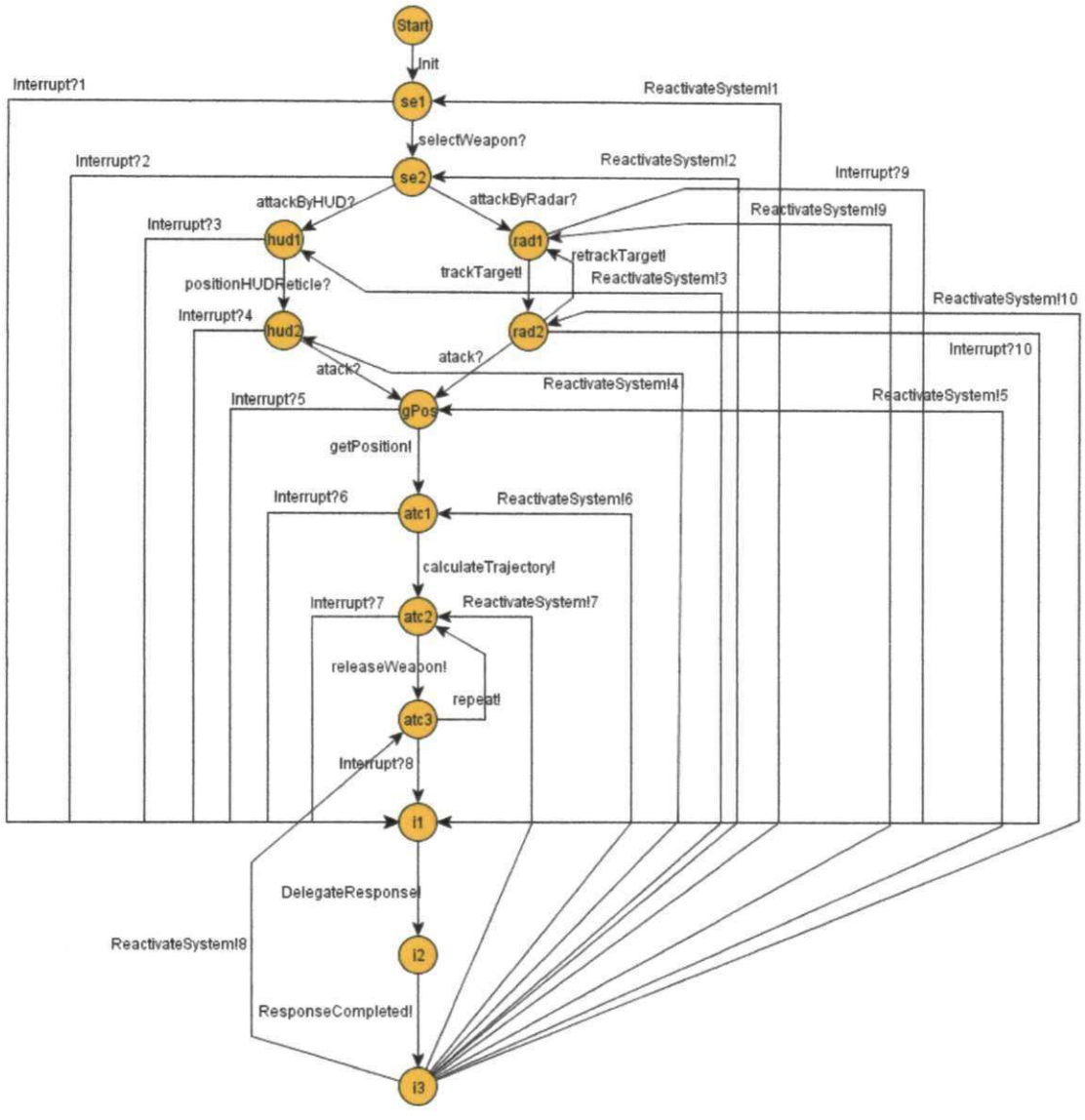


Figura C.5: Modelo TIOSTS do Ataque de um Caça

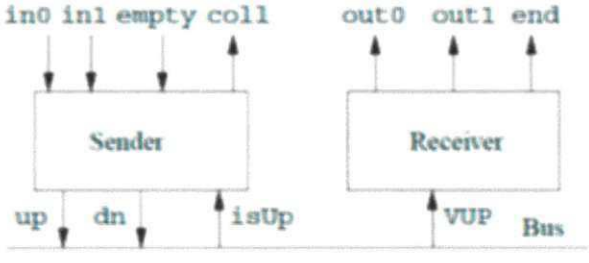


Figura C.6: Overview do Philips Audio Protocol

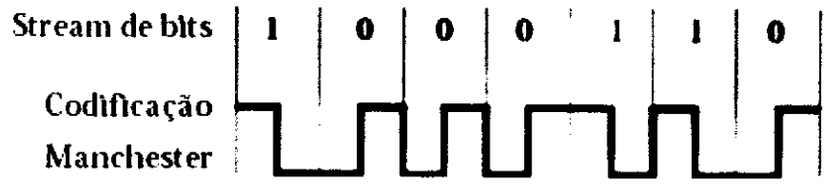


Figura C.7: Codificação Manchester do stream de bits 1000110.

($222\mu s$), denotado por q. As constantes básicas e os níveis de tolerâncias utilizadas no modelo são sumarizadas na Tabela C.5. No protocolo de áudio da Philips, um *sender* é capaz de codificar um bit enquanto envia outro. Além disso, para detectar colisões, o barramento deve ser amostrados 'aproximadamente' em três pontos específicos: depois de um quartil de um bit slot, depois de começar um sinal baixo, e logo após ajustar o barramento para sinal alto.

Tabela C.5: Constantes utilizadas na especificação do Philips Audio Protocol.

Símbolo	**Valor**	**Significado**			
q	2220	um quartil de um bit slot ($222\mu s$)			
d	220	detecção exatamente antes do up ($20\mu s$)			
g	220	em torno de 25% e 75% do bit slot ($22\mu s$)			
w	80000	estado de silêncio ($22ms$)			
t	0.05	tolerância (5%)			
A1min	2000	q - g	A1max	2440	q + g
A2min	6440	3q - g	A2max	6880	3q + g
Q2	4440	2q	Q2minD	4018	2q(1-t) - d
Q2min	4218	2q(1-t)	Q2max	4662	2q(1+t)
Q3min	6327	3q(1-t)	Q3max	6993	3q(1+t)
Q4	8880	4q	Q4minD	8236	4q(1-t) - d
Q4min	8436	4q(1-t)	Q4max	9324	4q(1+t)
Q5min	10545	5q(1-t)	Q5max	11655	5q(1+t)
Q7min	14763	7q(1-t)	Q7max	16317	7q(1+t)
Q9min	18981	9q(1-t)	Q9max	20979	5q(1+t)

A Figura C.8 ilustra o modelo TIOSTS do protocolo de áudio da philips segundo [44].

C.4 Modelos de Falhas

C.4.1 Sistema de Alarme

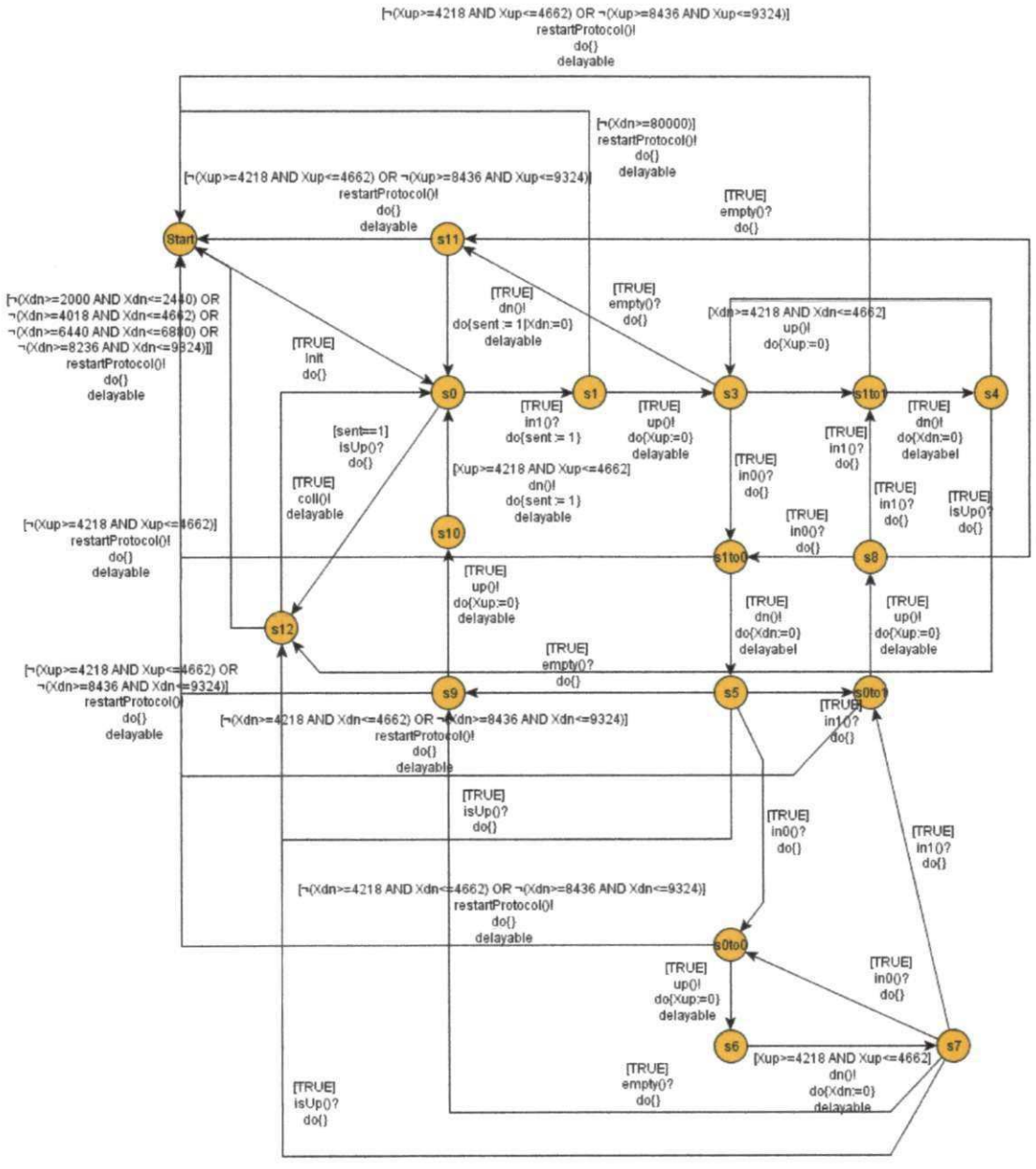


Figura C.8: Modelo TIOSTS do Protocolo de Áudio da Philips

  - **Tipo de Falha:** Falha de Mudança de Estado.

- **Falha 2:** O sensor de movimento é acionado sem que alguém tenha invadido o prédio;

  - **Tipo de Falha:** Falha de Mudança de Estado.

- **Falha 3:** O sensor de porta é acionado sem que alguém tenha invadido o prédio;

  - **Tipo de Falha:** Falha de Mudança de Estado.

C.4.2 Sistema de Alarme com Tratamento a Queda de Energia

- **Falha 4:** O sensor de porta não detecta a abertura de uma porta no momento em que há uma queda de energia;

  - **Restrição de Tempo:** a transferência de energia pode levar mais tempo do que no necessário para o sensor de porta informar ao monitor de processos do prédio a abertura de uma porta.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 5:** Em uma queda de energia o sistema não transfere a alimentação para a bateria de reserva em tempo hábil e o sistema para de funcionar;

  - **Restrição de Tempo:** a transferência de energia leva mais do que 50 milissegundos.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 6:** Após a transferência de energia o sistema não retorna ao mesmo ponto de execução em que estava;

  - **Tipo de Falha:** Falha de Mudança de Estado.

- **Falha 7:** O sensor de movimento não detecta a invasão de um intruso no momento em que há uma queda de energia;

  - **Restrição de Tempo:** a transferência de energia pode levar mais tempo do que no necessário para o sensor informar ao monitor de processos do prédio a detecção de movimento.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 8:** O sensor de janela não detecta a quebra de uma janela no momento em que há uma queda de energia;

  - **Restrição de Tempo:** a transferência de energia pode levar mais tempo do que no necessário para o sensor de janela informar ao monitor de processos do prédio a quebra de uma janela.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 9:** Quando um sensor de movimento detecta a presença de um intruso após ocorrida uma queda de energia, o sistema não informa a sala correta;

  - **Tipo de Falha:** Falha de Mudança de Estado.

- **Falha 10:** Após a transferência de energia o sistema soa o alarme sem que o prédio tenha sido invadido;

  - **Tipo de Falha:** Falha de Mudança de Estado.

C.4.3 Sistema de Ataque de um Caça

- **Falha 11:** A trajetória pode não ser calculada no tempo devido.

  - **Restrição de Tempo:** O tempo necessário para calcular a trajetória pode exceder o admissível para manter a sua precisão.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 12:** O míssil pode não ser lançado.

  - **Tipo de Falha:** Falha de Mudança de Estado.

- **Falha 13:** O radar detecta um alvo terrestre falso.

  - **Tipo de Falha:** Falha de Mudança de Estado.

C.4.4 Sistema de Ataque de um Caça com Tratamento a Ataque Inimigo

- **Falha 14:** O radar pode não reconhecer um alvo terrestre após o tratamento de uma ameaça.
  - **Restrição de Tempo:** O sistema leva muito tempo durante o tratamento da ameaça e o sistema não reconhece o alvo.
  - **Tipo de Falha:** Falha de Restrição de Tempo.
- **Falha 15:** A trajetória pode não ser calculada no tempo devido após o tratamento de uma ameaça.
  - **Restrição de Tempo:** O tempo necessário para calcular a trajetória pode exceder o admissível para manter a sua precisão devido a o atraso no tratamento de uma ameaça.
  - **Tipo de Falha:** Falha de Restrição de Tempo.
- **Falha 16:** A aeronave pode não detectar a ocorrência de um ataque inimigo.
  - **Tipo de Falha:** Falha de Mudança de Estado.
- **Falha 17:** Um ataque inimigo pode não ser detectado durante o ataque por radar.
  - **Tipo de Falha:** Falha de Mudança de Estado.
- **Falha 18:** Após o tratamento de uma ameaça a aeronave pode não manter os dados de posicionamento do alvo terrestre.
  - **Tipo de Falha:** Falha de Mudança de Estado.
- **Falha 19:** Há perda na precisão do alvo quando ocorrido um tratamento de uma ameaça à aeronave após um ataque por HUD.
  - **Restrição de Tempo:** O sistema leva muito tempo durante o tratamento da ameaça e a posição perde precisão.
  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 20:** Há perda na precisão do alvo quando ocorrido um tratamento de uma ameaça à aeronave após um ataque por radar.
  - **Restrição de Tempo:** O sistema leva muito tempo durante o tratamento da ameaça e a posição perde precisão.
  - **Tipo de Falha:** Falha de Restrição de Tempo.

C.4.5 Protocolo de Áudio da Philips com e sem Tratamento de Erro de Protocolo

- **Falha 21:** Uma colisão não é detectada.
  - **Tipo de Falha:** Falha de Mudança de Estado.
- **Falha 22:** O bit 0 pode ser enviado na segunda metade do bit slot
  - **Restrição de Tempo:** O bit 0 é enviado depois de 466,2μs.
  - **Tipo de Falha:** Falha de Restrição de Tempo.
- **Falha 23:** O bit 1 pode ser enviado na segunda metade do bit slot
  - **Restrição de Tempo:** O bit 1 é enviado depois de 466,2μs.
  - **Tipo de Falha:** Falha de Restrição de Tempo.
- **Falha 24:** Dois bits 0 são enviados muito rapidamente e o receiver entende como apenas um bit 0.
  - **Restrição de Tempo:** O stream de bits 00 é enviado no intervalo de tempo entre 421,8μs e 446,2μs.
  - **Tipo de Falha:** Falha de Restrição de Tempo.
- **Falha 25:** Dois bits 1 são enviados muito rapidamente e o receiver entende como apenas um bit 1.
  - **Restrição de Tempo:** O stream de bits 11 é enviado no intervalo de tempo entre 421,8μs e 446,2μs.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 26:** O sender envia um bit 1 e um bit 0 muito rapidamente e o receiver só entende um bit 0.

  - **Restrição de Tempo:** O stream de bits 10 é enviado no intervalo de tempo entre 421,8μs e 446,2μs.

  - **Tipo de Falha:** Falha de Restrição de Tempo.

- **Falha 27:** O sender envia um bit 0 e um bit 1 muito rapidamente e o receiver só entende um bit 1.

  - **Restrição de Tempo:** O stream de bits 01 é enviado no intervalo de tempo entre 421,8μs e 446,2μs.

  - **Tipo de Falha:** Falha de Restrição de Tempo.