Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Abordagem Dirigida a Modelo para a Geração de Casos de Teste Baseada na Detecção de Cenários Implícitos

Thiago Peixoto dos Reis

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof.ª Dr.ª Genaína Nunes Rodrigues

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Prof.ª Dr.ª Alba C. M. A. Melo

Banca examinadora composta por:

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientador) — CIC/UnB
Prof. Dr. Nabor das Chagas Mendonça — UNIFOR
Prof. Dr. George Luiz Medeiros Teodoro — CIC/UnB

CIP — Catalogação Internacional na Publicação

Reis, Thiago Peixoto dos.

Uma Abordagem Dirigida a Modelo para a Geração de Casos de Teste Baseada na Detecção de Cenários Implícitos / Thiago Peixoto dos Reis. Brasília : UnB, 2015.
88 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2015.

1. Teste de Software, 2. Teste Baseado em Modelos,
3. Dependabilidade, 4. Confiabilidade, 5. Cenários Implícitos.

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Abordagem Dirigida a Modelo para a Geração de Casos de Teste Baseada na Detecção de Cenários Implícitos

Thiago Peixoto dos Reis

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Nabor das Chagas Mendonça
UNIFOR

Prof. Dr. George Luiz Medeiros Teodoro
CIC/UnB

Prof.ª Dr.ª Alba C. M. A. Melo
Coordenadora do Mestrado em Informática

Brasília, 07 de Agosto de 2015

Dedicatória

Dedico este trabalho a Deus por estar sempre presente em minha vida e nortear minhas escolhas. Aos meus pais, meus irmãos, minha afilhada e minha namorada pelo incentivo, amor e carinho. E a minha família que sempre me apoiou em meio às dificuldades.

Agradecimentos

Agradeço aos meus pais e a minha família pela compreensão, por minha ausência, e pelo apoio durante esse período.

À minha namorada Mariane pelo carinho, companherismo e incentivo.

À Profª Genaína pelo apoio, paciência e confiança depositada em mim durante o desenvolvimento deste trabalho.

E aos amigos Daniel Souza, Alexandre Vaz e Ulisses Afonseca que me ajudaram e também incentivaram nessa jornada.

A todos que de alguma forma contribuiram com este trabalho, Muito Obrigado!

"A mente humana é um grande teatro. Seu lugar não é na platéia, mas no palco, brilhando na sua inteligência, alegrando-se com suas vitórias, aprendendo com as suas derrotas e treinando para ser a cada dia, autor da sua história."

Augusto Cury

Resumo

Confiabilidade é um pré-requisito, implícito e desejável na maioria dos sistemas. Para que um sistema seja confiável, tal exigência deve ser tida em conta em todas as fases de seu desenvolvimento. Cenários implícitos são uma anomalia gerada quando componentes de sistemas concorrentes se comunicam de uma forma inesperada, não descrita na especificação de comportamento do software. Teste em sistemas concorrentes é um exercício muito caro e difícil de revelar falhas especialmente quando o sistema já está implementado e em funcionamento. Teste baseado em modelo (TBM) é uma alternativa para automatizar a geração de casos de teste em sistemas concorrentes, a fim de reduzir o seu custo e à complexidade inerente à atividade de teste. Este trabalho aproveita a abordagem de teste dirigido a modelo para explorar detecção de cenários implícitos, de modo a melhorar a geração de casos de teste em sistemas concorrentes.

Palavras-chave: Teste de Software, Teste Baseado em Modelos, Dependabilidade, Confiabilidade, Cenários Implícitos.

Abstract

Reliability is a prerequisite, implicit and desirable in most systems. For a system to be reliable, such requirement should be taken into account during all development stages of a system. Implied scenarios are an anomaly generated when concur- rent system components communicate in an unexpected way, w.r.t. to the software behavior specification. Testing concurrent systems is a very expensive exercise and hard to reveal faults specially when the system is already deployed and running. Model-Based Testing is an alternative to automate test case generation in concurrent systems, in order to reduce their cost and the inherent complexity of the testing activity. This work leverages model- driven testing techniques to explore implied scenarios detection so as to improve the generation of test cases in concurrent systems.

Keywords: Software testing, Model Based Testing, Dependability, Reliability, Implied Scenarios.

Sumário

1 Introdução **1**

1.1 O Problema . . . 3

1.2 Solução Proposta . . . 3

1.3 Contribuições . . . 4

1.4 Trabalhos Relacionados . . . 4

1.5 Estrutura do Documento . . . 7

2 Fundamentação Teórica **8**

2.1 Dependabilidade . . . 8

2.1.1 Análise de Dependabilidade . . . 9

2.2 Cenários . . . 9

2.2.1 Cenários Implícitos . . . 10

2.3 Caracterização da Falha . . . 13

2.4 Teste de Software . . . 14

2.4.1 Teste Baseado em Modelos . . . 14

3 Metodologia **19**

3.1 Metodologia . . . 19

3.2 Aplicação da Metodologia . . . 26

3.2.1 Modelagem do Sistema em Cenários . . . 26

3.2.2 Geração do Modelo Comportamental . . . 27

3.2.3 Identificação dos Cenários Implícitos . . . 28

3.2.4 Identificação dos Caminhos que são Responsáveis pelo Cenário Implícito Negativo e Definição do Propósito de Teste . . . 29

3.2.5 Encontrar a Família de Cenários Implícitos e Prover a Geração dos Casos de Teste . . . 30

4 Estudo de Caso **34**

4.1 Contexto da Pesquisa . . . 34

4.2 Sistema *Smart Cams* . . . 35

4.3 Hipóteses 36
4.4 Configuração do Experimento 37
4.5 Análise e Apresentação dos Dados 42
4.5.1 Obtenção do Modelo Comportamental 42
4.5.2 Identificação dos Cenários Implícitos e Geração do Propósito de Teste 42
4.6 Geração dos Casos de Teste 52
4.6.1 Arquitetura 1 52
4.6.2 Arquitetura 2 53
4.7 Resultados e Conclusões 53
4.8 Ameaça à Validade 55

5 Conclusão e Trabalhos Futuros **56**
5.1 Conclusão 56
5.2 Trabalhos Futuros 56

A Diagramas bMSCs **63**
A.1 Diagramas bMSCs da Arquitetura 1, para o sistema *Smart Cams* 63
A.2 Diagramas bMSCs da Arquitetura 2, para o sistema *Smart Cams* 65

B Modelo Comportamental, Descrito em FSP **69**
B.1 Modelo comportamental da Arquitetura 1, para o sistema *Smart Cams*, expresso em FSP 69
B.2 Modelo comportamental da Arquitetura 1, para o sistema *Smart Cams*, expresso em FSP 73

Lista de Figuras

2.1 Ilustração do sistema de boiler (caldeira), usado para apresentar a metodologia [4]. 11
2.2 Modelagem em cenários do sistema de caldeira (Boiler) [4]. 11
2.3 Diagrama hMSC do sistema de gerência de caldeira [4]. 12
2.4 Cenário implícito negativo identificado para o sistema de gerência de caldeira, *Boiler*. 12
2.5 LTSs dos processos p1 e p2 [13]. 17

3.1 Visão Geral da Metodologia 20
3.2 Troca de mensagens onde uma potencial condição de corrida pode acontecer. No caso, $msg2_3$ acontecer antes de $msg2_1$ ou $msg3_1$ 22
3.3 Modelo arquitetural do sistema extraído do LTSA-MSC [18] 28
3.4 Outros cenários implícitos negativos identificados para o sistema de gerência de caldeira, *Boiler*. 29
3.5 Cenário Implícito Mínimo. 31
3.6 Menor caminho no grafo que contém o cenário implícito levando ao estado *STOP*. 32
3.7 Interface do *plug-in Test Generation -TG* 33

4.1 Interface do simulador do sistema *Smart Cams*. 38
4.2 Arquiteturas a serem analisadas 39
4.3 Cenários utilizados na modelagem [44] 40
4.4 Representação em MSC do cenário 1 da arquitetura 1 41
4.5 Representação em MSC do cenário 1 da arquitetura 2. 41
4.6 Primeiro cenário implícito negativo encontrado para a arquitetura 1 (IS1) . 43
4.7 Segundo cenário implícito negativo encontrado para a arquitetura 1 (IS2) . 44
4.8 Terceiro cenário implícito negativo encontrado para a arquitetura 1 (IS3) . 46
4.9 Quarto cenário implícito negativo encontrado para a arquitetura 1 (IS4) . . 48
4.10 Cenário implícito positivo encontrado para a arquitetura 2 (ISP1) 50
4.11 Cenário implícito negativo encontrado para a arquitetura 2 (IS5) 50

A.1 Diagrama bMSC referente ao cenário 1 da Arquitetura 1 63
A.2 Diagrama bMSC referente ao cenário 2 da Arquitetura 1 64
A.3 Diagrama bMSC referente ao cenário 3 da Arquitetura 1 64
A.4 Diagrama bMSC referente ao cenário 4 da Arquitetura 1 65
A.5 Diagrama bMSC referente ao cenário 5 da Arquitetura 1 65
A.6 Diagrama bMSC referente ao cenário 1 da Arquitetura 2 66
A.7 Diagrama bMSC referente ao cenário 2 da Arquitetura 2 66
A.8 Diagrama bMSC referente ao cenário 3 da Arquitetura 2 67
A.9 Diagrama bMSC referente ao cenário 4 da Arquitetura 2 67
A.10 Diagrama bMSC referente ao cenário 5 da Arquitetura 2 68

Lista de Tabelas

3.1 Complexidade dos algoritmos utilizados 26
3.2 laços presentes no sistema *Boiler* 30
3.3 Caminhos que possuem o cenário implícito extraídos do sistema *Boiler* a partir do Algoritmo 1, utilizando como estado final *E*. Os *traces* em negrito são os responsáveis por gerar o cenário implícito. 31
3.4 Caminhos que possuem o cenário implícito extraídos do sistema *Boiler* a partir do Algoritmo 1, e utilizando o *STOP* convergindo ao estado final *E*. Os *traces* em negrito são os responsáveis por gerar o cenário implícito. . . . 32

4.1 Configuração da vizinhança de cada câmera do experimento. 39
4.2 Cenários Implícitos identificados para as famílias da Arquitetura 1 53
4.3 Cenário Implícito identificado para a Arquitetura 2. 53

Capítulo 1

Introdução

A complexidade no desenvolvimento do software vem crescendo com o passar dos anos e a complexidade no desenvolvimento dos casos de teste tende a crescer mais rapidamente devido à busca contínua para uma melhor qualidade de software [1]. Logo, existe a necessidade de se desenvolver sistemas de softwares cada vez mais confiáveis sem extrapolar o orçamento e o cronograma [2]. Com o crescimento da complexidade do software cresce também a dificuldade de se desenvolver casos de teste eficazes, principalmente casos de teste que detectem comportamentos inesperados. Esses comportamentos podem ser gerados a partir da comunicação entre componentes em sistemas concorrentes.

A dificuldade no desenvolvimento dos testes se dá devido a vários fatores, dentre eles [19]:

- A inexperiência do testador em lidar com comportamentos inesperados. Isto implica que os casos de teste sejam desenvolvidos para testar somente os comportamentos esperados listados na especificação de requisitos e que não levem em consideração que exista a possibilidade do sistema fazer algo a mais do que foi previamente estabelecido.

- A criação de testes de forma confiável demanda muito tempo, e os testadores vivem sobre a intensa pressão do tempo devido ao cronograma a ser cumprido.

- Mesmo que o teste tenha sido desenvolvido de forma confiável, ele é amostral e não cobre todas as possibilidades de execução do sistema em tempo de execução, dessa forma alguns tipos de erros podem passar desapercebidos, inclusive os críticos.

- Alguns tipos de testes objetivam desvendar comportamentos inesperados do sistema, contudo, os testadores não estão cientes de todos os comportamentos existentes, então iterações potencialmente perigosas podem permanecer não testadas.

Esses fatores, dentre vários outros, resultam em softwares menos confiáveis que podem apresentar falhas, pois tanto em seu desenvolvimento quanto em sua fase de testes não foram avaliados todos os possíveis comportamentos que os sistemas poderiam exibir. Dentre esses comportamentos inesperados, que se dão a partir da comunicação entre os diversos componentes de um software, estão os chamados de cenários implícitos [4].

A modelagem em cenários é uma forma de modelagem do sistema. Ela tem sido considerada uma forma eficaz de se modelar e analisar o seu comportamento nos estágios iniciais do ciclo de desenvolvimento [12]. Os cenários podem ser especificados utilizando Diagramas de Sequência da UML (Unified Modeling Language) ou diagramas MSCs (Message Sequence Charts). Uma de suas vantagens é possibilitar uma maior participação dos *stackholders*.

Analisando a modelagem de um sistema em cenários podem ser verificadas algumas propriedades do sistema, como a presença de cenários implícitos [4]. Cenários implícitos são comportamentos que não foram especificados no documento de especificação de requisitos do software, contudo devido a composição dos componetes, em sistemas concorrentes, os mesmos podem emergir. Existem dois tipos de cenários implícitos: os positivos e os negativos. Cenários implícitos positivos são cenários que podem ser desejáveis, cenários em que o efeito produzido é aceito e integrado ao modelo. Cenários implícitos negativos geram consequências negativas, podendo levar o sistema a um estado de falha [4].

Os cenários implícitos podem causar um grande desconforto aos desenvolvedores e testadores de software, visto que, por ser um tipo de falha inconsistente [5], os seus efeitos podem ser de difícil detecção. Podendo resultar em softwares menos confiáveis, mesmo após terem sido submetidos a exaustivos processos de testes.

A confiabilidade desejada ao sistema é um atributo de dependabilidade, que é a habilidade de um sistema computacional executar serviços em que se possa justificadamente confiar [6]. Em [5] é apresentado uma metodologia que nos permite analisar de forma quantitativa e qualitativa o impacto da presença dos cenários implícitos na confiabilidade de um sistema, de natureza concorrente, a partir da sua modelagem em cenários. Essa metodologia se mostra útil ao fornecer informações importantes para se analisar propriedades emergentes na composição dos componentes do sistema. Essas informações podem ser utilizadas para orientar importantes ações de refinamento arquitetural, permitindo assim a construção de softwares com maior confiabilidade.

A partir da metodologia proposta por [5] é obtida uma análise que apresenta o impacto da presença dos cenários implícitos em um software. O impacto foi medido pela diferença entre a confiabilidade estimada para o sistema, quando não é avaliada a presença dos cenários implícitos, e a confiabilidade real do sistema, avaliando-se a presença dos cenários implícitos para se compor o resultado.

1.1 O Problema

Uma arquitetura que permita a presença de cenários implícitos pode acarretar falhas graves ao sistema. A confiabilidade de uma modelagem baseada em cenários pressupõe a identificação dos eventuais cenários implícitos presentes e, consequente, a correção do projeto para eliminar ou reduzir os efeitos negativos ocasionados por eles [5].

Existem, na literatura, várias abordagens para a detecção dos cenários implícitos, como por exemplo a abordagem usada pela ferramenta LTSA [17]. Contudo essas abordagens possuem limitações, com relação à detecção dos cenários implícitos. A detecção dos cenários implícitos, nessa abordagem, é um processo iterativo não determinístico que na presença de iterações cíclicas, entre os componentes de um sistema, pode ocasionar a detecção de infinitos cenários implícitos, tornando mais dificil sua representação e correção.

O Teste Baseado em Modelo (TBM) é uma alternativa para a verificação de uma arquitetura. O TBM consiste em uma técnica, caixa preta [1], para geração automática de um conjunto de casos de testes, com entradas e saídas esperadas, utilizando modelos extraídos a partir dos requisitos do software [30]. Para a utilização do TBM é necessário um formato apropriado para a automação das atividades de teste, como a representação de modelos utilizando métodos formais, máquinas de estado finito e a UML.

O TBM, no entanto, não realiza a análise do modelo necessária para a identificação dos cenários implícitos. Além disso, as técnicas atuais de TBM não estão adaptadas ao processo de detecção dos cenários implícitos, os quais podem causar falhas inconsistentes relativas à comunicação entre os componentes do sistema em ambiente de software concorrente.

Frente a esse problema, torna-se premente a necessidade de se complementar o ciclo de Teste Baseado em Modelo (TBM) para que o mesmo leve em consideração a análise dos cenários implícitos, uma vez que foi evidenciado o seu impacto na confiabilidade de um sistema [5]. Existe também a necessidade de se prover uma heurística para a representação a e identificação dos cenários implícitos de uma forma mais eficiente, que contemple as iterações cíclicas que podem existir entre os componentes de um sistema.

1.2 Solução Proposta

Devido aos cenários implícitos serem um tipo de falha inconsistente, de difícil detecção manual, são necessários artifícios que ajudem em sua identificação. Logo, a solução proposta é apresentar uma metodologia que possibilite a geração de casos de teste a partir da detecção dos cenários implícitos.

A metodologia mostra-se útil ao fornecer informações importantes para se analisar propriedades emergentes na composição dos componentes de um sistema de natureza concorrente. As informações obtidas a partir da metodologia podem ser utilizadas para orientar tomada de ações com relação à correção da arquitetura do sistema, por meio da identificação da extensão das falhas ocasionadas pelos cenários implícitos.

Outro benefício obtido pelo uso da metodologia é a possibilidade da reexecução dos casos de teste após a correção do modelo, a fim de se verificar se realmente foi corrigido o problema, ou se na correção não foram inseridos novos cenários implícitos â arquitetura.

1.3 Contribuições

Como principal contribuição temos:

- O desenvolvimento de uma metodologia para a geração de casos de teste, com base na busca dos cenários implícitos.

Como contribuição secundária temos:

- A criação de um algorítmo para a geração de casos de teste que identificam a presença dos cenários implícitos.
- A adaptação do algorítmo DFS [26] para que atenda às necessidades impostas pelo algoritmo de geração dos casos de teste.
- A criação de um plug-in para a ferramenta LTSA-MSC [18] que automatiza o processo de geração de caso de teste.
- A validação da metodologia com sua aplicação em um sistema real chamado *Smart Cams* [50, 51].

1.4 Trabalhos Relacionados

Em [24] de *Bertolino et al.* propuseram uma metodologia para a automação do desenvolvimento de casos de teste a partir da especificação UML, com o intuito de gerar os casos de teste antes do desenvolvimento do software. Essa abordagem possui como diferencial um aumento da acurácia e a diminuição do esforço no desenvolvimento dos casos de teste. Para isso eles recebem como entrada diagramas de estado, e diagramas de sequência, modelados em UML, daí eles verificam a conformidade do diagrama sequência com o diagrama de estado. Contudo, utilizam apenas o diagrama de estado como o modelo de referência, a fim de gerar os futuros casos de teste.

Para realizar a geração de casos de teste eles pressupõem que a geração dos conjuntos de diagramas de estado e diagramas de sequência podem ser falhos, pois podem ser inicialmente incompletos e ter somente uma especificação parcial ou inconsistente. Logo, em sua abordagem, é preciso verificar essas inconsistências antes de gerar os casos de teste.

Em seu trabalho *Bertolino et al.* utilizam o método UIT (*Use Interaction Test*) consiste em um método que constrói e define, de forma sistemática, um conjunto de casos de teste para a fase de testes de integração usando os diagramas UML como único modelo de referência, sem exigir a introdução de quaisquer formalismos adicionais.

Embora essa proposta esteja relacionada a TBM, ela não chega a gerar casos de teste que identifiquem os cenários implícitos. No entanto, ela verifica a presença dos cenários implícitos, utilizando o LTSA-MSC, e os corrige, por meio do refinamento arquitetural, antes da geração dos casos de teste.

Uma das possibilidades para o refinamento arquitetural, para tratar cenários implícitos, é o uso de contraints, o que pode não ser tão benéfico, pois em fases posteriores à fase de testes de integração ou de componentes fica mais difícil de se rastrear os cenários implícitos e verificar a conformidade entre o modelo e a especificação, por não haver, na suite de testes, os testes capazes de identificar os cenários implícitos. Em nossa abordagem propomos a geração de casos de teste, a partir da identificação dos cenários implícitos, em qualquer momento do desenvolvimento do software, desde a elicitação dos requisitos até a fase de manutenção do software.

Em [25] *Al-Azzani et al.* afirmam que uma das causas de vulnerabilidades em sistemas é a presença de cenários implícitos. Dessa forma, propuseram então o uso da detecção dos cenários implícitos para a criação de teste de segurança. A abordagem proposta por *Al-Azzani et al.* consiste em três etapas.

1. Detecção dos cenários implícitos: na primeira fase de sua proposta, Al-Azzani et al. identificam, utilizando a ferramenta LTSA [18], os cenários implícitos presentes em uma determinada especificação de software.

2. Revisão dos cenários implícitos detectados: na segunda fase, eles propõem encontrar as vulnerabilidades de segurança, violações, geradas partir dos cenários implícitos encontrados e catalogá-los. Um cenário implícito pode gerar vários tipos de violações.

3. As vulnerabilidades encontradas são utilizadas por testadores para criar novos casos de teste.

A abordagem proposta por Al-Azzani et al. cria um roteiro para a geração de casos de teste voltados para teste de segurança de forma semi-automatizada, uma vez que o

LTSA [18] retorna de forma automatizada somente os cenários implícitos. Embora em sua metodologia seja citada a criação de casos de teste, *Al-Azzani et al.* não chegam a gerar os casos de teste. Em nossa abordagem propomos uma automatização da geração de casos de teste para a verificação de cenários implícitos.

Felipe Cantal *et al.* [11] demonstram como o conceito de cenários implícitos, até agora restrita às fases iniciais do ciclo de vida do software, pode ser aplicado para apoiar a compreensão e testes em sistemas já existentes. Eles apresentam um processo de engenharia reversa para apoiar a extração e detecção de cenários implícitos a partir de informações capturadas durante a execução de aplicações concorrentes.

A abordagem proposta em [11] para a detecção dos cenários implícitos se baseia no uso de engenharia reversa para obter uma modelagem parcial do software de forma iterativa, das execuções de suas funcionalidades. Assim, novos cenários são gerados a partir dos dados obtidos com a detecção dos rastros de execução, onde a detecção dos cenários implícitos é então feita na ferramenta LTSA-MSC [18].

Embora essa abordagem promova a detecção dos cenários implícitos a partir do uso de engenharia reversa ela não realiza a geração de casos de teste. Contudo, deixa o ambiente mais favorável a sua geração, visto que promove a identificação dos cenários implícitos em sistemas já implementados, que podem não ter sido documentados corretamente. Em nosso trabalho propomos a geração de forma automática dos casos de teste a partir da detecção dos cenários implícitos. Logo o processo proposto por Cantal *et al.* pode ser integrado à nossa proposta a fim de gerar casos de teste em sistemas que não possuam um documento de especificação de requisitos.

Em [13], Nogueira *et al.* promovem, a partir de uma abordagem TBM, a criação de uma ferramenta que gera automaticamente casos de teste orientados a propósitos, a ATG [13]. Esta recebe como entrada o comportamento da aplicação, modelado em CSP (*Communicating Sequential Processes*) [46], e o propósito de teste, que é o cenário que se deseja testar. Este cenário é definido manualmente por um testador a partir da avaliação do documento de especificação de requisitos.

A partir do propósito de teste, a ferramenta ATG faz a análise de consistência de tal propósito e verifica se o mesmo de teste está consistente com a implementação. Para realizar essa análise de consistência, a ferramenta ATG encapsula outra ferramenta em seu código fonte, a FDR(*Failures and Divergences Refinament*), que verifica se, de acordo com os modelos semânticos do CSP, um processo (implementação) é refinamento do outro (especificação). Caso não seja, a FDR gera contra-exemplos que expõe a violação de propriedade do modelo semântico. Utilizando a avaliação promovida pelo FDR e os contra-exemplos gerados, a ferramenta ATG gera os casos de teste em CSP. A ATG verifica também propriedades clássicas de processos, como: ausência de *deadlock*, ausência

de *livelock* e o comportamento determinístico dos processos. Ela também gera o sistema de transição de estados (do inglês Labelled Transition System - LTS) de um processo CSP.

A ferramenta ATG promove, em uma abordagem de TBM, a geração automática de casos de teste, mas não trata especificamente sobre cenários implícitos, pois a ferramenta não promove uma análise da comunicação entre os componentes nos diferentes cenários. Logo o caminho que configura a presença do cenário implícito seria tratado como um caminho qualquer na modelagem, ficando a cargo do testador descobrir, a partir da avaliação do documento de requisito, se existe ou não a presença do cenário implícito. Isto torna-se uma tarefa muito cara, devido ao tempo despendido, se desenvolvida de forma manual.

1.5 Estrutura do Documento

Os demais capítulos dessa dissertação estão organizados da seguinte forma:

- O Capítulo 2 apresenta a fundamentação teórica abordando os principais conceitos utilizados nesse trabalho, incluindo os conceitos de Análise de Dependabilidade, Modelagem em Cenários e Cenários Implícitos, Caracterização da Falha e Teste de Software;
- O Capítulo 3 apresenta a metodologia proposta e ilustra o seu uso em um sistema de caldeira apresentado em trabalhos seminais [4, 19, 27];
- O Capítulo 4 apresenta um estudo de caso utilizando um sistema de câmeras inteligentes; e
- o Capítulo 5 apresenta as conclusões obtidas com esse estudo e a proposta para os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os principais conceitos utilizados nesse trabalho. Primeiro, a Seção 2.1 apresenta uma introdução sobre dependabilidade. Em seguida, na Seção 2.2, abordamos sobre a modelagem de componentes de um sistema com o uso de cenários, em seguida é introduzido o conceito de cenários implícitos, na Seção 2.2.1. Posteriormente, na Seção 2.3, abordamos o conceito de caracterização da falha, proposto por [6], a fim de descrever o tipo de falha que será tratada pela metodologia. A Seção 2,4 apresenta o conceito inicial de teste de software e a Seção 2.4.1, descreve teste baseado em modelo.

2.1 Dependabilidade

O conceito de dependabilidade está relacionado à habilidade do sistema em evitar falhas que são mais frequentes e severas que o aceitável. Assim como já mencionado no capítulo anterior, a dependabilidade é a habilidade de um sistema computacional executar serviços em que se possa justificadamente confiar. Tal conceito engloba os seguintes atributos [6]:

- Disponibilidade (*Availability*) - Prontidão do sistema em se executar o serviço de forma correta.
- Confiabilidade (*Reliability*) - Continuidade da execução do serviço de um sistema de software de forma correta.
- Segurança (*Safety*) - A execução do sistema não tem consequências catastróficas para o usuário ou para o ambiente.
- Integridade (*Integrity*) - A execução do sistema não gera alterações impróprias no mesmo.

- Manutenabilidade (*Maintainability*) - Facilidade na modificação ou no reparo do sistema.

2.1.1 Análise de Dependabilidade

A análise de dependabilidade de um sistema é o estudo dessas propriedades. A importância da análise de confiabilidade nas fases iniciais de desenvolvimento do software tornou-se cada vez mais evidente [7]. Uma vez que a qualidade do software é analisada nas fases iniciais do seu ciclo de vida, é possível antecipar a identificação de problemas relacionados ao desenvolvimento de software e evitar despesas imprevistas de custo, tempo e esforço [10].

Ao longo de mais de cinquenta anos foram desenvolvidos vários meios para se atingir os atributos de dependabilidade, os quais podem ser agrupado em quatro grandes grupos [6]:

- Previsão de Falhas - Evitar a ocorrência ou a introdução de defeitos;
- Tolerância a Falhas - Evitar falhas de serviço na ocorrência de uma falha;
- Remoção das Falhas - Reduzir o número e a gravidade das falhas; e
- Estimativa de Falhas - Meios para se estimar o número atual, a incidência futura e as prováveis consequências das falhas.

Neste trabalho é apresentada um nova forma para a elicitação de casos de teste, em sistemas concorrentes, que atua nas fases iniciais do desenvolvimento do software, mas não restrita a essas fases, a fim de se proporcionar uma maior qualidade aos casos de teste desenvolvidos. Com esta metodologia tem-se como objetivo evitar que ocorram falhas no sistema, geradas a partir da ocorrência de cenários implícitos, proporcionando uma maior confiabilidade ao sistema por meio de uma melhor previsão e estimativa das falhas.

2.2 Cenários

Um cenário descreve como um ou mais componentes de um sistema se relacionam para oferecer um conjunto de funcionalidades [11]. Cada cenário representa uma visão parcial do comportamento global do sistema, o qual, para ser entendido por completo, depende da combinação dessas diferentes especificações [12].

Especificações baseadas em cenários são geralmente expressas utilizando notações como *Message Sequence Charts* (MSCs) [13] e Diagramas de Sequência [14], bastante empregadas na descrição de requisitos de software. Eles representam, de uma forma intuitiva e flexível, os diferentes fluxos de mensagens entre componentes que caracterizam

a execução de um sistema concorrente [15]. Por outro lado, esse tipo de especificação sofre de uma séria limitação, na qual nem sempre um modelo de comportamento de um sistema é a soma exata do conjunto de comportamentos expressos nas especificações de seus cenários individuais.

Essas combinações inesperadas no modo como os componentes interagem podem forçar o aparecimento de comportamentos que não foram especificados nos cenários originais. Tais comportamentos são chamados de cenários implícitos [4], que surgem principalmente porque os componentes têm, em geral, uma visão local do que está acontecendo no sistema, e não uma visão do comportamento global esperado [16].

2.2.1 Cenários Implícitos

Os cenários implícitos capturam comportamentos inusitados que não foram especificados explicitamente, mas que podem ocorrer devido à natureza parcial e concorrente das especificações baseadas em cenários [16]. Por essa razão os cenários implícitos precisam ser detectados, validados e categorizados como cenários implícitos positivos ou negativos [17].

Cenários implícitos positivos são cenários que não foram especificados originalmente, mas o resultado proporcionado pelo mesmo é benéfico ao sistema, podendo ser integrado ao modelo. Cenários implícitos negativos são cenários que também não foram especificados e evidenciam um desvio do comportamento esperado para o sistema, ocasionando erros.

Para esse trabalho foi necessário definir um novo conceito, o de Família de Cenários Implícitos. Família de cenários implícitos é um conjunto de cenários implícitos negativos que contém o mesmo padrão de mensagens que são responsáveis pela sua formação. Essa família, por possuir o mesmo conjunto de caminhos responsáveis por sua formação, produz um mesmo tipo de falha que pode ser tratado de forma similar.

A ferramenta LTSA-MSC [17] oferece um suporte automatizado para a detecção, incremental, dos cenários implícitos. Ela foi desenvolvida como uma extensão da ferramenta de verificação de modelos concorrentes LTSA, Labelled Transition Systems Analyzer [15].

O LTSA-MSC possui uma linguagem baseada em diagramas MSCs que utiliza-se da composição paralela de cenários básicos. Eles são representados por MSCs básicos ou bMSCs (*basic Message Sequence Chart*), e são agrupados para criar uma descrição de cenários em alto nível, representados por MSCs de alto nível ou hMSCs (*high level Message Sequence Chart*) [18]. Uma especificação formada por um cenário de alto nível, produzida a partir de um conjunto de cenários básicos, é o necessário para que a LTSA modele todas as possíveis iterações entre os cenários e detecte a presença dos cenários implícitos na especificação.

Na Figura 2.1 temos a representação de um sistema de caldeira (*Boiler*) [19] que possui um cenário implícito. Este sistema representa uma caldeira com sensores que medem a

pressão interna da caldeira, e o processo de análise dos dados coletados pelos sensores resultam no aumento ou na diminuição da temperatura dentro da caldeira.

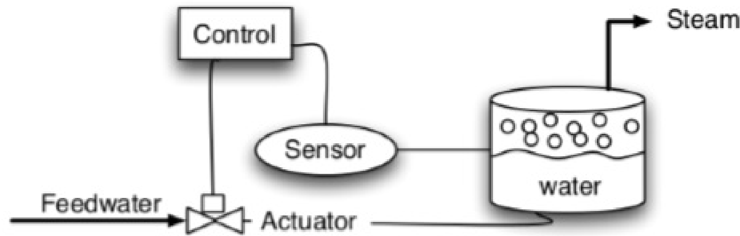


Figura 2.1: Ilustração do sistema de boiler (caldeira), usado para apresentar a metodologia [4].

Na Figura 2.3 temos o sistema de caldeira expresso em MSCs. No bMSC *Initialize*, a mensagem do componente *Control* para o componente Sensor representa a ativação do sensor. O bMSC *Register* representa o registro dos dados, referentes à pressão que foi coletada pelo sensor, no componente *Database*. O bMSC *Analysis* representa uma sequência de mensagens entre os componentes *Control*, *Database* e *Actuator*, essas mensagens correspondem ao processo de análise do sistema, que pode resultar no aumento ou na diminuição da temperatura da caldeira, dependendo dos valores de pressão coletados pelo sensor e armazenados no banco de dados. Por fim, o bMSC *Terminate* representa a ação de desativação do sensor pelo componente de controle.

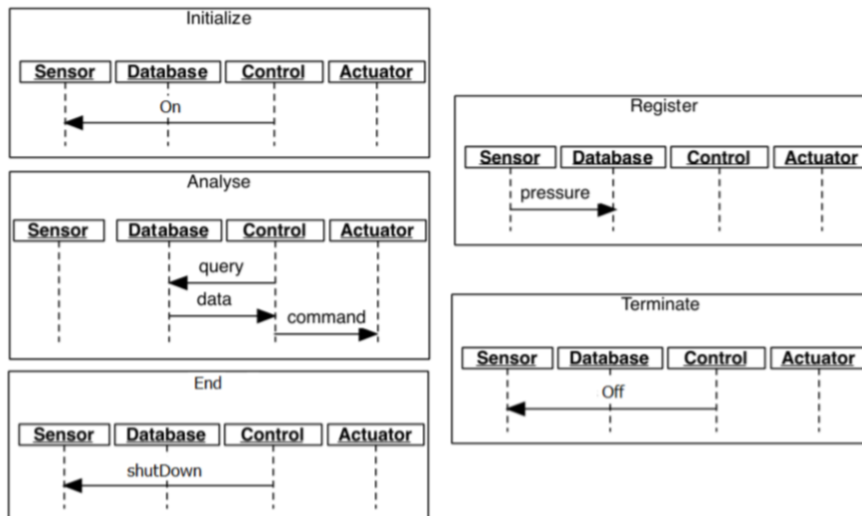


Figura 2.2: Modelagem em cenários do sistema de caldeira (Boiler) [4].

O cenário de alto nível (hMSC) mostrado na Figura 2.3 define as possíveis relações de continuidade entre os quatro cenários básicos do sistema. Note que, do ponto de vista individual de cada cenário, todos os componentes apresentam comportamentos válidos.

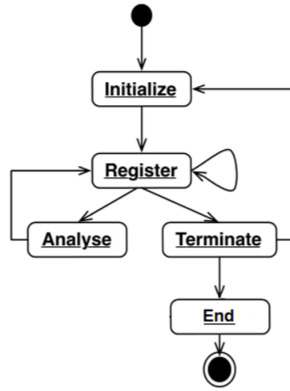


Figura 2.3: Diagrama hMSC do sistema de gerência de caldeira [4].

No exemplo em questão, o cenário implícito retornado pela ferramenta LTSA-MSC, apresentado na Figura 2.4, se caracteriza pela seguinte sequências de mensagens iniciadas pelo controlador: $On \rightarrow Pressure \rightarrow Off$, representando um caminho de mensagens trocadas entre sensor e banco de dados. Posteriormente, o sensor é ligado novamente por uma nova mensagem *On* do controlador ao sensor seguido imediatamente de uma mensagem *Query* partindo do controlador ao banco de dados e consultando uma temperatura que pode estar defasada, gerando uma tomada de ação, pelo sistema, que pode ser errada naquele momento.

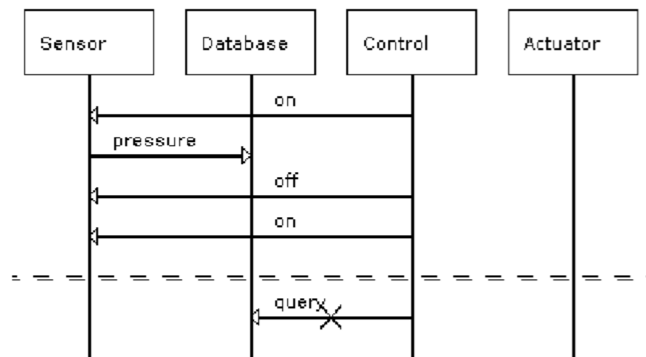


Figura 2.4: Cenário implícito negativo identificado para o sistema de gerência de caldeira, *Boiler*.

A aplicabilidade da detecção dos cenários implícitos é muito grande. Emmanuel Letier [38] apresenta exemplos reais de utilização de sua detecção, como:

- Em dois modelos diferentes de caixas eletrônicos [39, 40];
- Um aplicativo de interface web [41];
- Uma torradeira [42];

- Um sistema de caldeira [19], que é utilizado neste trabalho;
- Um protocolo gestão de mobilidade GSM [42];
- Cenários para um sistema de transporte automatizado [18]; e
- Para a injeção de segurança em um sistema de uma usina nuclear [43];

2.3 Caracterização da Falha

Uma falha ocorre quando o serviço entregue diverge daquele que foi especificado [6]. Para a análise dos cenários implícitos é importante a caracterização do tipo de falha que o sistema pode gerar. Em [6] foi proposto uma taxonomia para a classificação de falhas caracterizando-as em quatro pontos de vista: Domínio (Tempo ou Conteúdo), Detectabilidade (Detectável ou Indetectável), Consistência (Consistente ou Inconsistente) e Impacto (Mínima ou Catastrófica).

A categoria de falhas de Domínio é usada para distinguir as falhas de Conteúdo de falhas de Tempo:

- Falhas de Conteúdo: Quando o conteúdo da informação entregue por algum serviço desvia de sua especificação original.
- Falha de Tempo: Quando o tempo de chegada ou duração da informação entregue por algum serviço se desvia de sua especificação.

A Detectabilidade da falha nos permite saber sua real extensão.

- Falhas Detectáveis: As consequências das falhas podem ser detectadas por meio de técnicas de instrumentação, as quais detectam informações por meio de assertivas.
- Falhas Indetectáveis: As consequências das falhas não são detectadas até que a mesma ocorra.

A Consistência da falha se dá quanto à percepção da mesma pelos usuários.

- Falhas Consistentes: o serviço incorreto é percebido de modo idêntico a todos os usuários;
- Falhas Inconsistentes: o serviço incorreto é percebido de modo diferente pelos usuários.

O Impacto da falha permite a definição de sua severidade aos usuários e/ou ambiente.

- Falha Mínima: As consequências danosas da falha são limitadas ou no máximo similares aos benefícios providos pela operação correta do sistema.
- Falha Catastrófica: As consequências danosas das falhas são imensuravelmente maiores que os benefícios providos pela operação correta do sistema.

2.4 Teste de Software

Com a expansão do desenvolvimento e do uso de sistemas computacionais a qualidade e a confiabilidade do software tornou-se um pré-requisito intrínseco ao desenvolvimento do software. O teste de software tem como objetivo guiar e quantificar essa qualidade e confiabilidade.

Myers [20] estabelece um conjunto de regras que podem servir como objetivos de teste:

1. A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro.
2. Um bom caso de teste é aquele tem uma elevada probabilidade de revelar um erro ainda não descoberto.
3. Um teste bem sucedido é aquele que revela um erro ainda não descoberto.

No processo de teste de software estão disponíveis diversas técnicas, sendo possível optar por uma combinação dessas técnicas, pois elas se complementam. São algumas delas: teste funcional [21], teste estrutural [22], teste baseado em erros [22] e o teste baseado em modelo [29]. Cada técnica de teste de software está relacionada a uma fase do ciclo de desenvolvimento do software. Logo, cada uma tem objetos de teste diferentes, em que os objetos de teste são as fontes de informação para cada técnica. Por exemplo, as especificações funcionais são os objetos de teste da técnica funcional.

Em [23] *Yang et al.* citam os principais desafios em se desenvolver testes para programas concorrentes, entre eles, detectar situações não desejadas como: erros de sincronização, de fluxo de dados e deadlock. A construção de casos de teste a partir da detecção de cenários implícitos contribui para contornar esse desafio em particular.

2.4.1 Teste Baseado em Modelos

O Teste Baseado em Modelos (TBM) é uma técnica de teste formal [1]. O teste formal é baseado em um modelo ou em uma especificação, nele é verificado a conformidade do comportamento desejado. Esse comportamento é expresso em uma linguagem formal com uma sintaxe e semântica definidas com precisão [1].

O TBM consiste em uma técnica para geração automática de um conjunto de casos de testes, com entradas e saídas esperadas, utilizando modelos extraídos a partir dos requisitos do software [30]. Para a utilização do TBM é necessário um formato apropriado para a automação das atividades de teste, como a representação de modelos utilizando métodos formais, LTS, máquinas de estado finito e a UML.

O TBM é uma abordagem caixa-preta [1] que surge como uma abordagem aplicável para controlar a qualidade do software, assim como reduzir os custos associados ao processo de testes, visto que casos de teste podem ser gerados a partir da especificação do software, antes do seu desenvolvimento, utilizando procedimentos automáticos que podem ser menos suscetíveis a erros. O TBM pode gerar melhorias nas atividades de teste por meio da simplificação do seu planejamento, da semi-automação de suas atividades, e do controle, a partir da gerência dos testes e possibilidade da re-execução dos casos de teste de forma automática após modificações. No teste baseado em modelo uma implementação em teste é testada para conformidade com um modelo que descreve o comportamento exigido pela execução [1].

No teste baseado em modelo, os casos de teste são derivados a partir de um modelo que descreve aspectos, geralmente funcionais, do sistema a ser testado. Tais casos são conhecidos como a suite abstrata de testes, e seu nível de abstração está intimamente relacionado ao nível de abstração do modelo, podendo ser executados em um ambiente derivado em uma suite executável de testes, por exemplo, em uma ferramenta específica para essa finalidade.

Sua eficácia e utilização estão relacionadas ao seu potencial de automação. Se o modelo pode ser processado formalmente, os casos de teste podem ser derivados mecanicamente. O modelo geralmente é traduzido em uma máquina de estado finito que representa as possíveis configurações do sistema. Para encontrar os casos de teste, procura-se pelos caminhos de execução. Entretanto, dependendo da complexidade do sistema, a quantidade de caminhos pode ser muito grande, requerendo então heurísticas para encontrar os mais relevantes. Entre as formas de derivação da suite abstrata de teste estão: a geração por prova de teorema, programação com restrições, verificação de modelo, execução simbólica, modelo de fluxo de eventos e cadeias de Markov.

Embora existam diversos tipos de modelos que podem ser utilizados no TBM, nesse trabalho utilizamos as máquinas de estados finitas para modelar o comportamento da aplicação e gerar os casos de teste para a detecção dos cenários implícitos.

2.4.1.1 Representação Usando LTSs

Testar um sistema a partir de uma especificação em LTS consiste em verificar a conformidade da interação entre o agente formal, representado pela implementação sob teste (IUT - *Implementation Under Test*) e o caso de teste, ambos descritos em termos de LTSs [31]. Neste trabalho a IUT será representada pela modelagem, em LTS, do sistema a ser implemnntado. Para que os testes possam ser gerados de uma maneira coerente, é necessário estabelecer a noção de conformidade com a especificação. A conformidade é obtida pela relação de implementação entre o modelo formal da IUT e a especificação formal do comportamento do sistema a ser testado (SUT — *System Under Test*) [31,32], ou seja, a conformidade é a representada pela diferença entre o comportamento especificado e o comportamento obtido na implementação do sistema.

O caso de teste modela o comportamento das trocas de mensagens em um sistema durante uma execução do teste realizado sobre a IUT [13]. Para decidir sobre o sucesso de um caso de teste, sobre uma IUT, é associado a ele um veredito. Existem dois verediddos associados ao caso de teste: Passou (quando a IUT passou na execução do caso de teste) e Falhou (quando a IUT apresenta um comportamento que diverge do especificado). Existe um terceiro veredido o Inconclusivo (quando o comportamento da IUT observado é consistente, com a relação de implementação, no entanto, o objetivo do teste não pode ser alcançado), geralmente utilizado quando o teste é dirigido a objetivo [33].

Formalmente um caso de teste é um LTS que possui um mapeamento associando que associa vereditos a estados, onde v (denota um veredito) e S (denota um estado) [13].

$v : S \rightarrow (pass, fail)$

2.4.1.2 Relação de Implementação

Uma relação de implementação define as regras de conformidade que serão usadas em determinado processo de teste. Na literatura existem várias relações de implementação disponíveis, como a relação *trace Preorder* [34], a *Testing Preorder* [35] a *Conf* [36] e a *Ioco* [37], dentre outras.

Nesse trabalho utilizaremos a relação de implementação ioco (Input-Output Conformance) ela define a conformidade entre as implementações (IUT) e especificações. Essa relação considera não só as comunicações síncronas, mas também as assíncronas.

A relação ioco se baseia em IOLTS (input-output transition systems) é uma classe especial de LTS onde um conjunto de eventos L é particionado em eventos de entrada L_I e de saída L_O. Na ioco as entradas de um IOLTS se comunicam com as saídas de outro(s) IOLTS(s), e vice-versa. Em um contexto particular, no propósito de teste (o caminho do LTS que se deseja testear), que é representado por um caso de teste IOLTS, as ações de

saída do propósito podem ser ações de entrada para outro IOLTS da IUT, e as saídas da IUT podem ser entradas para outro caso de teste [13].

2.4.1.3 Geração de Testes

Como visto, existe na literatura um grande número de relações de implentação e vários algoritmos fazem uso dessas relações para proverem a geração dos caso de teste de forma automatizada. Como exemplo para a geração dos caso teste baseados em modelo, utilizando LTSs para a sua representação, temos o exemplo da Figura 2.5, extraído de [13], cujo os LTSs representados modelam o comportamento dos processos p1 e p2. Cada processo específica o funcionamento de uma máquina de café, L é o conjunto das possíveis ações realizadas pelas máquinas de café.

$L = \{shil, choc, liq\}$

A ação *shil* representa o ato de apertar botão para que se inicie um processo na máquina, e *liq* e *choc* representão respectivamente os eventos de adicionar o leite e adicionar o chocolate. Para o funcionamento da primeira máquina, *p1* apresentadado na Figura 2.5, aperta-se o botão *shil* para que a máquina se prepare para servir algo, em seguida o usuário deverá selecionar qual bebida deseja, leite (*liq*) ou chocolate (*choc*). Já o funcionamento da segunda máquina, *p2* apresentadado na Figura 2.5, após o usuário apertar o botão *shil* ela servirá uma das duas bebidas de forma aleatória, não-determinística, sem que o usuário tenha controle sobre qual será a bebida servida. Isto acontece pois a máquina p2 oferece dois eventos que possuem o mesmo nome, contudo levam a caminhos diferentes.

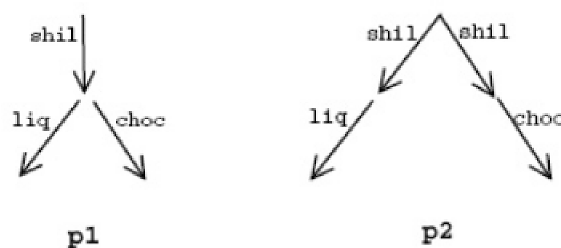


Figura 2.5: LTSs dos processos p1 e p2 [13].

Em [32] é definido um algoritmo para a geração de casos de testes que, a partir de especificações descritas em LTS, gera um conjunto coerente de casos de testes. A relação de implementação utilizada é a *conf*. Dado o processos p1 e p2 da Figura 2.5, temos um conjunto de testes, CT, especificado em uma notação CSP.

$CT = T1; T2$

Observando de maneira isolada os casos de teste T1 e T2, que possuem comportamentos complementares e podem ser executados sequencialmente, temos que o teste T1, possui a especificação:

$T1 : \{shil \rightarrow Skip\}$[1]

O caso de teste T1 verifica se a implementação consegue executar o evento shil a partir do estado inicial. O segundo caso de teste, T2, possui a especificação:

$T2 : \{choc \rightarrow Stop\ ou\ liq \rightarrow Stop\}$[2]

O caso de teste T2 verifica se após executar *shil* se a implementação permite que o usuário escolha uma das opções de bebida, *choc* ou *liq*. Os casos de teste T1 e T2 podem ser aplicados aos dois processos descritos na Figura 2.5. Para o teste T1 aplicado ao processo p1 e p2 temos o resultado foi que o caso de gerado passou com sucesso. O caso de teste T2 aplicado ao processo p1 passa com sucesso, enquanto para o processo p2 o caso de teste gerado falha, pois o processo p2 não permite que o usuário escolha da bebida.

[1]O *Skip* é um processo primitivo do CSP que representa o término de uma execução com sucesso.

[2]O *Stop* é um processo primitivo do CSP que representa um estado problemático de um sistema, também pode ser usado para representar um deadlock.

Capítulo 3

Metodologia

Neste capítulo apresentamos nossa abordagem de geração de casos de teste dirigida a modelos a partir da detecção de cenários implícitos, particularmente negativos. Essa geração de casos de teste é uma abordagem de TBM que tem por objetivo representar a real extensão da falha gerada pela presença dos cenários implícitos.

3.1 Metodologia

A Figura 3.1 apresenta a metodologia desenvolvida. Para as etapas iniciais do processo (etapas de I a IV) fazemos uso da metodologia desenvolvida nos trabalhos de Uchitel *et al.*, já abordadas nas Seções 2.2 e 2.2.1. Após a etapa de identificação dos cenários implícitos negativos, sucedem-se as etapas para a definição das assertivas de teste e a geração dos casos de teste.

As etapas V em diante constituem o núcleo da nossa metodologia. Uma vez realizada a validação do cenário implícito negativo detectado, define-se o propósito de teste. Esse propósito constitui, em essência, na sequência de mensagens causadoras do cenário implícito negativo detectado. A partir do propósito de teste é gerado o cenário de teste, ou seja, é gerado um caminho que parte do estado inicial do LTS e chega a um estado final, passando pelos *traces* do propósito de teste. A partir daí, é gerada automaticamente a família de cenários implícitos, seguida então da geração dos casos de teste. Esses casos de teste são gerados em FSP (Finite State Process) mas simbolicamente representados com a sequência de números que representam os estados no modelo arquitetural em LTS. Para este trabalho foi necessário definir um novo conceito, o de família de um cenário implícito.

Definição 1. *Família de um Cenário Implícito - Dado um caminho (trace), que caracteriza uma troca de mensagens em um cenário implícito negativo, uma família de um cenário implícito é composta por todos os caminhos finitos que contém a troca de mensagens que caracterizam o cenário implícito.*

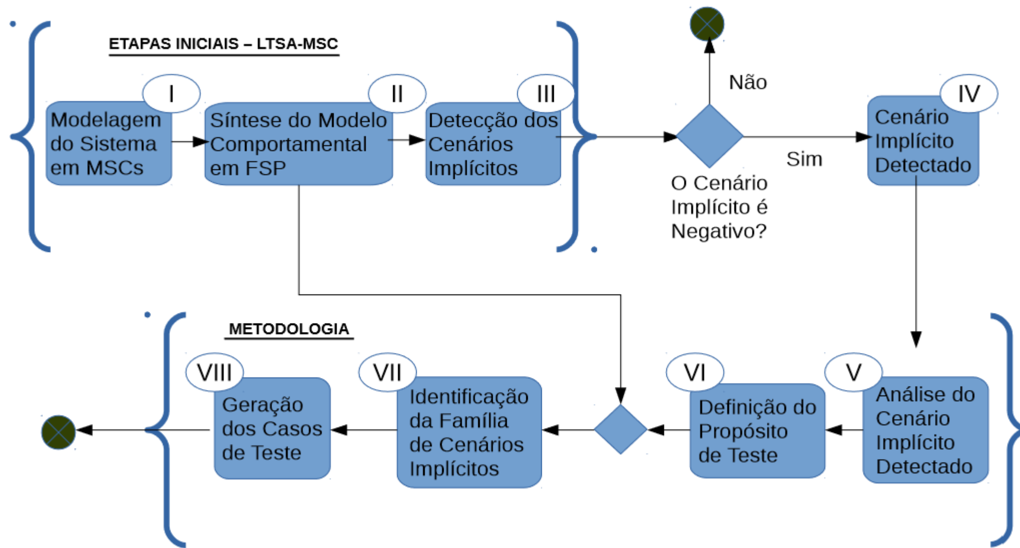


Figura 3.1: Visão Geral da Metodologia

Conforme [52] um caminho, ou um *trace*, em um LTS é um fragmento de caminho inicial maximal, onde um caminho maximal pode ser tanto finito quanto infinito. Caso seja finito, o caminho maximal finaliza com um estado terminal. No caso da família de um cenário implícito, ela caracteriza por conter todos os caminhos inicial e maximal finito que contém as mensagens que caracterizam um cenário implícito. No caso do cenário implícito ilustrado na Figura 2.4, a sequência de mensagens que caracterizam o cenário implícito é: $on \rightarrow query$. Sendo assim, a família do cenário implícito em questão é constituída por todos os caminhos inicial e maximais finitos do LTS que contém essa sequência de mensagens.

Para melhor descrevermos a metodologia desenvolvida, definimos oito passos, conforme segue:

- **Passo 1 -** Modelagem do sistema em cenários utilizando os diagramas MSC a fim de se inserir o modelo na ferramenta LTSA - A partir do documento de especificação de requisitos do diagrama de casos de uso o sistema pode ser modelado em cenários;

- **Passo 2 -** Geração do modelo comportamental do sistema - A ferramenta LTSA recebe como entrada o sistema modelado em cenários e uma de suas saídas é o modelo comportamental do sistema que é expresso na forma de um LTS e também na linguagem FSP;

- **Passos 3 e 4 -** Identificação dos cenários implícitos positivos e negativos, realizada pelo LTSA - Após a identificação ocorrerá a verificação manual para a avaliação dos cenários para então realizar a síntese do modelo arquitetural na ferramenta

LTSA-MSC, adequando o modelo para aceitar os cenários implícitos positivos, pois somente trabalharemos com geração de casos de teste para os cenários implícitos negativos;

- **Passo 5 -** A partir da detecção do cenário implícito, identificar os caminhos (*traces*) do sistema que contém as sequências de mensagens responsáveis pelos cenários implícitos negativos - Com a definição deste caminho podemos definir um propósito de teste;

- **Passo 6 -** Definir uma linguagem de entrada para o propósito de teste - Uma linguagem que será utilizada para a inserção do propósito de teste;

- **Passo 7 -** Identificar os cenários de teste, obtidos a partir do propósito de teste, e então identificar a família de cenários implícitos;

- **Passo 8 -** Gerar os casos de teste que identifiquem toda a família de cenários implícitos - O objetivo deste trabalho é identificar os casos de teste que falham com relação à presença dos cenários implícitos, ou seja, como resultado teremos os casos de teste que identifiquem a família de cenários implícitos;

Na descrição da metodologia, os passos de 1 a 3 são realizados utilizando a ferramenta LTSA-MSC. O passo 4 é uma avaliação manual, ou seja, o usuário, a partir do cenário implícito devolvido pelo LTSA-MSC verificará, por meio do uso da metodologia apresentada em 3.1, a comunicação responsável por gerar o cenário implícito. O passo 5 utiliza a representação do cenário implícito definida em 3.1 para definir o propósito de teste. Para a realização dos passos de 6 e 7 foi desenvolvido um plug-in para a ferramenta LTSA-MSC. O mesmo é responsável por descobrir a família de cenários implícitos e gerar os casos de teste que os detectem.

Abordaremos a seguir, de forma mais detalhada, os passos necessários para a definição do propósito (assertivas) de teste - passos 5 e 6 - e a automação da geração do casos de teste, passos 7 e 8.

Análise do Cenário Implícito e Definição das Assertivas de teste - Passos 5 e 6

Alexandre *et al.* [5] identificaram que falhas geradas a partir de cenários implícitos podem acontecer principalmente em virtude do *interleaving* (intercalações) de ações críticas em comunicações assíncronas. Dessa forma, podendo-se caracterizar como condição de corrida [49] entre dois componentes ou um grupo de componentes.

Para identificar os caminhos (*traces*) que são responsáveis por gerar o cenário implícito, que foi formado devido à presença de condições de corrida, basta identificar a sequência de mensagens que estão fora de ordem no cenário implícito identificado pelo LTSA-MSC.

Para a representação do propósito de teste que identifica a família de um cenário implícito, definimos o seguinte padrão para representar a sequência de mensagens que constituem o cenário implícito: $msg1, msg2$

Essa sequência de mensagens identifica a condição de corrida onde a mensagem *msg1* é seguida da mensagem *msg2*. Por exemplo, no caso do sistema de gerenciamento de caldeira seria representado por: *on, query*. Representando o *trace* que identifica o cenário implícito, conforme a Figura 2.4.

No caso em que a condição de corrida é caracterizada pelo *interleaving* das mensagens entre três ou mais componentes, torna-se necessário representar as variações de ocorrência da condição de corrida em virtude do *interleaving* entre as mensagens. Esses casos podem acontecer, em particular, nos modelos de chamada *call-return*, onde um componente central deve decidir as ações a serem executadas a partir do retorno das ações dos componentes adjacentes que participam da chamada.

Suponha o caso do cenário, representado na Figura 3.2, com os componentes simulando as trocas de mensagem da seguinte forma: $C1 \rightarrow C2$ e $C1 \rightarrow C3$ e $C2 \rightarrow C1$ e $C3 \rightarrow C1$. Somente após o retorno das mensagens de C2 e C3 para C1, uma próxima troca de mensagens entre C2 e C3 pode ser executada. Nesse caso, a condição de corrida poderia acontecer no caso em que C2 e C3 comunicam-se antes de C1 receber as mensagens de C2 e de C3.

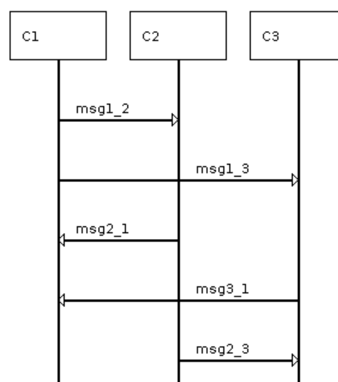


Figura 3.2: Troca de mensagens onde uma potencial condição de corrida pode acontecer. No caso, $msg2_3$ acontecer antes de $msg2_1$ ou $msg3_1$

A definição do propósito de teste que detecta a família de um cenário implícito, conforme o comportamento dos componentes, acima explicado, é elaborado da seguinte forma em nossa metodologia:

$msg2_1,\ msg2_3,\ msg3_1 \mid\mid msg3_1,\ msg2_3,\ msg2_1$

Onde *msg2_3* caracteriza-se como condição de corrida antes do retorno da mensagem de C2 para C1 (*msg2_1*) ou (||) antes do retorno da mensagem de C3 para C1 (*msg3_1*).

O propósito de teste serve como base para a identificação do cenário de teste. O cenário de teste é um cominho finito que contém os *traces* do propósito de teste.

Geração e Execução dos Casos de Testes - Passos 7 e 8

Para a obtenção dos casos de teste que identificam a família de cenários implícitos a partir do propósito de teste extraído do cenário implícito identificado pelo LTSA-MSC, definimos um algoritmo (Algoritmo 1). O Algorítmo 1 recebe como entrada o LTS do sistema e o propósito de teste, ou IOLTS usado para se gerar o cenário de teste. Então o algorítmo percorre o LTS como um grafo a fim de identificar o cenário de teste, um cenário que possui o propósto de teste, e a família do cenário implícito negativo detectado. Contudo esse LTS pode conter ciclos (loops), o que torna inviável representar todos os caminhos existentes que contenham os traces do propósito de teste, uma vez que este problema é caracterizado como problema NP-Completo [26]. Para tanto, definimos uma heurística (Algoritmo 2) onde estendemos o Algoritmo DFS (Depth-First Search) com o objetivo de tratarmos a presença presença dos ciclos. A seguir explicamos em detalhes tais algoritmos.

Em primeiro lugar, definimos o algoritmo que identifica o cenário de teste, realiza a busca pela família do cenário implícito no LTS e que provê a geração e a execução dos casos de teste, o Algoritmo 1 - *Test Generation*. Esse algoritmo primeiramente percorre o LTS para identificar todos os loops presentes no LTS em questão, caso exista. Esse procedimento é feito por meio da chamada ao Algoritmo $DFSLoop$ (linha 2). A identificação dos ciclos existentes tem por objetivo minimizar a quantidade de caminhos a serem percorridos, no LTS, para se identificar a família de cenários implícitos, evitando assim que o algorítmo possa gerar infinitos caminhos que possuem um cenário implícito.

Em seguida, são realizadas duas chamadas ao Algoritmo $Dijkstra$ a fim de se obter o cenário de teste, que será representado pelo menor caminho que contém o propósito de teste. A primeira chamada ao Algoritmo $Dijkstra$ é armazenada na variável $D1$ (linha 3). O caminho do LTS armazenado em $D1$ se inicia no nó inicial do LTS e vai até o nó que começa o cenário implícito, o nó em que se inicia o propósito de teste. A segunda chamada ao Algoritmo $Dijkstra$ é armazenada em $D2$ (linha 4). O caminho do LTS armazenado em $D2$ se inicia no nó final do cenário implícito, nó final do propósito de teste, e vai até o estado final do LTS. Daí, então, é realizada uma concatenação entre $D1$, a sequência de mensagens do cenário implícito representada pelo propósito de teste e $D2$ (linha 5), a fim de se obter o menor caminho ($CIminimo$) que possui o cenário implícito, ou seja,

Algorithm 1 TEST GENERATION

```
1: function TestImpliedScenarios(G, VetNo[])
2:     LinkedList L ← DFSLoop(G, 0);
3:     D1 ← DIJKSTRA(G, ni, VetNo[0])
4:     D2 ← DIJKSTRA(G, VetNo[length − 1], nf)
5:     CIminimo ← D1 + VetNo[] + D2
6:     for all (v ∈ CIminimo) do
7:         if (∃ L|inicio(L) == v) then
8:             cont ← 0
9:             for all (i ∈ L) do
10:                v ← v ∪ L(i)
11:                VetCI[cont] ← CIminimo
12:                cont + +
13:            end for
14:        end if
15:    end for
16:    return VetCI
17: end function
```

o cenário de teste. A partir do cenário de teste é identificada toda a família do cenário implícito negativo a partir da união dos ciclos retornados pelo Algorítmo *DFSLoop* com o $CIminimo$ (linhas 6 a 15). Na linha 16 são retornados todos os casos de teste que falham com relação a presença dos cenários implícitos.

O Algorítmo *DFSLoop* (Algoritmo 2), por sua vez, implementa a busca em profundidade no LTS. Em essência, o algoritmo estende o tradicional DFS [26] e contorna a questão dos loops infinitos do grafo armazenando em uma lista todos os ciclos presentes no LTS. O Algoritmo 2 basicamente realiza uma busca em profundidade no LTS e a cada nó que o algoritmo visita ele o adiciona em uma lista de nós visitados. Caso o Algorítmo *DFSLoop* visite um nó que já esta em sua lista de nós visitados ele então identifica um ciclo e armazena esse caminho, o processo até que não existam mais nós a serem visitados.

O Algoritmo 2 recebe como parâmetro de entrada um grafo G, que representa o LTS, e o vértice que será visitado v. Na linha 5, o algoritmo marca o vértice v como um vértice visitado. Em seguida, ele verifica se os vértices que são adjacentes a v, representados pela variável x, ainda não foram visitados (linha 7). Caso ele ainda não tenha sido visitado, ele é armazenado em um vetor de vértices visitados e a função $DFSLoop(G, x)$ é chamada novamente (linhas 7 a 10). Caso o nó x já tenha sido visitado o algoritmo percorre o LTS para armazenar o ciclo (linhas 12 a 20). Em seguida o algoritmo verifica se existe alguma lista que se inicie por x (linha 21), caso exista ele adiciona o ciclo à lista (linha 22), caso contrário, ele cria uma nova lista e adiciona o ciclo (linhas 23 a 26).

A Tabela 3.1 apresenta a complexidade de tempo e espaço do Algoritmo 1. Em seu

Algorithm 2 DFSLoop

```
1: cont ← 0                                              ▷ Contador
2: i ← 0                                                 ▷ Contador
3: VetVv[cont] ← v                                       ▷ Vetor de vértices visitados
4: function DFSLoop(G,v)
5:     v.dfs ← dfs.counter++
6:     for all (edge(x→v)) do
7:         if (x.dfs == −1) then
8:             cont++
9:             VetVv[cont] ← x
10:            DFSLoop(G, x)
11:        else
12:            for (j ← 0, j < n, j++) do
13:                if (VetVv[j] == x) then
14:                    k ← 0
15:                    while j < contn do
16:                        j++
17:                        VetAux[k] ← VetVv[j]
18:                        k++
19:                    end while
20:                    VetAux[k + 1] ← x
21:                    if (∃ LinkedList L | inicio(L) == x) then
22:                        L.listInsert(L, Vet)
23:                    else
24:                        New LinkedList L
25:                        L.listInsert(L, x)
26:                        L.listInsert(L, VetAux)
27:                    end if
28:                end if
29:            end for
30:        end if
31:    end for
32:    if (y.dfs == −1) then
33:        DFSLoop(G, y)
34:    else
35:        return L
36:    end if
37: end function
```

pior caso, o Algoritmo 1 funciona em tempo polinomial O(V.E), sendo "V"o número de vértices e "E"o número de arestas em um grafo. A complexidade desse algoritmo é definida em função dos algoritmos de Dijkstra [26] e o DFSLoop (Algoritmo 2). A Tabela 3.1 apresenta também a complexidade de tempo e espaço do DFSLoop, mostrando sua viabilidade computacional.

Algoritmo	**Complexidade de Tempo**	**Complexidade de espaço**
Dijkstra	O(E + V log V)	O(V + E)
DFSLoop	O(V.E)	O(V + E)
TestImpliedScenarios	O(V.E)	O(V + E)

Tabela 3.1: Complexidade dos algoritmos utilizados

3.2 Aplicação da Metodologia

Para ilustrar a aplicação da metodologia proposta neste trabalho, e detalhar suas etapas, utilizaremos um sistema simples, que já foi amplamente explorado em outros trabalhos relacionados com a detecção de cenários implícitos. Trata-se de um sistema chamado *boiler*, um armazenador térmico de água, composto por um sensor de vapor que detecta a presença de vapor e envia as informações para um módulo de controle que aciona um mecanismo para ativar ou desativar o sistema de aquecimento de água. Um importante requisito para o sistema é a necessidade do nível de água desse reservatório se manter em uma determinada faixa de valores, de modo que não fique com níveis altos ou baixos. Se esse requisito não for cumprido, o sistema da caldeira pode sofrer sérios danos [4]. A Figura 2.1 ilustra o referido sistema:

3.2.1 Modelagem do Sistema em Cenários

Para modelar o sistema em cenários utilizaremos a linguagem MSC, já explicada anteriormente. Baseado na descrição apresentada acima, temos 4 componentes que serão utilizados para a modelar esse sistema. São eles: Controlador (*Control*), Sensor, Atuador (*Actuator*) e o Banco de Dados (*Database*), para armazenar os dados enviados pelo sensor.

Na Figura 2.2 é apresentada a modelagem em cenários utilizada para o experimento, e na Figura 2.3 é apresentado o hMSC que representa todas as possíveis iterações entre os componentes do sistema. Com a modelagem do sistema em cenários o próximo passo é a geração do modelo comportamental do sistema.

3.2.2 Geração do Modelo Comportamental

Para a gerar o modelo comportamental do sistema será utilizado a ferramenta LTSA-MSC que recebe como entrada a modelagem do sistema em cenários na linguagem MSC. Dentre os resultados apresentados pela ferramenta está o modelo comportamental do sistema, expresso na linguagem FSP e o LTS do modelo. A Figura 3.3 apresenta o LTS resultante da composição de todos o cenários do sistema. A Listagem 3.1 exibe o FSP, gerado pelo LTSA-MSC, o qual retrata o comportamento da aplicação.

Listing 3.1: Modelo comportamental do sistema de gerência de caldeira expresso em FSP

```
ArchitectureModel = Q0,
Q0      = (ct.ss.on -> Q1),
Q1      = (ss.db.pressure -> Q2),
Q2      = (ss.db.pressure -> Q2
        | ct.db.query -> Q3
        | ct.ss.off -> Q9),
Q3      = (db.ct.data -> Q4),
Q4      = (ss.db.pressure -> Q5
        | ct.ac.command -> Q6),
Q5      = (ct.ac.command -> Q2
        | ss.db.pressure -> Q5),
Q6      = (ss.db.pressure -> Q2
        | ct.ss.off -> Q7),
Q7      = (ct.ss.on -> Q1
        | ct.ss.shutDown -> Q8),
Q8      = STOP,
Q9      = (ct.ss.shutDown -> Q10
        | ct.ss.on -> Q12),
Q10     = (endAction -> Q11),
Q11     = END,
Q12     = (ss.db.pressure -> Q2
        | ct.db.query -> Q13),
Q13     = (db.ct.data -> Q14),
Q14     = (ct.ac.command -> Q1
        | ss.db.pressure -> Q5).
```

Figura 3.3: Modelo arquitetural do sistema extraído do LTSA-MSC [18] .

3.2.3 Identificação dos Cenários Implícitos

Outro resultado produzido pelo LTSA-MSC é a identificação dos cenários implícitos existentes no software. O objetivo desse passo é avaliar os cenários implícitos produzidos pela ferramenta para posteriormente gerar os respectivos casos de teste a partir da geração da família de cenários implícitos negativos.

A Figura 2.4 apresenta um dos cenários implícitos detectados pela ferramenta. O mesmo será submetido a um processo manual de avaliação para identificá-lo como um cenário implícito positivo ou negativo.

Em [4] esse cenário implícito foi catalogado como negativo, pois, como já explicado anteriormente, o que acontece na Figura 2.4 é: O controlador liga o sensor com uma mensagem *on* e o sensor retorna uma mensagem *pressure* ao banco de dados com o valor da pressão da caldeira naquele momento, em seguida o sensor é desligado pelo controlador por uma mensagem *off*. Posteriormente o sensor é novamente ligado pelo controlador por meio de outra mensagem *on* e em seguida o controlador realiza uma consulta ao banco de dados, por meio de uma mensagem *query*, a fim de se obter a temperatura medida pelo sensor naquele instante, contudo o banco de dados possui uma temperatura que pode estar defasada, pois foi medida na primeira vez que em que o sensor foi ligado. Essa consulta então pode retornar um valor de temperatura que pode não condizer com a real temperatura da caldeira naquele momento, caracterizando assim o cenário implícito.

3.2.4 Identificação dos Caminhos que são Responsáveis pelo Cenário Implícito Negativo e Definição do Propósito de Teste

O LTSA-MSC tem por objetivo a identificação de todos os cenários implícitos existentes. Contudo, existem as família de cenários implícitos, que podem ser infinitos caminhos que levam ao mesmo estado de falha do sistema.

O sistema de caldeira apresenta outros cenários detectados além do cenário descrito na Figura 2.4. Contudo, eles são pertencentes à mesma família de cenários implícitos, pois os mesmos possuem o mesmo conjunto de caminhos (*traces*) responsáveis pela formação do cenário Implícito, constituído pela mensagem *on* seguida de *query*.

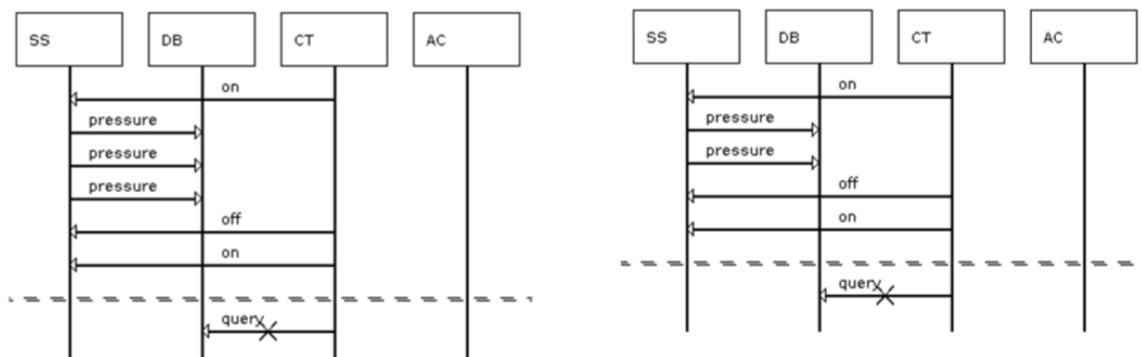


Figura 3.4: Outros cenários implícitos negativos identificados para o sistema de gerência de caldeira, *Boiler*.

Em nosso sistema temos como uma das entradas os traces que são responsáveis pela formação do cenário implícito, a fim de se identificar toda a família de cenários implícitos.

Analisando a Figura 2.4, verifica-se que a condição de corrida se dá devido à consulta a um valor de temperatura que pode estar defasado, presente em seu banco de dados. Logo, a condição de corrida se caracteriza pela mensagem de consulta ao banco de dados, *query*, chegar antes da mensagem de verificação da pressão junto ao sensor, *pressure*. Para identificamos unicamente essa família de cenários implícitos basta utilizarmos o propósito de teste: *on, query*.

Esse propósito de teste serve como entrada à metodologia a fim de se obter os casos de teste que identificam a família de cenários implícitos. A outra entrada será o modelo arquitetural do sistema, expresso em FSP.

3.2.5 Encontrar a Família de Cenários Implícitos e Prover a Geração dos Casos de Teste

Para encontrar a família de cenários implícitos e realizar a geração dos casos de teste serão utilizados os algoritmos descritos na Seção 3.1. O Algoritmo 1 que recebe como entrada o grafo G e um vetor, $VetNo$, com todos os nós que formam o cenário implícito. O LTS do sistema em questão, que é representado pelo FSP da Listagem 3.1 e o vetor com os nós responsáveis pela formação do cenário implícito, é obtido a partir do propósito de teste.

Em seguida, o Algoritmo 1 realiza uma chamada ao Algoritmo 2, responsável por identificar todos os laços presentes no sistema Boiler. Os ciclos identificados são representados na Tabela 3.2. Em seguida obtém-se o cenário de teste ($CIminimo$), representado pelo menor caminho no LTS que contém o propósito de teste. O cenário de teste é formado pela comunicação: 0-1-2-9-12-13-14-1-9-10-E. O LTS dessa comunicação é apresentado na Figura 3.5. O *CIminimo* e usado para encontrar a família de cenários implícitos e realizar a geração dos casos de teste.

Nó que Gera o laço	**laços Identificados**
1	2-3-4-6-7-1
1	2-9-12-13-14-1
2	2
2	3-4-5-2
2	3-4-6-2
2	9-12-2
5	5
5	2-9-12-13-14-5

Tabela 3.2: laços presentes no sistema *Boiler*

Após descobrir o menor caminho que contém o cenário implícito, o Algoritmo 1 realiza a união entre o cenário de teste e os resultados do DFSLoop (vide Tabela 3.2) para compor o resultado final constituindo assim, a família de cenários implícitos. A Tabela 3.3 demonstra todos os casos de teste gerados a partir da identificação da família de cenários implícitos. Nessa tabela são apresentados apenas os casos de teste que falharam com relação à presença dos cenários implícitos.

Salientamos também, que no caso do sistema de caldeira existe um estado final que não leva ao estado terminal E, como ilustrado na Figura 3.3, no entanto observa-se, na mesma figura, que existe um outro estado terminal, representado pelo nó número *8*, que é um estado de *deadlock*, situação na qual dois ou mais processos, ou componentes, ficam impedidos de continuar suas execuções devido alguma situação de impasse [49]. Esse

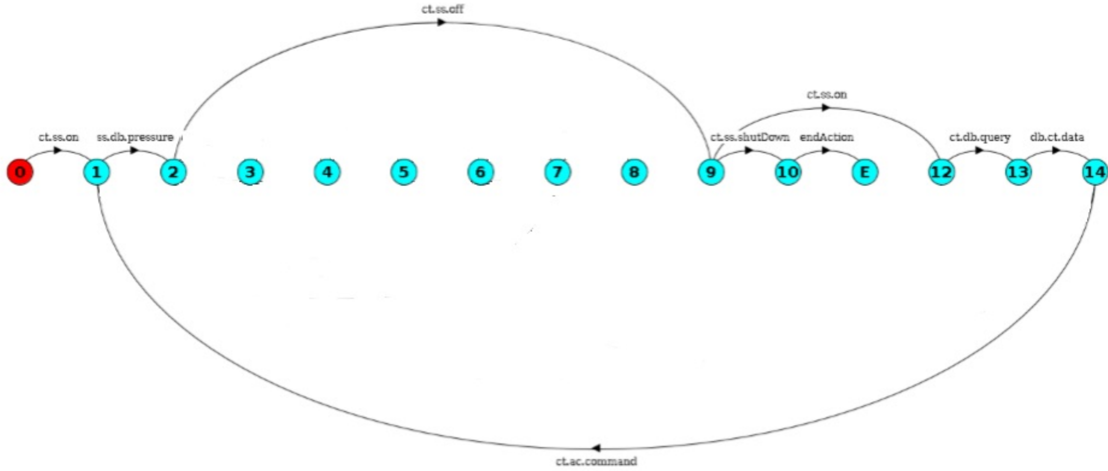


Figura 3.5: Cenário Implícito Mínimo.

laços	Caminhos
Sem laços	0-1-2-**9-12-13**-14-1-2-9-10-E
2-3-4-5-1	0-1-(2-3-4-5-1)*-2-**9-12-13**-14-1-2-9-10-E
2-3-4-6-7-1	0-1-(2-3-4-6-7-1)*-2-**9-12-13**-14-1-2-9-10-E
2-9-12-13-14-1	0-1-(2-9-12-13-14-1)*-2-**9-12-13**-14-1-2-9-10-E
2	0-1-(2)*-2-**9-12-13**-14-1-2-9-10-E
3-4-5-2	0-1-2-(3-4-5-2)*-**9-12-13**-14-1-2-9-10-E
3-4-6-2	0-1-2-(3-4-6-2)*-**9-12-13**-14-1-2-9-10-E
9-12-2	0-1-2-(9-12-2)*-**9-12-13**-14-1-2-9-10-E

Tabela 3.3: Caminhos que possuem o cenário implícito extraídos do sistema *Boiler* a partir do Algoritmo 1, utilizando como estado final *E*. Os *traces* em negrito são os responsáveis por gerar o cenário implícito.

deadlock é representado na Listagem 3.1 por uma mensagem *STOP*. Uma vez que o foco do nossos casos de teste é com base nos cenários implícitos e não em deadlock, sem perda de generalidade, representamos esse estado como se o mesmo convergisse ao estado final de sucesso E. Dessa forma, ainda que hajam caminhos que levem ao deadlock e que façam parte dos caminhos da família de cenário implícito, eles também serão evidenciados na geração dos caso teste.

Então para o sistema em questão teríamos, para a mesma família de cenários implícitos, dois estados finais, os que levam a *E* e os que levam ao *STOP*. Logo o Algoritmo 1 seria chamado duas vezes para compor o resultado, pois existem caminhos que levam ao estado de *deadlock* e que possuem os *traces* responsáveis pela formação do cenário implícito.

Os laços identificados pelo Algoritmo 2 serão os mesmos apontados na Tabela 3.2, pois

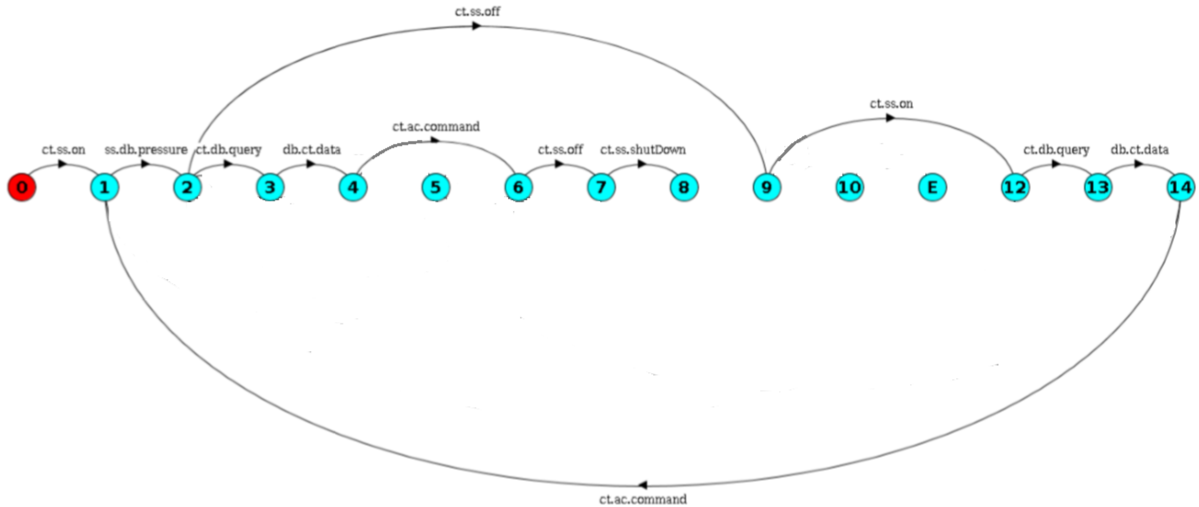


Figura 3.6: Menor caminho no grafo que contém o cenário implícito levando ao estado *STOP*.

se trata do mesmo sistema. Contudo, o caminho que leva do estado inicial ao estado final seria diferente, conforme apresentado na Figura 3.6.

A suite de testes gerada para o sistema de caldeira é representado pela união dos resultados das Tabelas 3.3 e 3.4. A verificação dos estados finais do sistema faz parte da automatização provida para essa metodologia, deixando assim o processo menos sucetível a erros desse gênero.

Laços	**Caminhos**
Sem Laços	0-1-2-**9-12-13**-14-2-3-4-6-7-8-STOP-E
2-3-4-5-1	0-1-(2-3-4-5-1)*-2-**9-12-13**-14-2-3-4-6-7-8-STOP-E
2-3-4-6-7-1	0-1-(2-3-4-6-7-1)*-2-**9-12-13**-14-2-3-4-6-7-8-STOP-E
2-9-12-13-14-1	0-1-(2-9-12-13-14-1)*-2-**9-12-13**-14-2-3-4-6-7-8-STOP-E
2	0-1-(2)*-2-**9-12-13**-14-2-3-4-6-7-8-STOP-E
3-4-5-2	0-1-2-(3-4-5-2)*-**9-12-13**-14-2-3-4-6-7-8-STOP-E
3-4-6-2	0-1-2-(3-4-6-2)*-**9-12-13**-14-2-3-4-6-7-8-STOP-E
9-12-2	0-1-2-(9-12-2)*-**9-12-13**-14-2-3-4-6-7-8-STOP-E

Tabela 3.4: Caminhos que possuem o cenário implícito extraídos do sistema *Boiler* a partir do Algoritmo 1, e utilizando o *STOP* convergindo ao estado final *E*. Os *traces* em negrito são os responsáveis por gerar o cenário implícito.

Todo o processo de geração e execução dos casos de teste descrito foi automatizado e integrado à ferramenta LTSA-MSC na forma de um *Plug-in* chamado *Test Generation - TG*.

Plug-in *Test Generation - TG*

Como entrada o *plug-in TG* recebe, assim como o Algoritmo 1, o cenário implícito, representado pelo propósito de teste, e o grafo do sistema, representado em FSP e gerado pelo LTSA-MSC. O *TG* produz como resultado o conjunto de casos de teste que falharam com relação à presença dos cenários implícitos, ou seja, ele retorna os casos de teste que identificam a família de cenários implícitos.

O modelo comportamental, expresso em FSP, é obtido automaticamente da ferramenta LTSA-MSC, cabendo ao usuário inserir somente o propósito de teste. A Figura 3.7 mostra a interface do *plug-in TG* com o modelo comportamental da aplicação, descrito em FSP, e o pedido de entrada do propósito de teste.

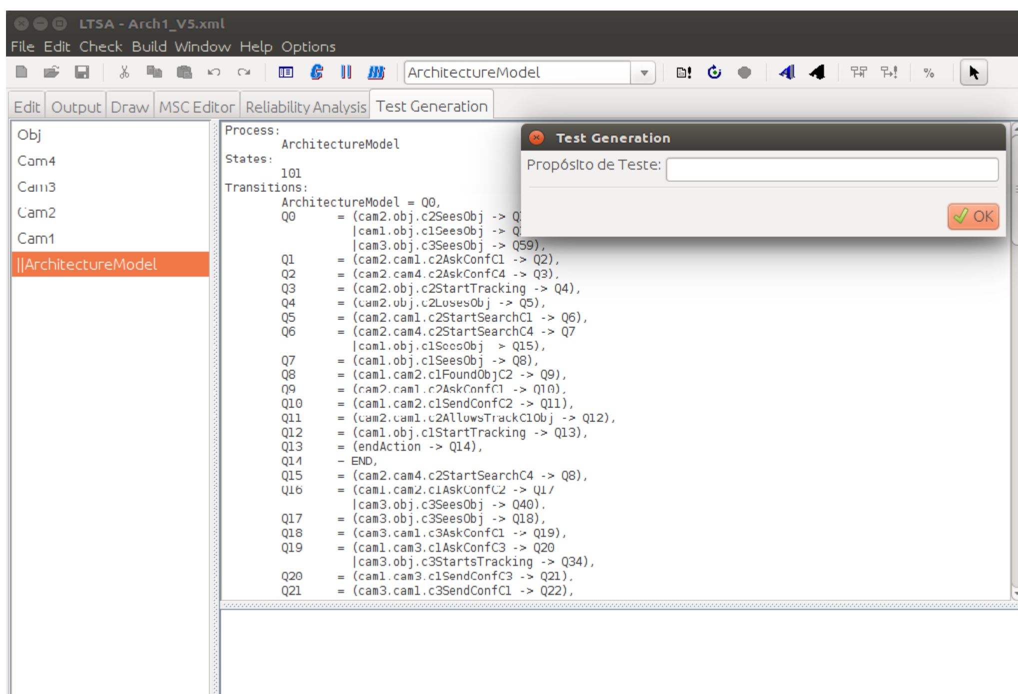


Figura 3.7: Interface do *plug-in Test Generation -TG*

Capítulo 4

Estudo de Caso

Neste capítulo será apresentado um estudo de caso baseado na metodologia definida no Capítulo 3. Para este estudo de caso foi utilizado o sistema de câmeras inteligentes, *Smart Cams*, que compartilham a responsabilidade pelo rastreamento de objetos que entram em seu campo de visão. O estudo de caso seguirá a estrutura defendida em [54]. Dessa forma, o estudo de caso deverá deixar bem definidos os seguintes aspectos:

- Contexto da pesquisa;
- Hipóteses a serem validadas;
- Configuração do experimento;
- Análise e apresentação de dados;
- Resultados e conclusões;
- Ameaças à validade.

4.1 Contexto da Pesquisa

Conforme já tratado em pontos anteriores desse trabalho, essa pesquisa foca na geração de casos de teste a partir de um cenário implícito identificado, objetivando-se descobrir o real impacto da família de cenários implícitos em um sistema concorrente, por meio da suite de testes gerada. Com uma perspectiva mais real da extensão da falha gerada pode-se tomar decisões mais precisas quanto a sua correção.

Nesse estudo de caso, busca-se, especificamente, utilizar esses casos de teste como subsídio para uma maior cobertura de casos de teste. Proporcionado assim meios para se prover uma maior confiabilidade, e assim se obter um sistema que possa alcançar um nível maior de dependabilidade.

4.2 Sistema *Smart Cams*

O sistema *Smart Cams* é um projeto de pesquisa entre universidades *Alpen-Adria-Universität Klagenfurt*, da Áustria, e a *University of Birmingham*, da Inglaterra. Esse projeto foi apresentado preliminarmente em [50] e, integralmente em [51].

De forma resumida, esse é um sistema descentralizado, pois não possui um nó central de controle. O controle é realizado pelas próprias câmeras que estabelecem uma comunicação entre si a fim de realizar esse controle. As câmeras não tem conhecimento da topologia completa da rede, conhecendo somente as câmeras que estão em sua vizinhança, tornando assim mais fácil alterar o arranjo da rede. Com isso, torna-se mais fácil a adição ou remoção de novas câmeras em tempo de execução, e proporcionando um menor impacto no relacionamento entre elas.

Foi proposto em [50] um algoritmo para orientar a forma de como as câmeras revezam-se na tarefa de rastrear os objetos. Esse algoritmo visa uma utilização eficiente de recursos, buscando otimizar a comunicação entre os componentes. Os autores definem essa abordagem como sendo sócio-econômica por seguirem os mecanismos de mercado.

Descrição do Sistema *Smart Cams*

Descreveremos nessa seção, de forma resumida, o funcionamento do sistema. Essa descrição foi extraída de [50]. No sistema de câmeras inteligentes cada câmera do sistema pode se comunicar somente com as câmeras vizinhas, ou seja as câmeras que estão fisicamente próximas entre si, de modo que possam trocar informações. Quando uma câmera entra no sistema ela envia mensagens, em *broadcast*, na rede a fim de se obter respostas de quais câmeras seriam suas vizinhas, alterando, assim, a topologia da rede de forma dinâmica.

Nesse sistema, cada câmera gerencia duas listas: uma de rastreamento (que indica os objetos sob sua responsabilidade) e outra de busca (que indica os objetos que ela deve buscar). A presença de um objeto em sua lista de busca significa que não existem câmeras vizinhas rastreando o objeto no momento ou, que a responsável pelo rastreamento está prestes a perdê-lo de vista. Em outras palavras, o objeto está prestes a deixar o campo de visão da câmera.

Existe um agente central nesse sistema responsável por informar às câmeras a necessidade de inserção ou retirada de objetos em suas respectivas listas de busca. Então, se um novo objeto é inserido no sistema, o agente central envia às câmeras a ordem para que o coloquem em suas listas de busca.

Quando um novo objeto entra no campo de visão de alguma câmera, a mesma envia às suas vizinhas mensagens para que o retirem de suas listas de busca. Nesse momento

ela passa a ser a responsável pelo rastreamento do objeto, incluindo-o em sua lista de rastreamento.

Se uma câmera está prestes a perder um objeto por ele estar saindo do seu campo de visão, ela envia mensagens às câmeras vizinhas para que o objeto seja incluído nas respectivas listas de busca. As câmeras que receberam a mensagem e possuem o objeto no seu campo de visão realizam um "leilão"a fim de se decidir que câmera irá rastrear o objeto.

Para realizarem esse leilão, as câmeras que tem o objeto em seu campo de visão enviam mensagens para a câmera responsável, a que perderá a visão do objeto em questão. A câmera responsável inicia então o leilão requisitando às câmeras que estão visualizando o objeto os seus índices de confiança, valores que representam o quanto o objeto está visível para determinada câmera. Após o recebimento dos índices de confiança das câmeras participantes do leilão, a câmera responsável transmite a responsabilidade de monitorar o objeto à câmera que apresentou o maior índice, significando que esta seria a mais bem posicionada para rastrear o objeto.

A câmera que recebeu esta responsabilidade inclui o objeto em sua lista de rastreamento e envia uma mensagem as suas vizinhas para que retirem o objeto de suas respectivas listas de busca. Maiores detalhes do sistema são encontrados em [51], inclusive detalhes sobre as configurações do hardware das câmeras.

4.3 Hipóteses

O objetivo desse estudo de caso é gerar casos de teste, a partir do cenário implícito detectado, a fim de se obter a família de cenários implícitos e ter uma perspectiva mais real da extensão da falha gerada pela sua presença sobre a definição de um arranjo das câmeras. Na literatura existem dois trabalhos, [44] e [45], motivados por uma parceria entre a *Universidade de Brasília*, do Brasil, e a universidade de *Birminghan*, da Inglaterra, que utilizam os mesmos arranjos a fim de se avaliar o impacto da presença dos cenários implícitos, com relação a confiabilidade do sistema [45] e com relação às vulnerabilidades de segurança geradas [44].

Esses trabalhos apresentam duas arquiteturas, contendo quatro câmeras cada uma, e a arquitetura mais adequada, sob o ponto de vista da confiabilidade ou por apresentar menor número de falhas de segurança, deverá ser a escolhida. A análise conjunta sob os focos de confiabilidade e de segurança teve o objetivo de definir a melhor escolha entre as arquiteturas.

Visto que essas arquiteturas já foram avaliadas em trabalhos anteriores, elas também foram utilizadas no estudo de caso, pois todos os cenários implícitos presentes nas duas

arquiteturas já foram identificados e catalogados, o que nos permitirá validar nossa metodologia de forma fidedigna. Diante do exposto, as hipóteses a serem validadas nesse estudo de caso são:

- **Hipótese I** – Em um cenário real a metodologia apresentada se demosntra eficaz e completa, no sentido de cobrir de fato a identificação de todas as famílias de cenários implícitos negativos presentes na arquitetura.
- **Hipótese II** – Com a geração dos casos de teste é possível avaliar a quantidade de caminhos atingidos pelos cenários implícitos tendo assim uma percepção mais real da quantidade de falhas que podem ser geradas pela presença das famílias de cenários implícitos, tornando possível propor alternativas ao modelo para que as mesmas possam ser evitadas.

Para a validação das hipóteses listadas, o sistema *Smart Cams* foi submetido à metodologia apresentada nesse trabalho. De modo mais específico, as *Hipóteses I* e *II* serão validadas por meio da massa de dados gerada, pois cada caso de teste identifica pelo menos o menor caminho finito que contém o cenário implícito. Dado o conjunto de casos de tese, teremos várias possibilidades da ocorrência de falhas geradas pela presença dos cenários implícitos, demonstrado assim como o sistema pode ser afetado por essa anomalia.

Ao submeter o sistema *Smart Cams* à metodologia serão apresentados o conjunto de casos de teste que identificam todas as família de cenários implícitos presentes no sistema. Para validar a *Hipótese I* será realizada uma comparação entre os casos de teste encontrados com o uso da metodologia e os cenários implícitos identificados nos trabalhos [44] e [45], os quais já identificaram e catalogaram os cenários implícitos presentes nesse sistema. Será feita também uma análise manual da arquitetura do sistema, a fim de verificar se os casos de teste realmente identicam todas as famílias de cenários implícitos presentes na arquitetura.

Com a identificação de um cenário implícito podemos identificar uma única falha ocasionada por sua presença. Já identificando a família de cenários implícitos teremos identificado então um conjunto de casos em que ocorre esse mesmo tipo de falha. A *Hipótese II* será validada a partir da análise dos casos teste gerados, pois sabendo dos casos em que o sistema pode falhar será possível propor alternativas ao modelo para que as mesmas possam ser evitadas.

4.4 Configuração do Experimento

Para simular o sistema *Smart Cams* foi desenvolvido um simulador em [50]. Esse simulador tem o objetivo de produzir as iterações do algoritmo que orienta a troca de

mensagens entre os componentes do sistema, ou seja as câmeras. Essa análise provida pelo simulador fornecerá subsídios que acrescidos do conjunto de casos de teste gerados para o sistema serviram para a melhoria do sistema real.

Esse simulador foi implementado utilizando a linguagem Java, possuindo 21 classes com 7695 linhas de código. A análise realizada sobre o simulador visa trazer meios para se prover uma maior dependabilidade ao sistema desde os estágios iniciais do desenvolvimento até a sua manutenção. A interface do simulador é apresentada na Figura 4.1.

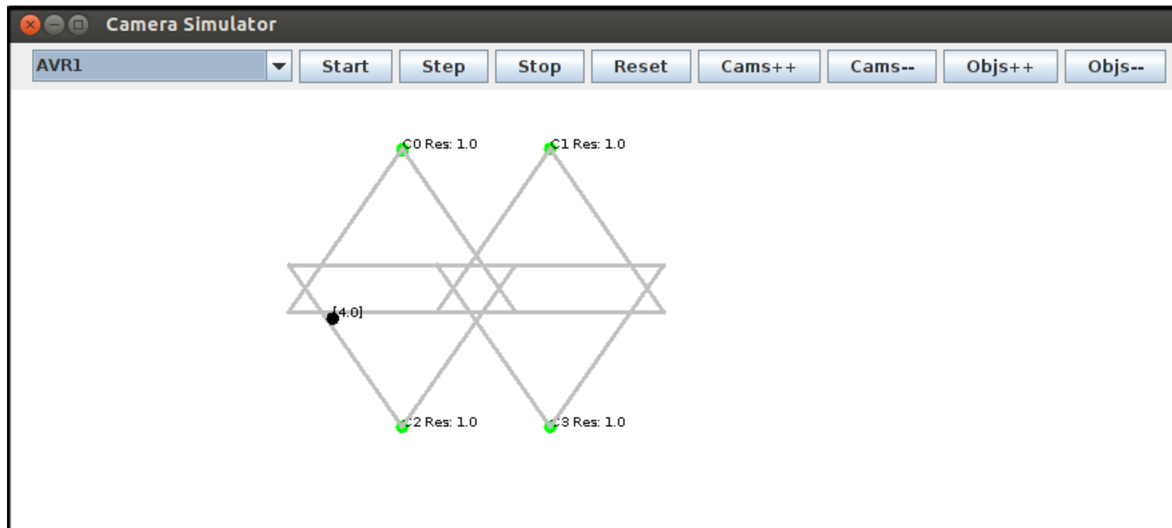


Figura 4.1: Interface do simulador do sistema *Smart Cams*.

Para as arquiteturas que serão estudadas, como já mencionado anteriormente, utilizaremos as arquiteturas utilizadas em [44] e [45]. Para a especificação dos cenários a serem modelados foi realizada uma análise minuciosa do código do simulador a fim de se identificar todas as mensagens trocadas entre os componentes do sistema *Smart Cams*. A lista a seguir mostra as mensagens que serão utilizadas na modelagem dos cenários:

- *Sees*: Mensagem que indica o momento em que o objeto entra no campo de visão de uma câmera;
- *StartTracking*: Mensagem que indica o momento em que uma câmera passa a ser a responsável por rastrear o objeto.
- *Loses*: Mensagem enviada às câmeras vizinhas pela câmera que atualmente rastreia o objeto, indicando o momento em que ele está prestes a deixar seu campo de visão.
- *Found*: Mensagem enviada pelas câmeras que receberam a mensagem *"Loses"*, e possuem o objeto em seu respectivo campo de visão;

- *StartSearch*: Mensagem enviada às câmeras vizinhas para que adicionem o objeto em suas listas de busca;
- *AskConfidence*: Mensagem enviada a uma câmera perguntando o índice de confiança dela com relação a visualização do objeto;
- *SendConfidence*: Mensagem utilizada para responder à pergunta *AskConfidence* enviada, enviando o índice de confiança como argumento.
- *AllowsTrack*: Mensagem enviada a uma câmera informando que ela é a nova responsável por rastrear o objeto;

As arquiteturas, extraídas de [44] e [45], são exibidas na Figura 4.2. Ambas possuem quatro câmeras e com a mesma configuração. A vizinhança das câmeras, ou seja, as câmeras que se comunicam entre si, é apresentado na Tabela 4.1.

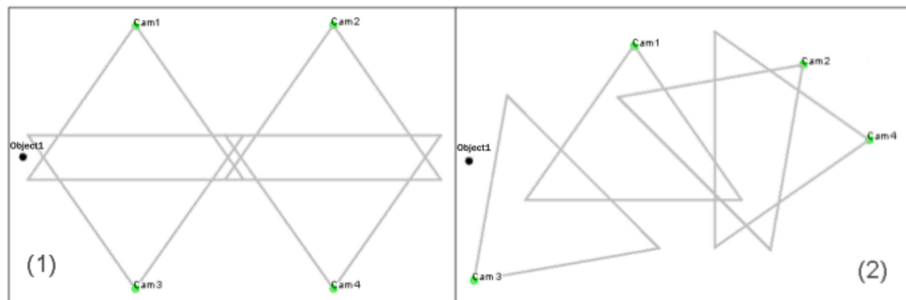


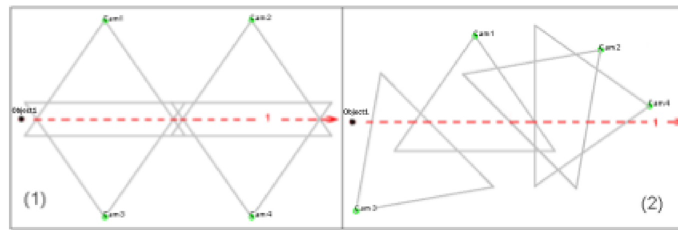
Figura 4.2: Arquiteturas a serem analisadas

Câmeras	**Vizinhos**
1	2,3
2	1,4
3	1
4	2

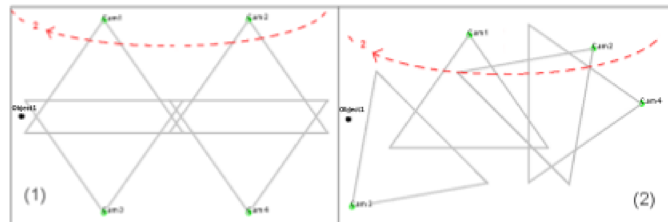
Tabela 4.1: Configuração da vizinhança de cada câmera do experimento.

A definição de vizinhança foi apresentada em [44]. Nessa configuração, para a modelagem do sistema em cenários foi proposto que se capturassem cenários a partir da trajetória do objeto na rede de câmeras. Então, foram configurados cinco cenários básicos para cada arquitetura dispostos de forma paralela.

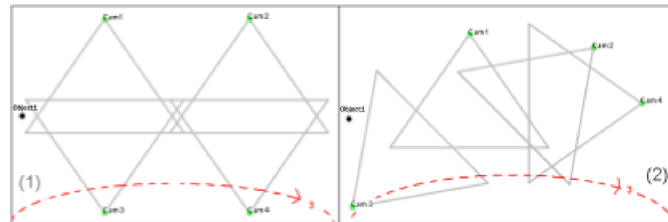
Cada cenário representa a trajetória de um objeto entre as câmeras, indicado pela linha tracejada em vermelho. A Figura 4.3 apresenta os diversos cenários utilizados em [44] e [45].



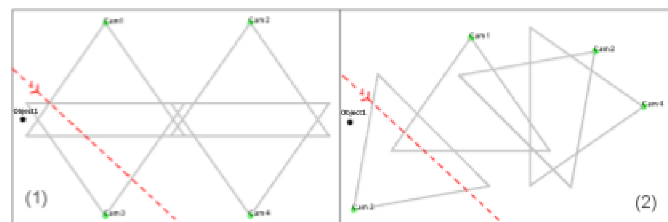
(a) Objeto atravessa o campo de visão das quatro câmeras tanto na Arch1 como na Arch2.



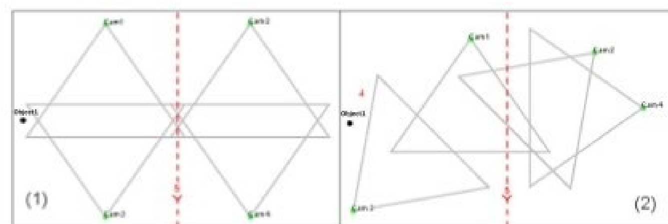
(b) Objeto atravessa o campo de visão das Câmeras 1 e 2 na Arch1 e das câmeras 4, 2 e 1 na Arch2.



(c) Objeto atravessa o campo de visão das câmeras 3 e 4 na Arch1 e das câmeras 3, 2 e 4 na Arch2.



(d) Objeto atravessa o campo de visão das câmeras 1 e 3 tanto na Arch1 quanto na Arch2.



(e) Objeto atravessa o campo de visão das quatro câmeras na Arch1 e das câmeras 1 e 2 na Arch2.

Figura 4.3: Cenários utilizados na modelagem [44]

As Figuras 4.4 e 4.5 apresentam os diagramas bMSCs para o cenário 1 das duas

arquiteturas. O conjunto completo dos diagramas bMSCs dos cenários especificados estão disponíveis no Anexo A.

Figura 4.4: Representação em MSC do cenário 1 da arquitetura 1

Figura 4.5: Representação em MSC do cenário 1 da arquitetura 2.

4.5 Análise e Apresentação dos Dados

Essa seção faz o uso da metodologia proposta no sistema de câmeras inteligentes, *Smart Cams*, a fim de validá-la demonstrado que a metodologia gera casos de testes que identificam todas as falhas ocasionadas pela presença dos cenários implícitos. A primeira etapa da metodologia envolve o uso da ferramenta LTSA-MSC e consiste na obtenção do modelo comportamental, obtido a partir dos cenários especificados, a identificação dos cenários implícitos.

A segunda parte consiste em uma avaliação manual dos cenários implícitos encontrados para se obter os cenários implícitos negativos e então definir o propósito de teste. A terceira etapa da metodologia será a utilização dos Algoritmos 1 e 2, em um processo totalmente automatizado, para gerar o conjunto de caso de teste que detectem todas as falhas geradas pela presença dos cenários implícitos. As seções seguintes detalham essas três etapas.

4.5.1 Obtenção do Modelo Comportamental

A partir dos cenários especificados em 4.3 foram gerados o bMSC referentes às duas arquiteturas. Por uma questão de forma os mesmos estão disponíveis no anexo A.

Dados os cenários modelados foram gerados os FSPs com o modelo comportamental das duas arquiteturas, sendo a arquitetura 1 com 95 estados e 122 mensagens trocadas e a arquitetura 2 com 70 estados e 115 mensagens trocadas. Também por uma questão de forma os FSPs das arquiteturas 1 e 2 estão disponíveis no anexo B.

4.5.2 Identificação dos Cenários Implícitos e Geração do Propósito de Teste

Conforme já apresentado anteriormente, o processo de identificação dos cenários implícitos, na LTSA-MSC, é iterativo, ou seja a cada execução pode ser apresentado um novo cenário para que o usuário informe se o mesmo é um cenário implícito positivo ou negativo. Como um dos objetivos propostos é prover meios de se aumentar a confiabilidade do sistema, então os cenários serão classificados como negativos quando seu comportamento estiver em desacordo com o comportamento esperado do sistema, descrito em seu documento de especificação de requisitos,

Utilizando LTSA-MSC foram identificados diferentes resultados, para as duas arquiteturas do sistema *Smart Cams*. Para a arquitetura 1, foram encontrados um total de quatro cenários implícitos, sendo todos eles negativos. Enquanto que para a arquitetura 2,

foram encontrados 52 cenários implícitos, sendo que apenas 1 classificado como negativo, o restante foram classificados como positivos.

Arquitetura 1

Nas Figuras 4.6, 4.7, 4.8 e 4.9 são apresentados os cenários implícitos negativos identificados para a arquitetura 1.

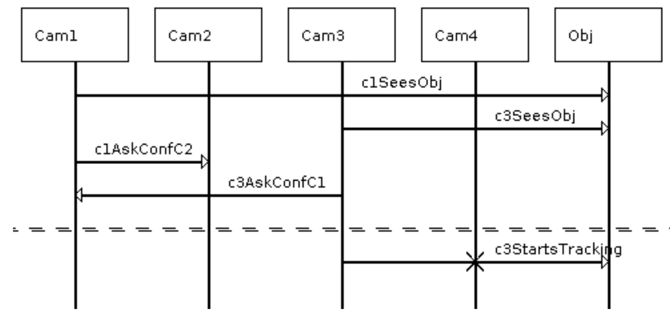


Figura 4.6: Primeiro cenário implícito negativo encontrado para a arquitetura 1 (IS1)

Analisando-se o cenário IS1 podemos constatar que o objeto entra no campo de visão das câmeras 1 e 3, os mesmos enviam suas respectivas mensagens *"Sees"*, identificando ao sistema que ambas possuem o objeto em seus respectivos campos de visão. Em seguida, as câmeras 1 e 3 enviam mensagens *"AskConfidence"* a sua vizinha, a fim de se obter o grau de confiabilidade que elas possuem ao enxergar o objeto. De acordo com a Tabela 4.1 a câmera 1 deveria enviar mensagens às câmeras 2 e 3, e a câmera 3 deveria enviar mensagens a 1 somente. Arbitrariamente a câmera 3 começa a monitorar o objeto, a partir da mensagem *"StartsTraking"*, sem aguardar a resposta da câmera 1, *"SendConfidence"*, com seu índice de confiança.

Partindo dessa análise do cenário implícito identificado, e utilizando os conceitos apresentados na seção 3.2.4, podemos chegar à conclusão que o cenário implícito em questão possui uma natureza de formação por mensagens assíncronas, pois mesmo a câmera 3 tendo solicitado o índice de confiança à câmera 1 ele não espera o índice chegar para decidir quem irá monitorar o objeto. Devido a sua natureza de formação, para identificar o cenário implícito em questão, temos que identificar as mensagens que são assíncronas, no mesmo, e verificar todas as possibilidades de composição das mensagens, a fim de se criar o propósito de teste. As mensagens responsáveis pela condição de corrida são:

$$c1AskConfC2,\ c3AskConfC1,\ c3StarsTracking$$

Esse problema se dá devido à câmera 3 começar a rastrear o objeto, através da mensagem *"c3StarsTracking"*, antes da resposta, *"SendConf"*, às mensagens *"c1AskConfC2, c3AskConfC1"*. Logo, para a formação desse propósito de teste devemos descrever todas as possibilidades de composição dessas mensagens, retirando-se os casos em que não existe a condição de corrida, como por exemplo o caso em que as mensagens *"SendConf"* chegam antes da mensagem *"c3StarsTracking"*. O propósito de teste, para o cenário implícito em questão é:

Listing 4.1: Propósito de Teste 1

```
c1AskConfC2, c3AskConfC1, c3StarsTracking || c1AskConfC2,
c2SendConfC1, c3AskConfC1, c3StarsTracking || c1AskConfC2,
c3AskConfC1, c2SendConfC1, c3StarsTracking || c1AskConfC2,
c3AskConfC1, c1SendConfC3,c 3StarsTracking
```

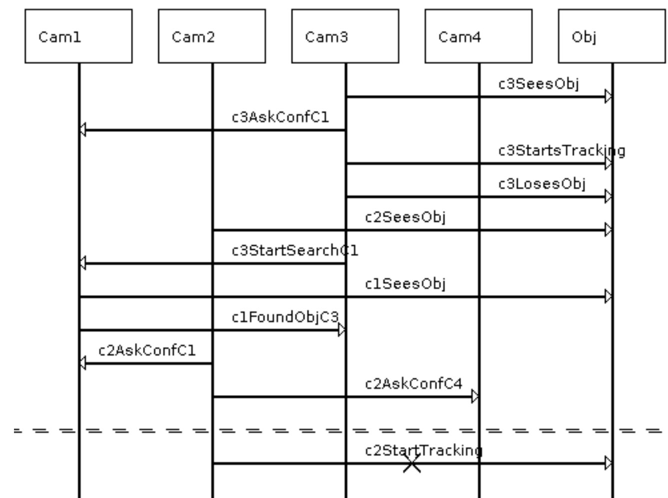


Figura 4.7: Segundo cenário implícito negativo encontrado para a arquitetura 1 (IS2)

Analisando o IS2 temos que a câmera 3 visualiza o objeto e envia uma mensagem *"Sees"* ao sistema dizendo que o objeto entrou em seu campo de visão. Em seguida manda uma mensagem *"AskConfidence"* à câmera vizinha, câmera 1, a fim de obter seu índice de confiança e inicia o rastreamento do objeto, com a mensagem *"StartsTraking"*, mesmo ainda não tendo chegado a resposta do índice de confiança, *"SendConfidence"* da câmera 1.

Após a mensagem *c3StarsTracking*, no IS2, a câmera 3 verifica que o objeto sairá, em breve, de seu campo de visão, e então envia ao sistema uma mensagem, *"Loses"*, indicando que não poderá continuar com o rastreamento. Simultaneamente à saída do objeto do

campo de visão da câmera 3, o mesmo entra no campo de visão da câmera 2, evidenciado pela mensagem *"Sees"* enviada pela câmera.

A câmera 3 envia uma mensagem *"StartSearch"* à câmera 1, sua única vizinha, a fim de saber se a mesma possui o objeto em seu campo de visão. Ao visualizar o objeto a câmera 1 envia uma mensagem *"Sees"* ao sistema informando que o possui em seu campo de visão, e em seguida responde a solicitação da câmera 3, com uma mensagem *"Found"*, informando que possui o objeto em questão em seu campo de visão.

A câmera 2, que já havia visualizado o objeto, envia então mensagens *"AskConfidence"* às câmeras 1 e 4, que são suas vizinhas para saber o seu grau de confiança na visualização do objeto. E seguida inicia o monitoramento do objeto antes de receber o índice de suas vizinhas.

A natureza de formação do IS2 é devido à ocorrência de uma condição de corrida, assim como no exemplo anterior. Então, para a definição das assertivas de teste deve-se utilizar o conjunto de mensagens responsáveis pela condição de corrida apresentada na Figura 4.7, que são:

$$c2AskConfC1,\ c2AskConfC4,\ c2StartTracking$$

O assincronismo se dá devido à câmera 2 começar a rastrear o objeto, através da mensagem *"c2StarsTracking"*, antes da resposta, *"SendConf"* das câmeras C1 e C4. Logo para a formação desse propósito de teste deve-se descrever todas as possibilidades de composição dessas mensagens, retirando-se os casos em que não existe a condição de corrida, como por exemplo o caso em que as mensagens *"SendConf"* chegam antes da mensagem *"c2StarsTracking"*. O propósito de teste, para o cenário implícito IS2 é:

Listing 4.2: Propósito de Teste 2

```
c2AskConfC1, c2AskConfC4, c2StartTracking || c2AskConfC1,
c1SendConfC2, c2AskConfC4, c2StartTracking || c2AskConfC1,
c2AskConfC4, c1SendConfC2, c2StartTracking || c2AskConfC1,
c2AskConfC4, c4SendConfC2, c2StartTracking
```

Analisando o IS3 temos que a câmera 3 avista o objeto e envia uma mensagem *"Sees"* ao sistema, indicando que o possui em seu campo de visão, então ela envia uma mensagem *"SendConfidence"* à câmera 1, sua única vizinha, a fim de se obter seu índice de confiança, então começa a monitorar o objeto, enviando uma mensagem *StartsTraking* ao sistema.

Após a mensagem *c3StartsTracking*, quando a câmera 3 está prestes a perder o objeto do seu campo de visão ela envia uma mensagem *"Loses"* ao sistema, indicando que outra

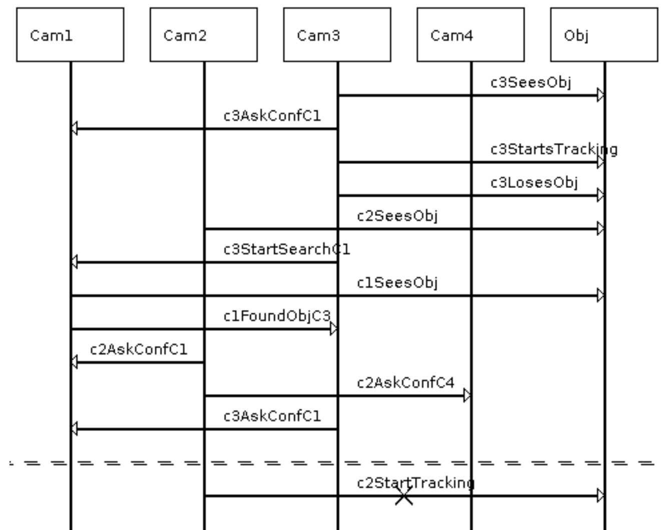


Figura 4.8: Terceiro cenário implícito negativo encontrado para a arquitetura 1 (IS3)

câmera deverá monitorá-lo. O objeto entra no campo de visão da câmera 2, que envia uma mensagem *"Sees"* ao sistema.

A câmera 3 envia uma mensagem *"StartSearch"* à câmera 1 para saber se a mesma, sua única vizinha, possui o objeto em seu campo de visão. A câmera 1 então avista o objeto e envia uma mensagem *"Sees"* ao sistema e uma mensagem *"FoundObj"* para a câmera 3, afirmando que o objeto está em seu campo de visão.

A câmera 2, que já havia sinalizado que possui o objeto em seu campo de visão, envia mensagens *"AskConfidence"* às sua vizinhas, as câmeras 1 e 4, para saber os seus respectivos índices de confiança. A câmera 3 manda também uma mensagem *"Ask Confidence"* à câmera 1 requisitando seu índice de confiança. Então a câmera 2 começa a monitorar o objeto por meio de uma mensagem *StarsTraking* ao sistema.

Assim como nos IS1 e IS2, a natureza de formação do IS3 está relacionada a presença de mensagens assíncronas. Logo, para a formação do propósito de teste temos que identificar as mensagens responsáveis pela condição de corrida no cenário IS3, representado na Figura 4.8. As mensagens são:

c2AskConfC1, c2AskConfC4, c3AskConfC1, c2StartTracking

A condição de corrida se dá devido à câmera 2 começar a rastrear o objeto, através da mensagem *"c2StarsTracking"*, antes da resposta, *"SendConf"*, às mensagens *"c2AskConfC1, c2AskConfC4, c3AskConfC1"*. Então, para a formação desse propósito de teste deve-se descrever todas as possibilidades de composição dessas mensagens, retirando-se os casos

em que não existe a condição de corrida, como por exemplo o caso em que as mensagens *"SendConf"* chegam antes da mensagem *"c2StarsTracking"*. O propósito de teste, para o cenário implícito IS3 é:

Listing 4.3: Propósito de Teste 3

```
c2AskConfC1,  c2AskConfC4, c3AskConfC1, c2StartTracking ||
c2AskConfC1, c1SendConfC2, c2AskConfC4, c3AskConfC1,
c2StartTracking || c2AskConfC1, c2AskConfC4, c1SendConfC2,
c3AskConfC1, c2StartTracking || c2AskConfC1, c2AskConfC4,
c3AskConfC1, c1SendConfC2, c2StartTracking || c2AskConfC1,
c2AskConfC4, c4SendConfC2, c3AskConfC1, c2StartTracking ||
c2AskConfC1, c2AskConfC4, c3AskConfC1, c4SendConfC2,
c2StartTracking || c2AskConfC1, c1SendConfC2, c2AskConfC4,
c4SendConfC2, c3AskConfC1, c2StartTracking || c2AskConfC1,
c1SendConfC2, c2AskConfC4, c3AskConfC1, c4SendConfC2,
c2StartTracking || c2AskConfC1, c2AskConfC4, c1SendConfC2,
c4SendConfC2, c3AskConfC1, c2StartTracking || c2AskConfC1,
c2AskConfC4, c4SendConfC2, c1SendConfC2, c3AskConfC1,
c2StartTracking || c2AskConfC1, c2AskConfC4, c1SendConfC2,
c3AskConfC1, c4SendConfC2, c2StartTracking || c2AskConfC1,
c2AskConfC4, c4SendConfC2, c3AskConfC1, c1SendConfC2,
c2StartTracking || c2AskConfC1, c2AskConfC4,  c3AskConfC1,
c4SendConfC2, c1SendConfC2, c2StartTracking || c2AskConfC1,
c2AskConfC4,  c3AskConfC1,  c1SendConfC2, c4SendConfC2,
c2StartTracking || c2AskConfC1, c2AskConfC4, c3AskConfC1,
c1SendConfC3, c2StartTracking || c2AskConfC1, c1SendConfC2,
c2AskConfC4, c3AskConfC1, c1SendConfC3, c2StartTracking ||
c2AskConfC1, c2AskConfC4, c1SendConfC2, c3AskConfC1,
c1SendConfC3, c2StartTracking || c2AskConfC1, c2AskConfC4,
c3AskConfC1, c1SendConfC2, c1SendConfC3, c2StartTracking ||
c2AskConfC1, c2AskConfC4, c3AskConfC1, c1SendConfC3,
c1SendConfC2, c2StartTracking || c2AskConfC1, c2AskConfC4,
c4SendConfC2, c3AskConfC1, c1SendConfC3, c2StartTracking ||
c2AskConfC1, c2AskConfC4, c3AskConfC1, c4SendConfC2,
c1SendConfC3, c2StartTracking || c2AskConfC1, c2AskConfC4,
c3AskConfC1, c1SendConfC3, c4SendConfC2, c2StartTracking
```

O propósito 3 identifica todas as possibilidades para o IS3, identificando assim sua família de cenários implícitos. A formação de todas as possibilidades para essa condição de corrida cresce rapidamente, contudo, isso é possível devido a quantidade limitada de mensagens representadas nos cenários implícitos identificados pelo LTSA-MSC.

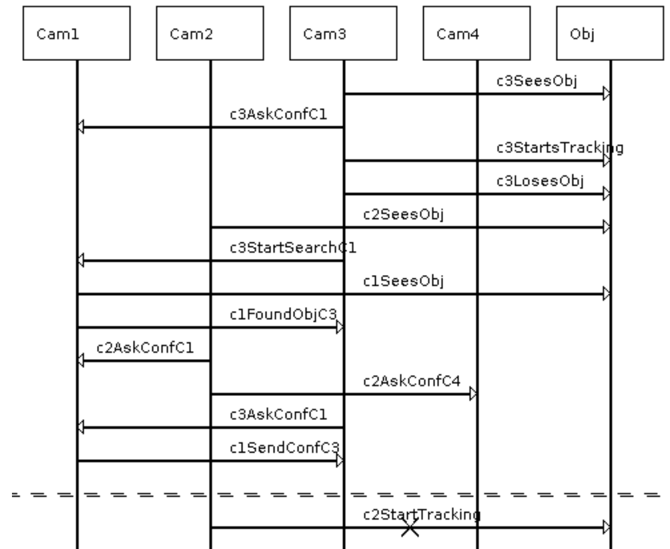


Figura 4.9: Quarto cenário implícito negativo encontrado para a arquitetura 1 (IS4)

Analisando o IS4 temos que a câmera 3 possui o objeto em seu campo de visão e envia uma mensagem *"Sees"* ao sistema, e envia também uma mensagem *"SendConfidence"* a sua vizinha, a câmera 1, a fim de se obter seu índice de confiança. Então, começa a monitorar o objeto, enviando uma mensagem *StartsTraking* ao sistema.

Quando a câmera 3 está prestes a perder o objeto do seu campo de visão ele envia uma mensagem *"Loses"* ao sistema, indicando que outra câmera deverá monitorá-lo. O objeto entra no campo de visão da câmera 2, que envia uma mensagem *"Sees"* ao sistema. Em seguida a câmera 3 envia uma mensagem *"StartSearch"* à câmera 1 para saber se a mesma possui o objeto em seu campo de visão. A câmera 1 então avista o objeto e envia uma mensagem *"Sees"* ao sistema e uma mensagem *"FoundObj"* para a câmera 3 respondendo sua solicitação.

A câmera 2 envia mensagens *"AskConfidence"* às sua vizinhas, as câmeras 1 e 4, a fim de se obter os seus respectivos índices de confiança. A câmera 3 envia também uma mensagem *"Ask Confidence"* à câmera 1 requisitando seu índice de confiança, que responde em seguida com a mensagem *"SendConfidence"* com seu índice. Então a câmera 2 começa a monitorar o objeto e envia uma mensagem *StarsTraking* ao sistema.

Assim como nos cenários IS1, IS2 e IS3 a natureza de formação do cenário IS4 está relacionada à presença de condições de corrida. Então, para a formação do propósito de

teste temos que identificar o conjunto de mensagens responsáveis pela condição de corrida no IS4, apresentado na Figura 4.9.

A condição de corrida, no IS4, se dá devido à câmera 2 começar a rastrear o objeto, por meio da mensagem *"c2StarsTracking"*, antes da resposta, *"SendConf"* das câmeras adjacentes, C1 e C4. As mensagens responsáveis pela condição de corrida são:

c2AskConfC1, c2AskConfC4, c3AskConfC1, c1SendConfC3, c2StarsTracking

Então para a formação desse propósito de teste deve-se descrever todas as possibilidades de composição das mensagens responsáveis pela condição de corrida, retirando-se os casos em que ele não existe, como por exemplo o caso em que as mensagens *"SendConf"* chegam antes da mensagem *"c2StarsTracking"*. Contudo nota-se que o conjunto de mensagens retratado, para o IS4, é um caso particular do propósito de teste 3. Logo podemos identificar o IS4 como sendo pertencente à família de cenários implícitos do IS3, não sendo necessário criar um novo propósito de teste, pois o mesmo já é identificado pelo propósito de teste 3.

Embora todos os quatro cenários implícitos encontrados para a arquitetura 1 (IS1, IS2, IS3 e IS4) possam gerar um mesmo tipo de falha, ficou evidente a presença de três famílias de cenários implícitos, pois existem três conjuntos diferentes de mensagens trocadas, que geram diferentes cenários implícitos. A primeira família é representada pelo IS1, a segunda pelo IS2 e a terceira família é representada pelos cenários implícitos pelo IS3 e IS4.

Arquitetura 2

Para a segunda arquitetura, representada na Figura 4.2, foram encontrados 52 cenários implícitos, sendo somente um deles classificado como um cenário implícito negativo. As Figuras 4.10 e 4.11 apresentam um exemplo dos cenários implícitos positivos encontrados e o cenário implícito negativo encontrado, respectivamente.

Os cenários implícitos positivos encontrados na arquitetura 2 não produzem nenhum efeito prejudicial à arquitetura, eles somente são indicados como cenários implícitos por possuírem uma sequência de mensagens que não foram especificadas em seu modelo, podendo serem integradas à arquitetura sem nenhum prejuízo.

A Figura 4.10 é um exemplo de um cenário implícito positivo. Neste exemplo, a câmera 3 avista o objeto (*"Sees"*), pede a câmera 1, sua vizinha, seu índice de confiança (*"AskConfidence"*) e em seguida começa a rastrear o objeto (*"StarsTraking"*). Em seguida a câmera 3 irá perder o objeto de seu campo de visão, e envia então uma mensagem *"Loses"*

Figura 4.10: Cenário implícito positivo encontrado para a arquitetura 2 (ISP1)

para informar o sistema. A câmera 1 visualiza o objeto e informa ao sistema, por meio da mensagem *"Sees"*.

A câmera 3 continua seu protocolo e envia uma mensagem *StartSearch* à câmera 1 para saber se ela possui o objeto em seu campo de visão, e ela responde à solicitação com uma mensagem *"Found"*, indicando que possui a visualização do objeto. A câmera 3 pede então o seu índice de confiança (*"AskConfidence"*), sendo respondido com uma mensagem *"SendConfidence"*.

A câmera 3 passa então a responsabilidade de rastrear o objeto a câmera 1 (*"Allows-Track"*) que informa ao sistema, por meio de uma mensagem *"StartTracking"*. Então o objeto começa a sair do campo de visão da câmera 1, que envia uma mensagem *"Lose"* ao sistema para informá-lo e em seguida envia, também, uma mensagem *"StartSearch"* à câmera 3 para verificar se ela possui o objeto em seu campo de visão.

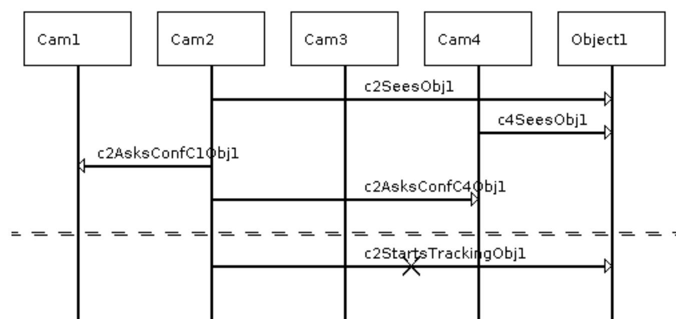


Figura 4.11: Cenário implícito negativo encontrado para a arquitetura 2 (IS5)

O cenário implícito negativo encontrado na arquitetura 2, IS5, apresentado na Figura 4.11, é semelhante ao IS1, da arquitetura 1, no que diz respeito à troca de mensagens. No IS5, as câmeras 2 e 4 possuem o objeto em seu campo de visão e enviam mensagens *"Sees"* ao sistema para notificá-lo que estão aptas a rastreá-lo. Em seguida, a câmera 2 faz uma requisição do índice de confiança das suas vizinhas, as câmeras 1 e 4, por meio de mensagens *"AskConfidence"*. Antes delas responderem à requisição a câmera 2 começa a monitorar o objeto.

Assim como todos os cenários implícitos encontrados para a arquitetura 1, o cenário IS5 também é formado devido à presença de condições de corrida. A câmera 2 não aguarda a chegada das mensagens *"SendConfidence"* de suas vizinhas, com o seu índice de confiança, para decidir qual deve rastrear o objeto. Então, é necessário identificar as mensagens responsáveis pela condição de corrida e compô-las a fim de representar todos os possíveis caminhos. As mensagens são:

c2AskConfC1Obj1, c2AskConfC4Obj1, c2StarsTrackingObj1

A condição de corrida se dá devido à câmera 2 começar a rastrear o objeto, através da mensagem *"c2StarsTrackingObj1"*, antes da resposta, *"SendConf"* da câmeras C1 e C4. Então, para a formação desse propósito de teste deve-se descrever todas as possibilidades de composição dessas mensagens, retirando-se os casos em que não existe a condição de corrida, como por exemplo o caso em que as mensagens *"SendConf"* chegam antes da mensagem *"c2StarsTrackingObj1"*. O propósito de teste, para o cenário implícito IS5 é:

Listing 4.4: Propósito de Teste 4

```
c2AskConfC1Obj1, c2AskConfC4Obj1, c2StarsTrackingObj1 ||
c2AskConfC1Obj1, c1SendConfC2Obj1, c2AskConfC4Obj1,
c2StarsTrackingObj1 || c2AskConfC1Obj1, c2AskConfC4Obj1,
c1SendConfC2Obj1, c2StarsTrackingObj1 || c2AskConfC1Obj1,
c2AskConfC4Obj1, c4SendConfC2Obj1, c2StarsTrackingObj1
```

Em ambas as arquiteturas a presença dos cenários implícitos negativos pode gerar uma diminuição na confiabilidade do sistema. Devido ao sistema não esperar o índice de confiança das câmeras vizinhas, antes de começar a rastrear o objeto, o mesmo pode tomar ações equivocadas levando-o a um estado de falha, como por exemplo uma câmera pode monitorar um objeto que está muito longe, podendo não visualizar o objeto de forma tão precisa quanto outra câmera que possua o objeto em seu campo de visão.

4.6 Geração dos Casos de Teste

Para a geração dos casos de teste faremos uso da metodologia proposta no Capítulo 3. Como já apresentado, todo o processo descrito na metodologia foi automatizado e integrado à ferramenta LTSA-MSC na forma de um *plug-in* chamado *Test Generation*, que por simplificação o chamaremos de *TG*. O *plug-in TG* faz uso dos algoritmos desenvolvidos para gerar o conjunto de casos de teste, apresentados no Capítulo 3.

4.6.1 Arquitetura 1

Para a geração dos casos de teste referentes à arquitetura 1, utilizaremos o propósito de teste 1 para identificar a primeira família de cenários implícitos, representado pelo IS1, o propósito de teste 2 para identificar a segunda família, representado pelo IS2, e o propósito de teste 3 para identificar a terceira família, representada pelos IS3 e IS4.

A Arquitetura 1 possui dois estados finais, um que leva ao estado final de sucesso do sistema, representado pela aresta *END*, e outro que leva a um estado final de falha devido a uma situação de impasse, representado pela aresta *STOP*. O *Plug-in TG* realizará uma busca pelo cenário implícito, desde o estado inicial do modelo comportamental, até cada um dos estado finais. Para o estado final que leva a um impasse será representada uma aresta hipotética levando ao estado final de sucesso do sistema. O modelo comportamental da arquitetura 1 é extraído automaticamente do LTSA-MSC restando apenas inserir o propósito de teste no *TG*.

A partir do propósito de teste e do modelo arquitetural o *plug-in TG*faz uso dos algorítmos descritos na metodologia, Capítulo 3, a fim de se gerar os casos de teste. Como o sistema *Smart Cams* possui dois estado finais, denotados pelas arestas *END* e *STOP*, ele realizará duas chamadas ao método a fim de se obter os menores caminhos que possuam o cenário implícito.

O conjunto de cenários de teste, ou cenários implícitos mínimos (assim chamados, pois são os menores caminhos do grafo que possuem o cenário implícito), obtidos para o sistema *Smart Cams* são apresentados na Tabela 4.2. Esse conjunto de cenários representa também o conjunto de casos de teste gerado para a arquitetura 1, pois a mesma não possui ciclos, possui somente caminhos alternativos entre doi nós do LTS.

Em seguida o *TG* realiza uma chamada ao método *DFSLoop* que recebe como entrada o grafo e retorna como saída uma lista com todos os ciclos que existentes no grafo, esse método retornará uma lista vazia, pois, como já mencionado a Arquitetura 1 não apresenta ciclos em sua representação.

Então o *TG* realiza um merge entre o cenário implícito mínimo e os resultados obtidos pelo *DFSLoop* identificando onde existe ciclos no cenário implícito e adicionando-o, a fim

de compor o resultado. A Tabela 4.2 apresenta os resultados que possuem os casos de teste gerados para a arquitetura 1.

Propósitos de Teste - PT	Cenários Implícitos Identificados
PT1 - Família 1	0, 44, [68, 46, 47, 62], 63, 64, 65, 66. STOP, 16.
PT2 - Família 2	0, 1, 2, 3, 4, 22, 6, 7, [8, 9, 20, 21], 19, 17. STOP, 16.
PT3 - Família 3 - IS3	0, 1, 2, 3, 4, 22, 6, 7, [8, 9, 20, 18, 19], 17. STOP, 16.
PT3 - Família 3 - IS4	0, 1, 2, 3, 4, 22, 6, 7, [8, 9, 20, 18, 12, 17]. STOP, 16.

Tabela 4.2: Cenários Implícitos identificados para as famílias da Arquitetura 1

4.6.2 Arquitetura 2

A arquitetura 2 possui uma família de cenários implícitos representada pelo IS5. Logo utilizaremos o propósito de teste 4 para a identificar sua família de cenários implícitos. Assim como feito para a arquitetura 1, a *TG* obterá o FSP da arquitetura 2, apresentado no anexo B, automaticamente do LTSA-MSC. Para o propósito de teste 4 como entrada, teremos o cenário de teste apresentado na Tabela 4.3, que também é o caso de teste gerado para a Arquitetura 2.

Posteriormente, o *TG* identifica os laços presentes na Arquitetura 2, retornado então uma lista vazia, pois assim como a Arquitetura 1, a Arquitetura 2 não possui ciclos, possui somente caminhos alternativos entre dois nós do grafo. Em seguida o *plug-in* realiza um merge entre os dois resultados, produzindo os casos de teste, apresentados na Tabela 4.3, para o propósito de teste 4.

Propósitos de Teste - PT	Cenário Implícito identificado
PT4 - Família Única	0, 1, [25, 14, 15, 16], 17, 18. STOP, 13.

Tabela 4.3: Cenário Implícito identificado para a Arquitetura 2.

4.7 Resultados e Conclusões

Para a Arquitetura 1, temos presentes três famílias de cenários implícitos, a primeira representado pelo IS1, a segunda pelo IS2 e a terceira pelos IS3 e IS4. Essas famílias geram o mesmo tipo de falha. Contudo, os caminhos responsáveis por sua representação são diferentes, levando assim a existência dessas três diferentes famílias de cenários implícitos.

Para a Arquitetura 2, existe somente uma família de cenários implícitos que gera o mesmo tipo de falha ocasionada pela Arquitetura 1. Devido a ambas as arquiteturas

apresentarem falhas do tipo inconsistente e terem a mesma natureza de formação, condição de corrida, elas então produzem o mesmo tipo de falha, como apresentado em [45].

Para identificar a primeira família de cenários implícitos, representado pelo IS1, foi utilizado o propósito de teste 1, criado a partir das mensagens que configuram a condição de corrida presente no IS1. De forma semelhante ocorre com os cenários IS2 para a arquitetura 1, e IS5, para a arquitetura 2. Suas famílias de cenários implícitos podem ser identificadas pelo seu conjunto de mensagens que são responsáveis pela condição de corrida identificada em seu respectivos cenários implícitos.

O IS3 e o IS4 pertencem a uma mesma família de cenários implícitos e podem ser identificados pelo propósito de teste 3, pois o cenário IS4 é umas das possíveis possibilidades da formação da condição de corrida apresentada pelo IS3. Embora o propósito de teste 3 seja maior, em comparação aos demais descritos, ele avalia todas as possibilidades da condição de corrida presentes no IS3, o que seria benéfico em outras situações, como por exemplo uma situação em que a família de cenários implícitos seja infinita, devido à presença de ciclos em seu modelo comportamental. Nesse caso a metodologia identificaria toda a família de cenários implícitos e representaria, utilizando a heurística apresentada na metodologia, todos os ciclo existentes.

Este estudo de caso teve como objetivo validar a metodologia a partir das hipóteses apresentadas: ***(Hipótese I)*** Em um cenário real com características distintas da condição de corrida trivial, como exemplificado no sistema de caldeira, a metodologia apresentada se demosntra eficaz e completa, no sentido de cobrir de fato a identificação de todas as famílias de cenários implícitos negativos presentes na arquitetura, e ***(Hipótese II)*** Com a geração dos casos de teste é possível avaliar os caminhos atingidos pelos cenários implícitos tendo assim uma percepção mais real da natureza das falhas que podem ser geradas pela presença de cada família de cenários implícitos, podendo criar assim subterfúgios para sua correção.

Os resultados consolidados são apresentados a seguir:

Hipótese I: A partir dos casos de teste gerados pelo *Plug-in Test Generation* para o sistema *Smart Cams*, apresentados na Tabela 4.2 para a arquitetura 1 e na Tabela 4.3 para a arquitetura 2, foi realizada uma comparação com os cenários implícitos identificados nos trabalhos [44] e [45]. Foi constatado que todos os cenários implícitos catalogados nos trabalhos são identificados pelos casos de teste gerados pelo *Plug-in Test Generation*. E a análise manual do modelo arquitetural do sistema, para as duas arquiteturas, não apresentou outros cenários implícitos além dos que foram identificados pelo conjunto de casos de teste retornado.

Hipótese II: No sistema *Smat Cams* os defeitos gerados pelos cenários implícitos afetam o sistema de forma relevante. Com a avalição dos casos de teste gerados temos

uma melhor percepção da natureza de formação das falhas geradas, ou seja, somos capazes de identificar o motivo da falha e quais caminhos levam a aquele tipo de falha, pois teremos representados todos os caminhos em que o sistema testado leva a um tipo específico de falha gerada pela presença daquela família de cenários implícitos.

4.8 Ameaça à Validade

Diante do apresentado nas seções anteriores, destaca-se uma ameaça que poderia dificultar a obtenção dos resultados.

- O crescimento do tamanho do propósito de teste é um fator limitante à metodologia? Acreditamos que sim, pois embora o conjunto de mensagens retornadas pelo LTSA-MSC seja limitado, nos casos utilizados nesse trabalho, não existem estudos que comprovem um tamnho máximo dos cenários implícitos retornados pelo LTSA-MSC, podendo sim ser um fator limitante quanto à complexidade em se gerar o propósito de teste. No entanto, para o estudo de caso realizado, a abordagem se mostrou viável no contexto do SmartCams. Para que a nossa abordagem não seja apenas viável, mas também eficiente, pretendemos integrar o uso de técnicas de lógica temporal para diminuir o tamanho do propósito de teste.

Capítulo 5

Conclusão e Trabalhos Futuros

5.1 Conclusão

Este trabalho teve como objetivo apresentar uma nova metodologia para a geração de casos de teste, baseados em modelo, para a identificação de uma anomalia comum em sistemas concorrentes, os cenários implícitos. Para a criação da metodologia foram desenvolvidos algoritmos, a fim de se realizar uma busca por todos os caminhos que possuam os cenários implícitos, em um grafo.

Para a realização do estudo de caso foi implementado um *plug-in* para a ferramenta LTSA-MSC, chamado *Test Generation*, que faz uso da metodologia desenvolvida. Utilizamos então um sistema de câmeras inteligentes, *Smart Cams*, que também foi amplamente utilizado na literatura, e que possui todos os cenários implícitos já identificados, a fim de gerar um conjunto de casos de teste que os represente de forma confiável, validando assim nossa metodologia.

Os resultados obtidos vieram a validar a metodologia proposta salientando assim a importância da geração dos casos de teste que identifiquem as famílias de cenários implícitos em sistemas concorrentes. Com essa metodologia, temos então, um subsídio para se alcançar a confiabilidade estimada ao sistema.

5.2 Trabalhos Futuros

Uma limitação desse trabalho é um problema clássico da literatura onde, em virtude da condição de corrida, a quantidade de combinações entre as mensagens pode crescer muito rapidamente, com o aumento de troca das mensagens entre os componentes. Isso pode ser um fator limitante devido à complexidade em se gerar o propósito de teste.

Em [48] é estendido o FLTL [47] com versões limitadas dos operadores de lógica temporal para modelar propriedades de tempo em modelos baseados em eventos de tempo discreto.

Uma possibilidade de trabalho futuro seria diminuir o tamanho do propósito de teste. E uma possível solução para diminuir o seu tamanho é usar os operadores de lógica temporal, definidos em [48], para identificar o conjunto de *traces*, dentro do modelo arquitetural do sistema, que ocorre o cenário implícito, diminuindo assim o tamanho do propósito de teste para casos de iterações mais complexas entre componentes.

Em [53] é apresentado uma abordagem que faz uso de cálculos probabilísticos a fim de se desenvolver uma ferramenta de suporte à análise de confiabilidade de forma composicional. Outra possibilidade de trabalho futuro seria a integração de nossa metodologia ao LTSA-PCA.

Essa integração agregaria mais valor a metodologia e trazendo como resultado a probabilidade de ocorrência dos caminhos que possuem o cenário implícito. Com essa possibilidade o projetista poderá avaliar, a partir do custo da correção do cenário implícito e de sua probabilidade de ocorrência, se é viável sua correção.

Referências Bibliográficas

[1] Tretmans, Jan *Model Based Testing with Labelled Transition Systems* Formal Methods and Testing. pp. 1-38, Springer-Verlag. 2008. 1, 3, 14, 15

[2] Michael R. Lyu. *Software Reliability Engineering: A Roadmap.* In 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 153-170. 1

[3] S. Uchitel, J. Kramer, and J. Magee, *Incremental elaboration of scenario-based specifications and behavior models using implied scenarios.* ACM Trans. Softw. Eng. Methodol., 13(1):37–85.(2004).

[4] S. Uchitel, J. Kramer, and J. Magee, *Detecting Implied Scenarios in Message Sequence Chart Specifications,* In ACM Proceedings of the Joint 8th ESEC and 9th FSE", pp. 74-82, 2001. x, 2, 7, 10, 11, 12, 26, 28

[5] Vaz Roriz, A. and Nunes Rodrigues, G. and Laranjeira, L.A. *Analysis of the Impact of Implied Scenarios on the Reliability of Computational Concurrent Systems,* In Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium. pages 105-114. 2, 3, 21

[6] A. Avizienis, J. Claude Laprie, B. Randell, and C. Landwehr, *Basic Concepts and Taxonomy of Dependable and Secure Computing,* IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January - March 2004. 2, 8, 9, 13

[7] L. Hoffman, *Search of Dependable Design,* In ACM, vol. 51, no. 7, pp. 14-16, 2008. 9

[8] Daohua Wu, Schnieder, E. and Krause, J. *Model-based test generation techniques verifying the on-board module of a satellite-based train control system model.* In Intelligent Rail Transportation (ICIRT), 2013 IEEE International Conference, Page(s): 274 - 279. Beijing. 2013.

[9] Jeff Offutt and Aynur Abdurazik. *Generating Tests from UML Specifications.* In Second International Conference on the Unified Modeling Language (UML'99), pages 416-429, Fort Collins, CO, October 1999.

[10] Genaína Nunes Rodrigues and Vander Alves and Renato Silveira and Luiz A. Laranjeira, *Dependability analysis in the Ambient Assisted Living Domain: An exploratory case study,* Journal of Systems and Software. pp. 112-131, Vol. 85. 2012. 9

[11] Felipe Cantal de Sousa and Nabor C. Mendonça and Sebastián Uchitel and Jeff Kramer, *Detecting Implied Scenarios from Execution Traces,* In 14th Working Conference on Reverse Engineering (WCRE), October 2007, Vancouver, BC, Canada. Pages 50-59. 6, 9

[12] S. Uchitel, J. Kramer, and J. Magee, *Synthesis of Behavioral Models from Scenarios,* IEEE Trans. on Software Engineering, pp. 99-115, 2003. 2, 9

[13] Sidney Nogueira and Augusto Sampaio and Alexandre Mota, *Guided Test Generation from CSP Models,* In Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, 2008. Pages 258-273. x, 6, 9, 16, 17

[14] J. Magee, and J. Kramer, *Concurrency: State Models and Java Programs,* John Wiley & Sons Ltd., New York, 1999. 9

[15] Alur, Rajeev and Etessami, Kousha and Yannakakis, Mihalis, *Inference of Message Sequence Charts,* IEEE Trans. Software Eng. ICSE, Limerick, Ireland, 2000. Pages 623-633. 10

[16] S. Uchitel, J. Kramer, and J. Magee, *Negative Scenarios for Implied Scenario Elicitation,* In Proc. 10th ACM SIGSOFT Symp. on the Found. of Software Engineering FSE, Charleston, SC, USA, 2002. 10

[17] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, *LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios,* In Proc. 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems TACAS, LNCS Vol. 2619, Springer, 2003. 3, 10

[18] S. Uchitel, J. Kramer, and J. Magee, *Incremental Elaboration of Scenariobased Specifications and Behaviour Models Using Implied Scenarios,* ACM Trans. Softw. Eng. Methodol. 2004, Pages 37-85. x, 4, 5, 6, 10, 13, 28

[19] Myers, G. J., *The Art of Software Testing,* John Wiley & Sons, New York, 2004. 1, 7, 10, 13

[20] Pressman, R. S., *Software Engineering: A Practitioner's Approach,* McGraw-Hill, Boston, 2005. 14

[21] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement,* Wiley-IEEE Computer Society Press, Hoboken, 2005. 14

[22] A. M. R. Vincenzi, et al. *Introdução ao Teste de Software,* Notas Didáticas do ICMC-USP, São Paulo, 2004. 14

[23] C. D. Yang, and L. L. Pollock, *The challenges in automated testing of multithreaded programs,* In Proceedings of the 14th International Conference on Testing Computer Softwar, pp. 157-166, 1997. 14

[24] A. Bertolino, E. Marchetti, and H. Muccini, *Introducing a Reasonably Complete and Coherent Approach for Model-based Testing,* Journal Electronic Notes in Theoretical Computer Science Volume 116, pp. 85-97, 2005. 4

[25] S. Al-Azzani, S. and R. Bahsoon, *Using Implied Scenarios In Security Testing,* In SESS 10 Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, pp. 15-21, 2010. 5

[26] Thomas H. Cormen, Charles E. Leiserson, Ronaldo L. Rivest, and Clifford Stein, *Introduction to Algorithms,* Third Edition, MIT Press, 2009. 4, 23, 24, 26

[27] G. Nunes Rodrigues, S. Uchitel, and D. Rosenblum, *Using scenarios to predict the reliability of concurrent component-based software systems,* In FASE'05, pp. 111-126. 7

[28] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools,* In AddisonWesley, 2000.

[29] Zoltán Micskei, *Model-based testing - MBT,* Department of Measurement and Information Systems Budapest University of Technology and Economics `http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html`", Accessed: 2014-12-08, 2014. 14

[30] Mark Utting, Alexander Pretschner, and Bruno Legeard, *A Taxonomy of Model-based Testing Approaches,* Softw. Test. Verif. Reliab., Vol 22, nº 5, pp. 297-312, August 2012. 3, 15

[31] ISO/IEC JTC1/SC21 WG7 and ITU-T SG 10/Q.8 *Information retrieval, transfer and management for osi; framework: Formal methods in conformance testing* ITU-T proposed recommendation Z.500 CD 13245-1, ISO - ITU-T, Geneve, 1996. Committee Draft. 16

[32] Jan Tretmans *Conformance testing with labelled transition systems: Implementation relations and test generation* Computer Networks and ISDN Systems, 29(1):49–79, 1996. 16, 17

[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, 2005. 16

[34] J.Baeten and W. Weijland *Process Algebra* volume 18 of of Cambridge tracts in theoretical computer science. Cambridge University Press, 1990. 16

[35] R De Nicola *Extensional equivalence for transition systems* Acta Inf., 24(2):211–237, 1987. 16

[36] E. Brinskma *A theory for the derivation of tests* Protocol Specification, Testing and Verification, VIII:63–74, 1988. 16

[37] J. Tretmans *Test Generation with Inputs, Outputs and Repetitive Quiescence* Software—Concepts and Tools, 17(3):103–120, 1996. 16

[38] Letier, E. and Kramer, J. and Magee, J. and Uchitel, S. *Monitoring and Control in Scenario-Based Requirements Analysis,* In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. Pages 382-391. 12

[39] Hanêne Ben-Abdallah and Stefan Leue, *MESA: Support for Scenario-Based Design of Concurrent Systems,* 1997. 12

[40] J. Whittle, and J. Schumann, *Generating Statechart Designs from Scenarios,* In Proceedings of the 22nd International Conference on Software Engineering, ICSE - Limerick, Ireland, 2000. 12

[41] Robert Chatley, Jeff Kramer, Jeff Magee, and Sebastian Uchitel, *Model-based Simulation of Web Applications for Usability Assessment,* In ICSE Workshop on Bridging the Gaps Between Software Engineering and Humam-Computer Interaction, 2003. 12

[42] Leue, Stefan and Mehrmann, Lars and Rezai, Mohammad *Synthesizing ROOM Models from Message Sequence Chart Specifications,* In Proc. of 13th IEEE Conf. on Automated Software Engineering. 1998. 12, 13

[43] Heitmeyer, Constance L. and Jeffords, Ralph D. and Labaw, Bruce G. *Automated Consistency Checking of Requirements Specifications,* ACM Trans. Softw. Eng. Methodol. Vol 5, n. 3, pp. 231-261. July 1996. 13

[44] S. Al-Azzani, *"Architecture-Centric Testing for Security"* PhD thesis, University of Birmingham, 2013. x, 36, 37, 38, 39, 40, 54

[45] A. Vaz Roriz, *"Análise do Impacto de Cenários Implícitos na Confiabilidade de Sistemas Computacionais"* Dissertação apresentada ao curso de Mestrado em Informática da Universidade de Brasília, 2014. 36, 37, 38, 39, 54

[46] S. D. BROOKES, C. A. R. HOARE, and A. W. ROSCOE, *"A Theory of Communicating Sequential Processes "* J. ACM, Vol. 31, No 3, pp.560-599, July 1984. 6

[47] D. Giannakopoulou and J. Magee, *"Fluent Model Checking for Event-Based Systems"* In Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE-11. Helsinki, Finland. Pages 257-266. 57

[48] Letier, E. Kramer, J. Magee, J. Uchitel, S., *"Fluent temporal logic for discrete-time event-based models"* In Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software. 2005. 57

[49] Tanenbaum, Andrew S. *"Sistemas Operacionais Modernos,"* 3ª Ed. 2010. 21, 30

[50] Esterle, L., Lewis, P. R., *et al.*, *A socio-economic approach to online vision graph generation and handover in distributed smart camera networks.* Proceedings of the Fifth ACM/IEEE International Conference on Distributed Smart Cameras, pages 1–6, 2011. 4, 35, 37

[51] Esterle, L., Lewis, P. R., *et al.*, *Socio-Economic Vision Graph Generation and Handover in Distributed Smart Camera Networks.* In Proceedings of the ACM Transactions on Sensor Networks, vol. 10, no. 2, pp. 20:1–20:24, Jan. 2014. 4, 35, 36

[52] Baier, C., Kaoten, JP. *Principles of Model Checking.* The MIT Press, Cambridge, Massachusetts (2008). 20

[53] Rodrigues, G. N., Rosenblum, D. and Uchitel, S. *Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems.* In Proc. ETAPS 2005 Conference on Formal Approaches to Software Engineering, pages 111-126. Springer, LNCS 3442, 2005. 57

[54] D. E. Perry, A. A. Porter, and L. G. Votta, *Empirical Studies of Software Engineering: A Roadmap,* in Proceedings of the Conference on The Future of Software Engineering, New York, NY, USA, 2000, pp. 345–355. 34

Anexo A

Diagramas bMSCs

A.1 Diagramas bMSCs da Arquitetura 1, para o sistema *Smart Cams*

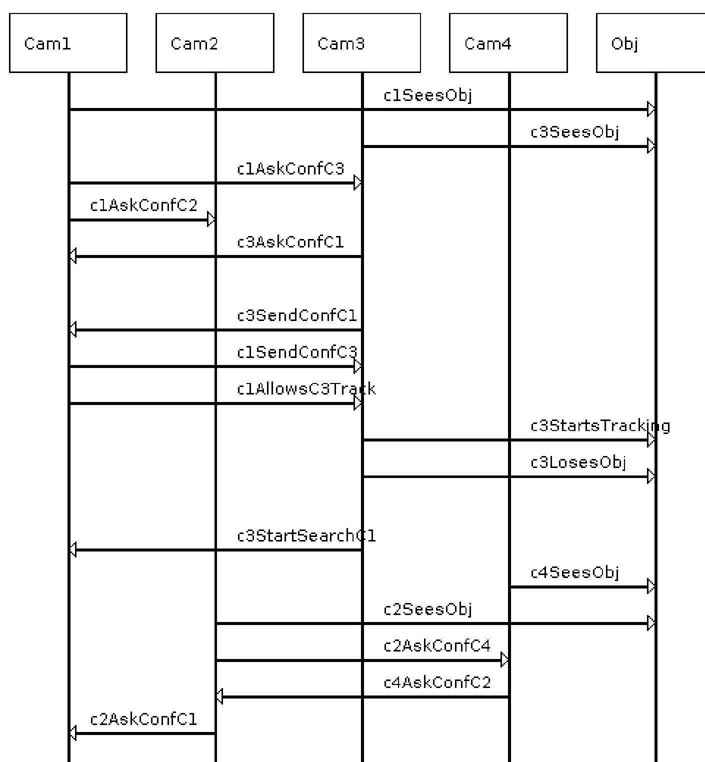


Figura A.1: Diagrama bMSC referente ao cenário 1 da Arquitetura 1

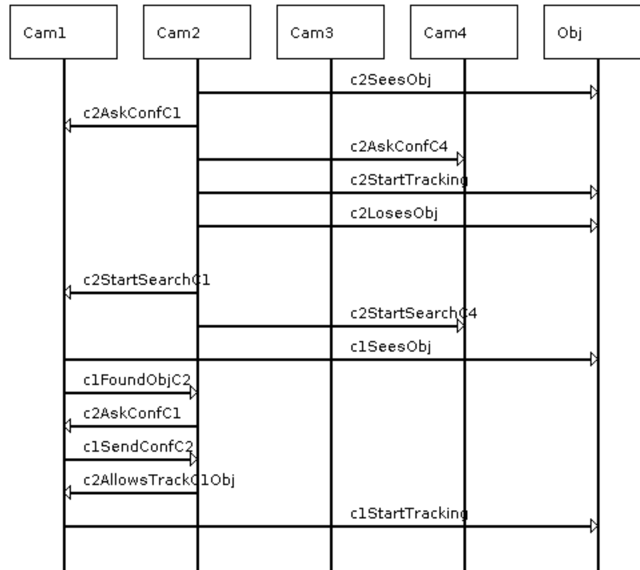


Figura A.2: Diagrama bMSC referente ao cenário 2 da Arquitetura 1

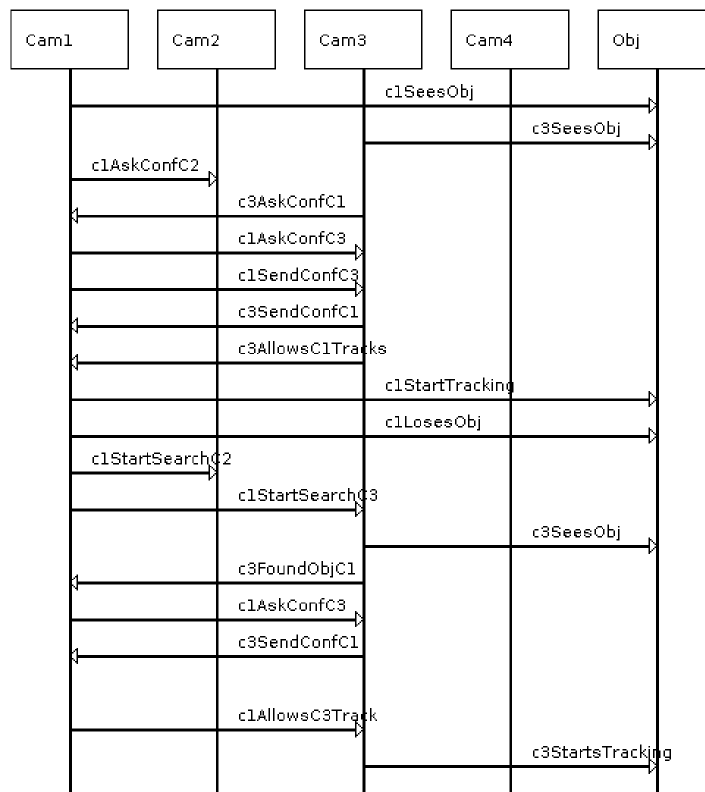


Figura A.3: Diagrama bMSC referente ao cenário 3 da Arquitetura 1

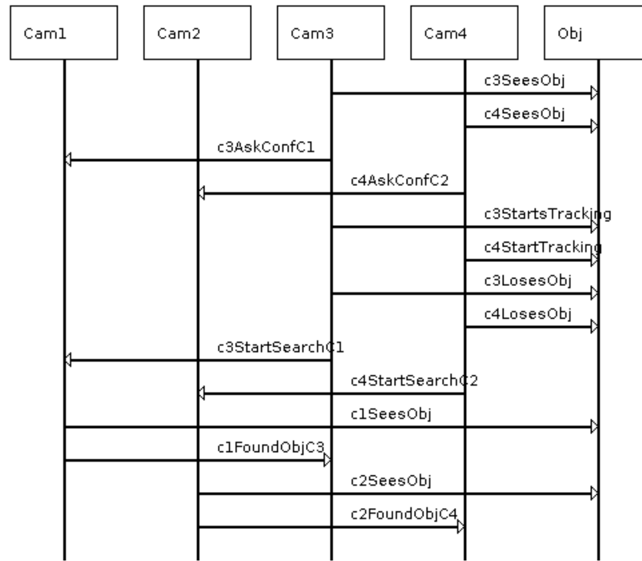


Figura A.4: Diagrama bMSC referente ao cenário 4 da Arquitetura 1

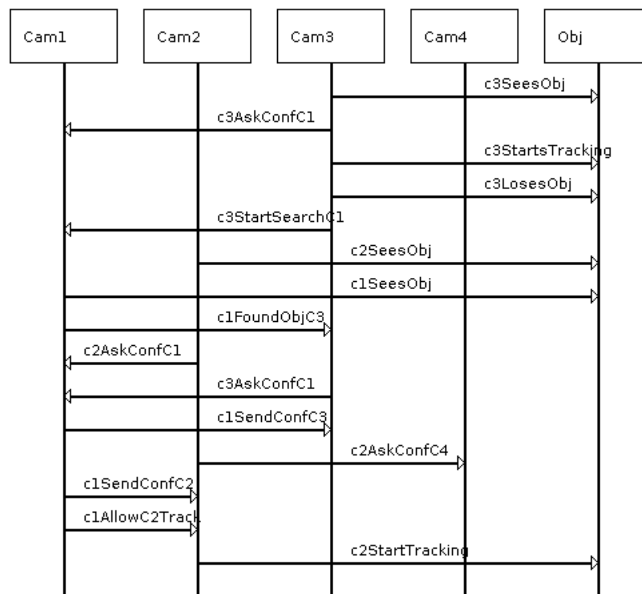


Figura A.5: Diagrama bMSC referente ao cenário 5 da Arquitetura 1

A.2 Diagramas bMSCs da Arquitetura 2, para o sistema *Smart Cams*

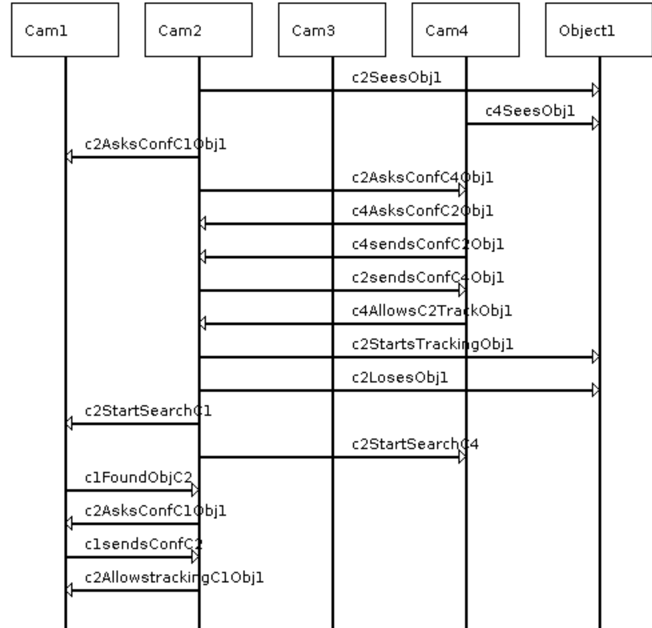


Figura A.6: Diagrama bMSC referente ao cenário 1 da Arquitetura 2

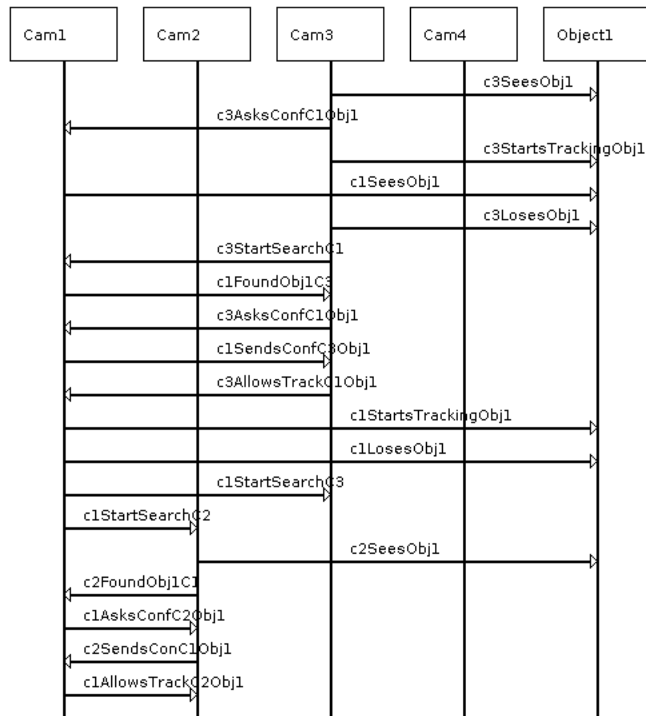


Figura A.7: Diagrama bMSC referente ao cenário 2 da Arquitetura 2

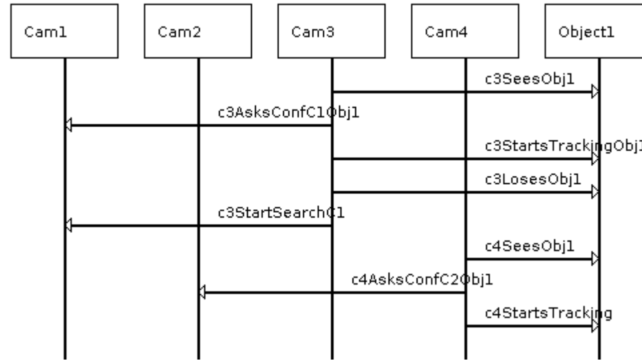


Figura A.8: Diagrama bMSC referente ao cenário 3 da Arquitetura 2

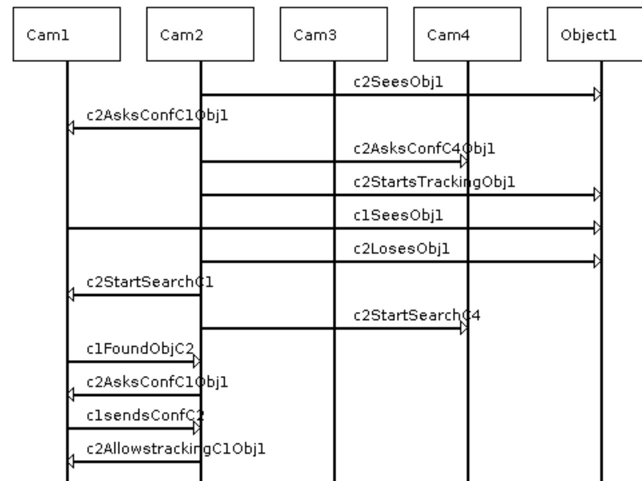


Figura A.9: Diagrama bMSC referente ao cenário 4 da Arquitetura 2

Figura A.10: Diagrama bMSC referente ao cenário 5 da Arquitetura 2

Anexo B

Modelo Comportamental, Descrito em FSP

B.1 Modelo comportamental da Arquitetura 1, para o sistema *Smart Cams*, expresso em FSP

Listing B.1: Modelo comportamental da Arquitetura expresso em FSP

```
        Process:
        ArchitectureModel
States:
        101
Transitions:
        ArchitectureModel = Q0,
        Q0      = (cam3.obj.c3SeesObj -> Q1
                  |cam1.obj.c1SeesObj -> Q44
                  |cam2.obj.c2SeesObj -> Q87),
        Q1      = (cam3.cam1.c3AskConfC1 -> Q2
                  |cam4.obj.c4SeesObj -> Q42),
        Q2      = (cam3.obj.c3StartsTracking -> Q3
                  |cam4.obj.c4SeesObj -> Q23),
        Q3      = (cam3.obj.c3LosesObj -> Q4),
        Q4      = (cam3.cam1.c3StartSearchC1 -> Q5
                  |cam2.obj.c2SeesObj -> Q22),
        Q5      = (cam2.obj.c2SeesObj -> Q6),
        Q6      = (cam1.obj.c1SeesObj -> Q7),
```

```
Q7      = (cam1.cam3.c1FoundObjC3 -> Q8),
Q8      = (cam2.cam1.c2AskConfC1 -> Q9),
Q9      = (cam3.cam1.c3AskConfC1 -> Q10
          |cam2.cam4.c2AskConfC4 -> Q20),
Q10     = (cam1.cam3.c1SendConfC3 -> Q11
          |cam2.cam4.c2AskConfC4 -> Q18),
Q11     = (cam2.cam4.c2AskConfC4 -> Q12),
Q12     = (cam1.cam2.c1SendConfC2 -> Q13
          |cam2.obj.c2StartTracking -> Q17),
Q13     = (cam1.cam2.c1AllowC2Track -> Q14),
Q14     = (cam2.obj.c2StartTracking -> Q15),
Q15     = (endAction -> Q16),
Q16     = END,
Q17     = STOP,
Q18     = (cam1.cam3.c1SendConfC3 -> Q12
          |cam2.obj.c2StartTracking -> Q19),
Q19     = (cam1.cam3.c1SendConfC3 -> Q17),
Q20     = (cam3.cam1.c3AskConfC1 -> Q18
          |cam2.obj.c2StartTracking -> Q21),
Q21     = (cam3.cam1.c3AskConfC1 -> Q19),
Q22     = (cam3.cam1.c3StartSearchC1 -> Q6),
Q23     = (cam4.cam2.c4AskConfC2 -> Q24
          |cam3.obj.c3StartsTracking -> Q41),
Q24     = (cam3.obj.c3StartsTracking -> Q25),
Q25     = (cam4.obj.c4StartTracking -> Q26),
Q26     = (cam3.obj.c3LosesObj -> Q27),
Q27     = (cam3.cam1.c3StartSearchC1 -> Q28
          |cam4.obj.c4LosesObj -> Q39),
Q28     = (cam4.obj.c4LosesObj -> Q29),
Q29     = (cam4.cam2.c4StartSearchC2 -> Q30
          |cam1.obj.c1SeesObj -> Q37),
Q30     = (cam1.obj.c1SeesObj -> Q31),
Q31     = (cam1.cam3.c1FoundObjC3 -> Q32
          |cam2.obj.c2SeesObj -> Q35),
Q32     = (cam2.obj.c2SeesObj -> Q33),
Q33     = (cam2.cam4.c2FoundObjC4 -> Q34),
Q34     = (endAction -> Q16),
```

```
Q35      = (cam1.cam3.c1FoundObjC3 -> Q33
          |cam2.cam4.c2FoundObjC4 -> Q36),
Q36      = (cam1.cam3.c1FoundObjC3 -> Q34),
Q37      = (cam4.cam2.c4StartSearchC2 -> Q31
          |cam1.cam3.c1FoundObjC3 -> Q38),
Q38      = (cam4.cam2.c4StartSearchC2 -> Q32),
Q39      = (cam3.cam1.c3StartSearchC1 -> Q29
          |cam4.cam2.c4StartSearchC2 -> Q40),
Q40      = (cam3.cam1.c3StartSearchC1 -> Q30),
Q41      = (cam4.cam2.c4AskConfC2 -> Q25),
Q42      = (cam3.cam1.c3AskConfC1 -> Q23
          |cam4.cam2.c4AskConfC2 -> Q43),
Q43      = (cam3.cam1.c3AskConfC1 -> Q24),
Q44      = (cam1.cam2.c1AskConfC2 -> Q45
          |cam3.obj.c3SeesObj -> Q68),
Q45      = (cam3.obj.c3SeesObj -> Q46),
Q46      = (cam3.cam1.c3AskConfC1 -> Q47),
Q47      = (cam1.cam3.c1AskConfC3 -> Q48
          |cam3.obj.c3StartsTracking -> Q62),
Q48      = (cam1.cam3.c1SendConfC3 -> Q49),
Q49      = (cam3.cam1.c3SendConfC1 -> Q50),
Q50      = (cam3.cam1.c3AllowsC1Tracks -> Q51),
Q51      = (cam1.obj.c1StartTracking -> Q52),
Q52      = (cam1.obj.c1LosesObj -> Q53),
Q53      = (cam1.cam2.c1StartSearchC2 -> Q54),
Q54      = (cam1.cam3.c1StartSearchC3 -> Q55),
Q55      = (cam3.obj.c3SeesObj -> Q56),
Q56      = (cam3.cam1.c3FoundObjC1 -> Q57),
Q57      = (cam1.cam3.c1AskConfC3 -> Q58),
Q58      = (cam3.cam1.c3SendConfC1 -> Q59),
Q59      = (cam1.cam3.c1AllowsC3Track -> Q60),
Q60      = (cam3.obj.c3StartsTracking -> Q61),
Q61      = (endAction -> Q16),
Q62      = (cam3.obj.c3LosesObj -> Q63),
Q63      = (cam4.obj.c4SeesObj -> Q64),
Q64      = (cam2.obj.c2SeesObj -> Q65),
Q65      = (cam2.cam4.c2AskConfC4 -> Q66),
```

```
Q66      = (cam4.cam2.c4AskConfC2 -> Q67),
Q67      = STOP,
Q68      = (cam1.cam2.c1AskConfC2 -> Q46
           |cam1.cam3.c1AskConfC3 -> Q69),
Q69      = (cam1.cam2.c1AskConfC2 -> Q70),
Q70      = (cam3.cam1.c3AskConfC1 -> Q71),
Q71      = (cam3.cam1.c3SendConfC1 -> Q72),
Q72      = (cam1.cam3.c1SendConfC3 -> Q73),
Q73      = (cam1.cam3.c1AllowsC3Track -> Q74),
Q74      = (cam3.obj.c3StartsTracking -> Q75),
Q75      = (cam3.obj.c3LosesObj -> Q76),
Q76      = (cam3.cam1.c3StartSearchC1 -> Q77
           |cam4.obj.c4SeesObj -> Q83),
Q77      = (cam4.obj.c4SeesObj -> Q78),
Q78      = (cam2.obj.c2SeesObj -> Q79),
Q79      = (cam2.cam4.c2AskConfC4 -> Q80),
Q80      = (cam4.cam2.c4AskConfC2 -> Q81),
Q81      = (cam2.cam1.c2AskConfC1 -> Q82),
Q82      = (endAction -> Q16),
Q83      = (cam3.cam1.c3StartSearchC1 -> Q78
           |cam2.obj.c2SeesObj -> Q84),
Q84      = (cam3.cam1.c3StartSearchC1 -> Q79
           |cam2.cam4.c2AskConfC4 -> Q85),
Q85      = (cam3.cam1.c3StartSearchC1 -> Q80
           |cam4.cam2.c4AskConfC2 -> Q86),
Q86      = (cam3.cam1.c3StartSearchC1 -> Q81),
Q87      = (cam2.cam1.c2AskConfC1 -> Q88),
Q88      = (cam2.cam4.c2AskConfC4 -> Q89),
Q89      = (cam2.obj.c2StartTracking -> Q90),
Q90      = (cam2.obj.c2LosesObj -> Q91),
Q91      = (cam2.cam1.c2StartSearchC1 -> Q92),
Q92      = (cam2.cam4.c2StartSearchC4 -> Q93
           |cam1.obj.c1SeesObj -> Q100),
Q93      = (cam1.obj.c1SeesObj -> Q94),
Q94      = (cam1.cam2.c1FoundObjC2 -> Q95),
Q95      = (cam2.cam1.c2AskConfC1 -> Q96),
Q96      = (cam1.cam2.c1SendConfC2 -> Q97),
```

```
Q97      = (cam2.cam1.c2AllowsTrackC1Obj -> Q98),
Q98      = (cam1.obj.c1StartTracking -> Q99),
Q99      = (endAction -> Q16),
Q100     = (cam2.cam4.c2StartSearchC4 -> Q94).
```

B.2 Modelo comportamental da Arquitetura 1, para o sistema *Smart Cams*, expresso em FSP

Listing B.2: Modelo comportamental da Arquitetura 2 expresso em FSP

```
        Process:
        ArchitectureModel
States:
        86
Transitions:
        ArchitectureModel = Q0,
        Q0      = (c2SeesObj1 -> Q1
                  |c3SeesObj1 -> Q26),
        Q1      = (c2AsksConfC1Obj1 -> Q2
                  |c4SeesObj1 -> Q25),
        Q2      = (c2AsksConfC4Obj1 -> Q3
                  |c4SeesObj1 -> Q14),
        Q3      = (c2StartsTrackingObj1 -> Q4),
        Q4      = (c1seesObj1 -> Q5),
        Q5      = (c2LosesObj1 -> Q6),
        Q6      = (c2StartSearchC1 -> Q7),
        Q7      = (c2StartSearchC4 -> Q8),
        Q8      = (c1FoundObjC2 -> Q9),
        Q9      = (c2AsksConfC1Obj1 -> Q10),
        Q10     = (c1sendsConfC2 -> Q11),
        Q11     = (c2AllowstrackingC1Obj1 -> Q12),
        Q12     = (endAction -> Q13),
        Q13     = END,
        Q14     = (c2AsksConfC4Obj1 -> Q15),
        Q15     = (c2StartsTrackingObj1 -> Q16
                  |c4AsksConfC2Obj1 -> Q19),
        Q16     = (c2LosesObj1 -> Q17),
```

```
Q17     = (c2StartSearchC1 -> Q18),
Q18     = STOP,
Q19     = (c4sendsConfC2Obj1 -> Q20),
Q20     = (c2sendsConfC4Obj1 -> Q21),
Q21     = (c4AllowsC2TrackObj1 -> Q22),
Q22     = (c2StartsTrackingObj1 -> Q23),
Q23     = (c2LosesObj1 -> Q24),
Q24     = (c2StartSearchC1 -> Q7),
Q25     = (c2AsksConfC1Obj1 -> Q14),
Q26     = (c3AsksConfC1Obj1 -> Q27),
Q27     = (c3StartsTrackingObj1 -> Q28),
Q28     = (c1SeesObj1 -> Q29
          |c3LosesObj1 -> Q49),
Q29     = (c3LosesObj1 -> Q30),
Q30     = (c3StartSearchC1 -> Q31),
Q31     = (c1FoundObj1C3 -> Q32),
Q32     = (c3AsksConfC1Obj1 -> Q33),
Q33     = (c1SendsConfC3Obj1 -> Q34),
Q34     = (c3AllowsTrackC1Obj1 -> Q35),
Q35     = (c1StartsTrackingObj1 -> Q36),
Q36     = (c1LosesObj1 -> Q37),
Q37     = (endAction -> Q13
          |c2SeesObj1 -> Q38
          |c1StartSearchC2 -> Q40
          |c1StartSearchC3 -> Q48),
Q38     = (endAction -> Q13
          |c1StartSearchC3 -> Q39),
Q39     = (endAction -> Q13),
Q40     = (endAction -> Q13
          |c2SeesObj1 -> Q41
          |c1StartSearchC3 -> Q47),
Q41     = (endAction -> Q13
          |c1StartSearchC3 -> Q42),
Q42     = (endAction -> Q13
          |c2FoundObj1C1 -> Q43),
Q43     = (endAction -> Q13
          |c1AsksConfC2Obj1 -> Q44),
```

```
Q44        = (endAction -> Q13
             |c2SendsConC1Obj1 -> Q45),
Q45        = (endAction -> Q13
             |c1AllowsTrackC2Obj1 -> Q46),
Q46        = (endAction -> Q13),
Q47        = (endAction -> Q13
             |c2SeesObj1 -> Q42),
Q48        = (endAction -> Q13
             |c2SeesObj1 -> Q39
             |c1StartSearchC2 -> Q47),
Q49        = (c1SeesObj1 -> Q50
             |c3startSearchC1 -> Q80
             |c4SeesObj1 -> Q83),
Q50        = (c3StartSearchC1 -> Q51),
Q51        = (c1FoundObj1C3 -> Q52),
Q52        = (c3AsksConfC1Obj1 -> Q53),
Q53        = (c1SendsConfC3Obj1 -> Q54),
Q54        = (c3AllowsTrackC1Obj1 -> Q55),
Q55        = (c1StartsTrackingObj1 -> Q56),
Q56        = (c1LosesObj1 -> Q57),
Q57        = (endAction -> Q13
             |c1StartSearchC2 -> Q58
             |c1StartSearchC3 -> Q73
             |c4SeesObj1 -> Q77),
Q58        = (endAction -> Q13
             |c1StartSearchC3 -> Q59
             |c4SeesObj1 -> Q71),
Q59        = (endAction -> Q13
             |c4SeesObj1 -> Q60),
Q60        = (endAction -> Q13
             |c2SeesObj1 -> Q61),
Q61        = (endAction -> Q13
             |c2FoundObj1C1 -> Q62),
Q62        = (endAction -> Q13
             |c1AsksConfC2Obj1 -> Q63
             |c4AsksConfC2Obj1 -> Q66),
Q63        = (endAction -> Q13
```

```
                |c2SendsConC1Obj1 -> Q64),
Q64         = (endAction -> Q13
                |c1AllowsTrackC2Obj1 -> Q65),
Q65         = (endAction -> Q13),
Q66         = (endAction -> Q13
                |c2sendsConfC4Obj1 -> Q67),
Q67         = (endAction -> Q13
                |c1AsksConfC2Obj1 -> Q68),
Q68         = (endAction -> Q13
                |c2SendsConC1Obj1 -> Q69),
Q69         = (endAction -> Q13
                |c1AllowsTrackC2Obj1 -> Q70),
Q70         = (endAction -> Q13),
Q71         = (endAction -> Q13
                |c1StartSearchC3 -> Q60
                |c2SeesObj1 -> Q72),
Q72         = (endAction -> Q13
                |c1StartSearchC3 -> Q61),
Q73         = (endAction -> Q13
                |c1StartSearchC2 -> Q59
                |c4SeesObj1 -> Q74),
Q74         = (endAction -> Q13
                |c1StartSearchC2 -> Q60
                |c2SeesObj1 -> Q75
                |c4AsksConfC2Obj1 -> Q76),
Q75         = (endAction -> Q13),
Q76         = (endAction -> Q13),
Q77         = (endAction -> Q13
                |c1StartSearchC2 -> Q71
                |c1StartSearchC3 -> Q74
                |c2SeesObj1 -> Q78
                |c4AsksConfC2Obj1 -> Q79),
Q78         = (endAction -> Q13
                |c1StartSearchC3 -> Q75),
Q79         = (endAction -> Q13
                |c1StartSearchC3 -> Q76),
Q80         = (c4SeesObj1 -> Q81),
```

```
Q81        = (c4AsksConfC2Obj1 -> Q82),
Q82        = (cam4.object1.c4StartsTracking -> Q70),
Q83        = (c3startSearchC1 -> Q81
             |c4AsksConfC2Obj1 -> Q84),
Q84        = (c3startSearchC1 -> Q82
             |cam4.object1.c4StartsTracking -> Q85),
Q85        = (c3startSearchC1 -> Q70).
```