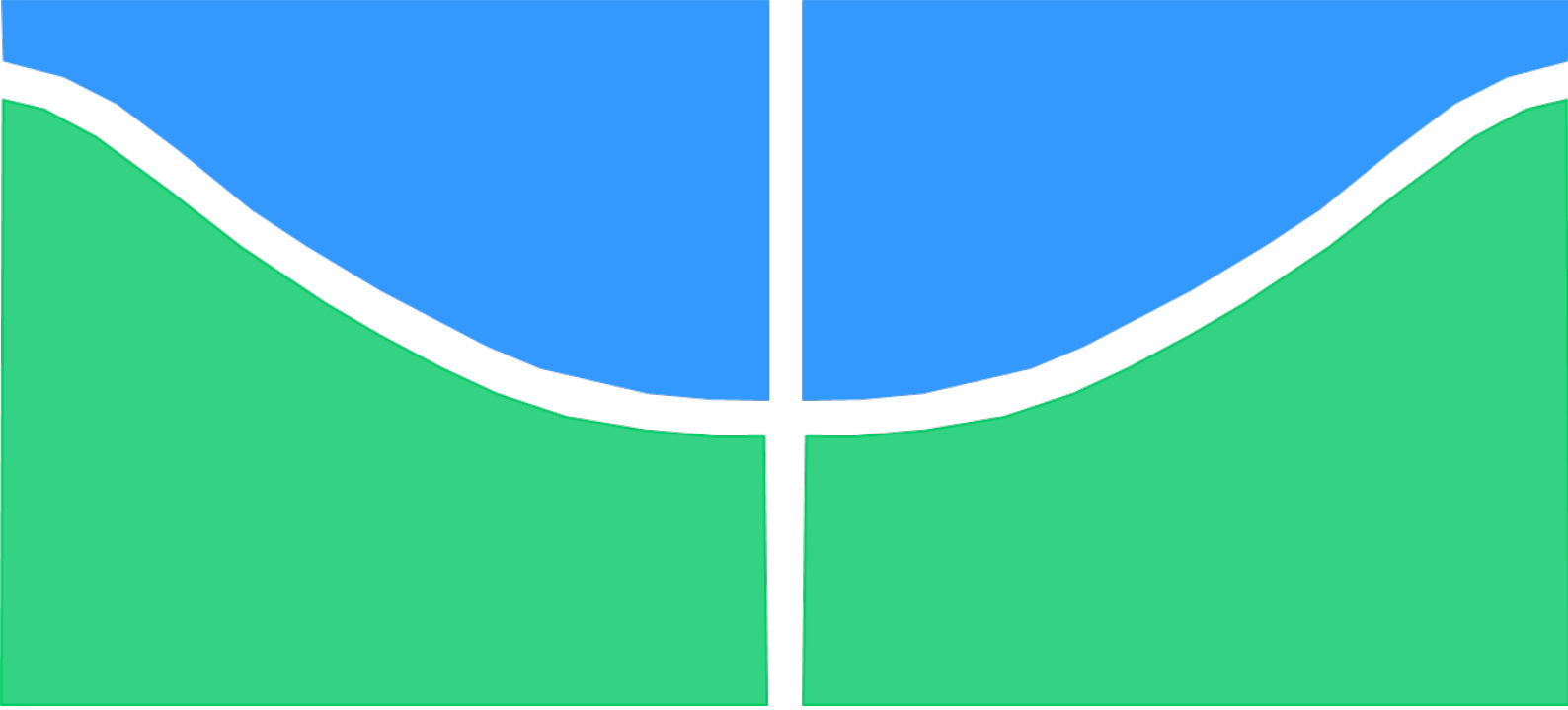

TRABALHO DE GRADUAÇÃO

ABORDAGEM SISTEMÁTICA PARA TESTE DE SISTEMAS AUTÔNOMOS

Por

André Luís Teles Carvalho

Brasília, abril de 2013

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

ABORDAGEM SISTEMÁTICA PARA TESTE DE SISTEMAS AUTÔNOMOS

Por,
André Luís Teles Carvalho

Relatório submetido como requisito parcial de obtenção de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Dr. Antônio Padilha Lanari Bó, ENE/UnB
Orientador

Profa. Dra. Carla Maria Chagas e Cavalcante Koike, CIC/UnB
Co-orientadora

Eng. Michael Fiegert, Siemens AG
Co-orientador

Prof. Dr. Adolfo Bauchspiess, ENE/UnB
Examinador interno

Dra. Mariana Costa Bernardes, ENE/UnB
Examinadora externa

Brasília, abril de 2013

FICHA CATALOGRÁFICA

CARVALHO, ANDRÉ LUÍS TELES
Abordagem Sistemática para Teste de Sistemas Autônomos,

[Distrito Federal] 2013.

vii, 75p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2012). Trabalho de Graduação – Universidade de Brasília.Faculdade de Tecnologia.

1. Teste de Sistemas Autônomos 2.Verificação de Comportamento
3. Validação Funcional

I. Mecatrônica/FT/UnB

REFERÊNCIA BIBLIOGRÁFICA

CARVALHO, A. L. T., (2012). Abordagem Sistemática para Teste de Sistemas Autônomos. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-$n°$021, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 75p.

CESSÃO DE DIREITOS

AUTOR: André Luís Teles Carvalho

TÍTULO DO TRABALHO DE GRADUAÇÃO: Abordagem Sistemática para Teste de Sistemas Autônomos.

GRAU: Engenheiro ANO: 2012

André Luís Teles Carvalho

Dedicatória

A meus pais, Maria Irene e José Arimatéa, e a meus irmãos, Ana Karolina e Julio Cesar

André Luís Teles Carvalho

RESUMO

Sistemas autônomos oferecem varias oportunidades sem precedentes de se aumentar a eficiência de muitas tarefas, desde exploração espacial até tarefas domésticas. Ao mesmo tempo, o risco de mau funcionamento desses sistemas culmina em relutâncias na implantação de novas e complexas tecnologias que esses sistemas permitem utilizar. A questão principal relacionada a esse risco de mau funcionamento é a validação incompleta de sistemas autônomos. Nesse trabalho foi desenvolvida uma abordagem de testes sistemática para sistemas autônomos, que tem como objetivo promover a detecção de falhas, lidando com partes específicas desses sistemas. A ferramenta desenvolvida nesse trabalho foi validada em simulação para a detecção de falhas em um modelo de sistema em máquina de estados.

Palavras Chave: Validação, teste, sistemas autônomos.

ABSTRACT

Autonomous systems offer several unprecedented opportunities to increase efficiency for many purposes, from space exploration to house tasks. At the same time, the risk of malfunction leads to reluctance in using the new and complex technologies that these systems allow. The main problem related to this malfunction risk is the incomplete validation of autonomous systems. In this work we develop a systematic testing approach for autonomous systems, which aims to perform failure detection, coping with specific portions of these systems. The tool developed in this work has been validated in simulation for detecting failures in a state machine model of system.

Keywords:Validation, testing, autonomous systems.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Siglas

UAST	*Unmanned and Autonomous System Testing*
M&S	Modelagem e Simulação
T&E	Teste e Avaliação
UML	*Unified Modeling Language*

Capítulo 1

Introdução

Os sistemas autônomos tem se popularizado e expandido seus campos de atuação. Assim como qualquer outro sistema comercial, esses sistemas precisam ser testados corretamente para que, ao final da fase de desenvolvimento, operem de maneira adequada.

Este capítulo se propõe a prover uma visão geral sobre o tema principal aqui tratado: o teste de sistemas autônomos. Para isso, são abordados os seguintes temas: os testes de sistemas, a problemática a ser tratada, os objetivos específicos do trabalho e a organização do mesmo.

1.1 Contextualização

Antes de se iniciar o desenvolvimento da abordagem proposta nesse trabalho, algumas definições são de grande auxílio à compreensão do conteúdo. O objeto de estudo desse trabalho são sistemas computacionais e, portanto, o termo sistemas deve ser entendido como tal. Existe uma diferenciação entre os sistemas não autônomos (o que já se consegue testar) e sistemas autônomos (o que se pretende testar). Considera-se nesse trabalho que sistemas não autônomos são sistemas que possuem comportamento bem conhecido desde o momento de projeto, devido à ausência total de autonomia. A maioria dos circuitos digitais que constituem aparelhos eletrônicos e *softwares* presentes em um computador pessoal se enquadra nessa definição. Para esses sistemas, existe uma grande variedade de sistemas de teste que serão chamados tradicionais, os quais incluem técnicas como análise estática, análise dinâmica, teste de desempenho etc.

Cada vez mais se verifica a necessidade de testes para garantir que um produto possa ser comercializado, já que quando produtos comerciais são sujeitos a testes inadequados ou não passam por qualquer procedimento de teste pode-se culminar em falhas põem em risco a integridade do ambiente e até mesmo de seres vivos. Dessa forma, é imprescindível que as metodologias de teste para um produto garantam propriedades fundamentais a tais, como funcionalidade e segurança.

As abordagens tradicionais de teste dão ênfase nas medições de desempenho e também de

efetividade, o que inclui características como confiabilidade, manutenabilidade e suportabilidade do sistema. Nesse âmbito, as abordagens tradicionais de teste tem obtido resultados incontestáveis para a determinação de níveis adequados dessas características sobre sistemas não autônomos, o que não pode ser afirmado sobre a utilização das mesmas metodologias em sistemas autônomos [3].

Considerando as características relevantes para o teste de sistemas não autônomos, percebe-se a deficiência das abordagens de teste tradicionais para lidar com sistemas autônomos. Isso se deve ao fato de que, em se tratando de um sistema autônomo, diversas outras características devem tomar a frente da discussão, como flexibilidade e adaptabilidade.

A próxima seção trata a problemática a respeito do teste de sistemas autônomos.

1.2 Apresentação do Problema

Muitas técnicas tem sido utilizadas com sucesso para garantir a efetividade de sistemas em geral. Todavia, com o surgimento de sistemas autônomos, novos desafios para teste fizeram frente às metodologias de teste então existentes, que se tornaram insuficientes para se determinar em termos gerais a qualidade de sistemas autônomos.

A autonomia inerente a um sistema autônomo leva a características como incerteza, comportamento emergente e complexidade, os quais implicam, na prática, em comportamento não determinístico. Isso contrasta diretamente com as funcionalidades de teste para sistemas não autônomos, que assumem que o sistema a ser testado possui um comportamento bem conhecido e sobretudo limitado. Com isso as técnicas de teste tradicional alcançam elevados níveis de confiabilidade em sistemas não autônomos.

Para exemplificar o contraste, imaginemos um sistema não autônomo bastante comum: uma calculadora digital. Por maior que seja o número de funcionalidades de uma calculadora, o seu comportamento é bem definido e conhecido no momento de projeto. Nesse caso, ferramentas tradicionais como a análise lógica e a análise de desempenho são suficientes para testar e validar o funcionamento da calculadora. Já um sistema com algum nível de autonomia, mesmo que pequeno, pode apresentar complicações para teste. Um caso bastante conhecido é o do aspirador de pó robótico Roomba, fabricado pela *iRobot*. Em suas primeiras versões não foi considerado que o robô poderia sofrer quedas quando chegava a um desnível e, mesmo com sensoriamento suficiente para detectar a presença de um desnível, o robô sofria quedas quando encontrava uma escada, falha que foi corrigida nas versões posteriores [7].

Sendo assim, a evolução das metodologias de teste deve ser tal qual a evolução dos sistemas autônomos, permitindo atingir níveis de confiabilidade do sistema autônomo sendo testado e garantindo o grau de segurança e funcionalidade requeridos para esse sistema.

1.3 Objetivos

Estando-se ciente dos desafios de teste de sistemas autônomos, pretende-se desenvolver uma proposta de abordagem para teste de sistemas autônomos que seja capaz de identificar falhas de operação de sistemas autônomos.

Nesse trabalho, é esperado que a abordagem de teste desenvolvida seja sistemática, para que os testes se realizem de forma metódica, regular e ordenada, simplificando assim a visualização dos processos de teste e suas inter-relações. A metodologia de teste deve ser capaz de fornecer dados que possibilitem a realização dos processos de verificação, validação e análise de desempenho de sistemas autônomos.

Outra característica almejada é a versatilidade da metodologia de testes, com a flexibilidade de se incorporar ou suprimir processos de teste de acordo com os objetivos do teste e com as necessidades do sistema testado.

1.4 Metodologia

O primeiro passo para determinar a abordagem que poderá ser utilizada é verificar quais informações o sistema a ser testado pode fornecer e com essa determinação é possível iniciar o desenvolvimento de uma abordagem de teste.

Em seguida, inicia-se a investigação de estratégias de teste. Para isso, divide-se o sistema autônomo a ser testados em porções que apresentam características semelhantes relativas ao teste e validação. Cada porção do sistema autônomo é então analisada individualmente para a definição se suas propriedades relevantes para a determinação do respectivo método de teste.

A abordagem é desenvolvida de modo flexível para poder tratar todos os métodos que são admitidos para as diferentes porções do sistema. Esse procedimento possibilita uma maior adequação da abordagem de teste às necessidades do sistema autônomo.

Quando da proposta de abordagem desenvolvida, são realizadas simulações para determinar a funcionalidade das ferrametas de teste desenvolvidas.

1.5 Organização do Trabalho

O trabalho está dividido em 5 capítulos, incluindo o presente capítulo introdutório. O Capítulo 2 tem como objetivo apresentar os conceitos básicos de sistemas autônomos e de teste desses sistemas, discutindo as definições e abordagens de teste disponíveis na literatura.

No Capítulo 3 serão apresentadas as descrições dos métodos propostos, incluindo as técnicas desenvolvidas para o teste de sistemas autônomos. Nesse capítulo, serão abordadas as metodologias de teste para a camada funcional e para a camada de planejamento de um sistema autônomo, além da estrutura básica utilizada para testes.

As simulações para mostrar as funcionalidades da ferramenta desenvolvida são mostradas no Capítulo 4. Adicionalmente, são encontrados nesse capítulo os resultados do trabalho e as discussões a respeito desses resultados.

Por fim, o Capítulo 5 se destina às considerações finais, detalhando as contribuições realizadas e as perspectivas para trabalhos futuros.

Capítulo 2

Fundamentação

É possível, intuitivamente, dizer do que se trata um sistema autônomo. Entretanto é fundamental relacionar as características desse sistema que sejam associadas aos testes realizados sobre esse tipo de sistema. Nesse capítulo serão apresentadas as características dos sistemas autônomos relativas ao teste e também o que tem sido desenvolvido nessa área.

Observa-se uma constante evolução de sistemas, fazendo necessário também o desenvolvimento de metodologias adequadas de teste. Nesse contexto, várias abordagens tem sido utilizadas para testar sistemas não autônomos proporcionando com isso a obtenção de resultados muito confiáveis e, mais importante, suficientes para a validação do sistema.

Com o surgimento de sistemas mais complexos e sofisticados a obtenção de resultados de teste tem sido uma tarefa mais árdua devido à emergência de novos desafios ao processo de teste.

Nesse capítulo, será feita a descrição das características de sistemas autônomos relevantes para teste na seção 2.1, alguns métodos de testes de sistemas não autônomos na seção 2.2 e o estado da arte, contemplando abordagens de teste e conceitos relevantes abordados na literatura serão discutidos na seção 2.3.

2.1 Sistemas Autônomos

Os sistemas autônomos modernos podem ser quase inimaginavelmente complicados. Muitos desses são constituídos de inúmeros módulos funcionais que atuam de modo conjunto compartilhando informações sobre o ambiente, sobre o próprio sistema autônomo e sobre as tomadas de decisão do mesmo. A descrição de um sistema autônomo que procure abordar todos os seus conceitos e premissas não estão no âmbito desse trabalho, mas podem ser encontrados no trabalho de Hawkinson [8]. Por isso, nesta seção, pretende-se descrever os sistemas autônomos abordando os aspectos relevantes para o desenvolvimento da abordagem de teste desses sistemas.

Os sistemas autônomos podem ser descritos como sistemas que ao menos em determinado nível são capazes de, sem intervenção humana, reconhecer o ambiente, calcular, planejar e executar tarefas [5]. Tais características podem ser compreendidas como a presença de certo nível de autonomia no sistema. No nosso exemplo, o aspirador robótico possui tais características, reconhecendo o ambiente precisa ser aspirado, calculando a trajetória, planejando os próximos locais a serem aspirados e executando a aspiração como principal tarefa, apesar de haver outras tarefas, como evitar obstáculos.

A literatura descreve como os sistemas autônomos (assim como um sistema de rede central e um sistema de sistemas [5]) podem ser caracterizados como sistemas complexos [9]. De fato, há diferenças fundamentais entre os sistemas não autônomos e os sistemas complexos. Considera-se aqui que sistemas não autônomos são aqueles para os quais as metodologias de teste estão consolidadas e são capazes de produzir resultados que garantam o correto funcionamento para a totalidade do sistema testado. Isso se deve ao fato de que o projeto de sistemas não autônomos é realizado de modo que os mesmos operem com tarefas bem definidas e ambientes conhecidos e limitados, como é o caso da calculadora digital. Os sistemas complexos, por outro lado, operam em ambientes não determinísticos [10]. Segundo Macias [3], um sistema complexo é composto de partes interconectadas e é sempre maior do que a soma dessas partes. São características dos sistemas autônomos a emergência de comportamento e a complexidade. A emergência de comportamento é definida como sendo o surgimento de novas e coerentes estruturas, padrões e propriedades durante o processo de auto-organização de um sistema complexo [11]. Nesse contexto, a complexidade é a propriedade de originar comportamentos coletivos de um sistema a partir das relações entre as partes do sistema e de como o sistema interage e faz relações com o ambiente [10]. A figura 2.1 ilustra as diferenças entre sistemas não autônomos e autônomos.

Figura 2.1: Confronto entre Sistema Não Autônomo (à esquerda) e Sistema Autônomo (à direita). Redirados de [1, 2].

Pela natureza dos sistemas autônomos, é fácil perceber que sua característica mais relevante, sua flexibilidade (ou seja, a capacidade de responder e de se adaptar a uma potencial alteração de fatores internos e externos), evidencia sua maior fraqueza quando se trata de testar e validar esses sistemas. Um sistema que apresenta alto grau de flexibilidade associado à autonomia resulta em um número de estados considerado infinito para fins práticos, inviabilizando uma metodologia de teste que busque testar todos os estados do sistema [3].

As propriedades que surgem da autonomia do sistema constituem os principais desafios à elaboração de uma abordagem de testes de sistemas autônomos. Em seu trabalho, Mikaelian [5] define quais são esses desafios, como segue na tabela 2.1.

Tabela 2.1: Desafios ao Teste de Sistemas Autônomos

Complexidade	Sistemas autônomos são tipicamente caracterizados por complexidade, que se deve parcialmente à amplitude do sistema, assim como às interdependências e interações com o ambiente operacional.
Incerteza	Existe uma incerteza significativa cercando o desenvolvimento, teste e operação de sistemas autônomos. Considerando os recursos limitados para teste, decisões estratégicas de teste devem ser consideradas sobre as incertezas relevantes.
Propriedades Emergentes	Sistemas autônomos apresentam comportamento emergente com frequência, definido como comportamento de alto nível que não é pré-programado, mas sim resultados da compilação de ações locais dos subelementos do sistema.

A tabela 2.1 mostra as principais propriedades de um sistema complexo que implicam em dificuldades para teste. Voltando ao exemplo do aspirador robótico Roomba pode-se imaginar a presença dessas propriedades. A incerteza aparece, por exemplo, quando o Roomba se encontra em uma configuração que faz com que a decisão tomada não tenha sido prevista ou tenha sido prevista de forma diferente. Já as propriedades emergentes podem ser consideradas no caso de uma situação conflitante, como o encontro de um lugar que deve ser limpo, mas se encontra em um desnível, fazendo com que o Roomba tenha um comportamento que seja uma combinação desses fatores. A complexidade é o resultado da combinação dos vários elementos presentes no sistema, nesse caso a detecção de obstáculos e desníveis, o estado atual do robô, o nível de bateria, a presença de pó a ser aspirado, etc.

Na próxima seção serão discutidos os desafios à criação da metodologia de teste que surgem da autonomia assim como as abordagens presentes na literatura que atacam o problema.

2.2 Teste de Sistemas Não Autônomos

Os sistemas computacionais não autônomos são algoritmos implementados como código ou como circuito eletrônico. Por serem mais utilizados no campo de sistemas autônomos, serão tradados os testes de sistemas implementados como código, também conhecidos como *softwares*.

Algumas definições são essenciais para a compreensão das metodologias de teste propostas. A primeira delas é a diferenciação entre teste estático e teste dinâmico. **Teste estático** diz respeito a um conjunto de revisões e inspeções que ocorrem no código. **Teste dinâmico** ocorre com a execução do sistema implementado, utilizando casos de teste.

Nesse trabalho, serão tratadas as metodologias que têm como objetivo determinar informações sobre a verificação, a validação e o desempenho de sistemas. As próximas subseções abordam tipos de testes e metodologias de teste relativas principalmente ao teste dinâmico.

2.2.1 Tipos de Teste

Os tipos de teste são tradicionalmente classificados como teste de caixa-preta e teste de caixa-branca. Além disso, existe um tipo de teste chamado teste da caixa-cinza. Esse último teste será negligenciado, porque consiste de uma combinação daqueles dois elementos, mas pode ser encontrado no trabalho de Copeland [12]. As informações a seguir foram adaptadas também desse trabalho.

O **Teste de Caixa-Preta** é uma estratégia na qual o teste se baseia nos requisitos e especificações do sistema sendo testado. Nesse tipo de teste, a produção de resultados de teste se dá apenas por uma relação de entrada e saída do sistema, não sendo necessário nenhum conhecimento de partes e estruturas internas, ou mesmo da implementação do sistema sob teste.

Em contraste, o **Teste de Caixa-Branca** se constitui de uma estratégia na qual o teste se baseia nos caminhos e estruturas internas, ou seja, na implementação do sistema sob teste. Sendo assim, esse tipo de teste requer informações detalhadas sobre as características de programação do sistema.

2.2.2 Metodologias de Teste

Existem diversas metodologias de teste bem conhecidas, como os métodos de injeção de falhas, os testes de mutação, os testes baseados em modelos, etc. Devido aos objetivos de teste (validação, verificação e desempenho) nesse trabalho explicitaremos cinco tipos de teste que serão utilizados posteriormente: a análise estática, a análise dinâmica, o teste de desempenho, a verificação de modelos e a cobertura de código.

A **Análise Estática** [13] consiste da análise de um programa computacional sem que ocorra de fato a execução desse programa. Essa análise é realizada por uma ferramenta que interpreta parcialmente ou integralmente o código do programa considerando as especificações do programa. A ferramenta utiliza essas informações para destacar possíveis erros de codificação (os quais não são encontrados por compiladores) e para verificar matematicamente a adequação do programa a suas especificações através de métodos formais. Ferramentas conhecidas para a realização dessa metodologia são Lint para C/C++ e Jtest para Java.

No caso da **Análise Dinâmica** [14], ocorre a análise de propriedades de um sistema durante sua execução. Para isso são examinadas uma ou mais execuções do sistema para a verificação de propriedades. A análise dinâmica não é capaz de provar que um programa satisfaz uma propriedade em particular, ela detecta violações das propriedades assim como provê informações úteis para programadores sobre o comportamento do sistema. Duas características essenciais da análise dinâmica são: a precisão da informação e a dependência nas entradas do sistema. A precisão da informação é obtida pela instrumentação do sistema com a utilização de uma ferramenta, gerando dados sobre a execução de uma propriedade específica e determinando ligações dentro do sistema. Com a dependência nas entradas do sistema, tem-se uma relação direta entre modificações na entrada do sistema, efeitos nos estados internos do sistema e saídas do sistema. Ferramentas conhecidas de análise dinâmica são a Rational AppScan da IBM, Jtest da Parasoft e Parallel Inspector da Intel.

Segundo Ball [14], as análises estática e dinâmica são metodologias complementares nas seguintes dimensões: completude (identificando as variáveis estáticas e dinâmicas), escopo (dependências internas) e precisão (determinação do domínio de execução) do sistema.

Com relação ao **Teste de Desempenho** [15], é verificado que esse tipo de teste é geralmente realizado com a imposição de uma carga de trabalho sobre o sistema com a finalidade de se determinar características relacionadas à utilização de recursos e tempo de resposta. Além disso, outros atributos de qualidade alcançados pelo teste de desempenho, como a confiabilidade e a escalabilidade. Ferramentas conhecidas de teste de desempenho são o Visual Studio Team System da Microsoft, o LoadRunner da Hewlett Parckard e o JMeter da Apache.

O conceito de **Verificação de Modelos** [16] é a checagem automática da corretude de propriedades de um sistema de estados finitos. As técnicas dessa checagem consistem de processos exaustivos na qual são detectadas incompatibilidades com as especificações e também a presença de *deadlocks*. Ferramentas conhecidas para a verificação de modelos são CPAchecker da SoSy-Lab e BANDERA da SAnToS.

Por fim, trataremos a **Cobertura de Código**. Como descrito por Cornett [17], a cobertura de código é o processo de teste que verifica porções do código utilizadas pela execução de casos de teste, tendo como finalidade:

- Encontrar áreas de um programa que não foram executadas por uma série de casos de teste;
- Criar casos de teste adicionais para aumentar a cobertura do código;
- Determinar uma medida quantitativa de cobertura de código, que é uma medida indireta de qualidade;

Essas características são observáveis em relatórios de cobertura, mostrando todo o código e estatísticas de sua utilização. Ferramentas comuns de cobertura de código são EMMA, Gcov e dotCover.

2.3 Teste de Sistemas Autônomos

Antes de começarmos o desenvolvimento da abordagem de teste proposta, é essencial relacionar o estado da arte nesse campo de estudos. Nesta seção serão apresentados conceitos relativos às características principais das metodologias de teste para sistemas autônomos que vem sendo desenvolvidas nos últimos anos. Muitas dessas serão utilizadas no desenvolvimento da abordagem de teste proposta.

Um objetivo primordial dessa seção é estabelecer a distinção de conceitos utilizados no desenvolvimento de metodologias de teste. Para tanto, serão introduzidas algumas definições que constituirão a terminologia utilizada durante o desenvolvimento deste trabalho. As definições apresentadas na tabela 2.2 foram extraídas de [18, 19, 20].

Tabela 2.2: Definições da Terminologia de teste

Validação	O processo de determinar o grau para o qual um modelo e os dados a ele associados constituem uma representação exata do mundo real da perspectiva da utilização pretendida para o modelo.
Verificação	O processo de determinar se a implementação do modelo e seus dados associados constituem uma representação exata da descrição conceitual e das especificações do desenvolvedor.
Acreditação	A certificação oficial de que um modelo, simulação ou o conjunto de modelos e simulações e seus dados associados são aceitáveis para a utilização para um determinado objetivo.
Teste e Avaliação (*Test and Evaluation*: T&E)	Processo pelo qual um sistema ou componentes são solicitados e os resultados são analisados para prover informação relativa ao desempenho. T&E possibilita estimar a consecução de desempenho, as especificações e a maturidade do sistema para determinar se os sistemas são operacionalmente efetivos, adequados e resistentes para o uso pretendido.
Teste de Desenvolvimento	Compreende todas as atividades de teste que ocorrem enquanto o sistema ainda está sendo desenvolvido.
Teste de Operação	É conduzido de forma a avaliar a efetividade operacional, adequabilidade e resistibilidade como suporte para a produção completa de revisões de decisões. O sistema é testado em várias partes do seu ciclo de vida com a finalidade de avaliar sua funcionalidade.
Modelagem e Simulação (M&S)	Processos pelos quais representações simplificadas da realidade (ou seja, modelos) são utilizadas para predizer como o sistema pode se comportar ou resistir sobre várias condições ou ambientes.

A proposta abordada nesse trabalho trata o desenvolvimento de uma metodologia de teste de sistemas autônomos referente à verificação e ao teste e avaliação, implementável de forma flexível para lidar tanto com testes de operação quanto com testes de desenvolvimento.

Uma forma simples de visualizar a importância do desenvolvimento de uma metodologia de testes específica para sistemas autônomos é confrontar as necessidades do teste desses sistemas com as ferramentas oferecidas por metodologias tradicionais de teste.

Como já foi dito na seção anterior, a característica mais relevante de um sistema autônomo é sua flexibilidade. Essa característica, em conjunto com a autonomia, implica nos principais desafios para o teste de sistemas autônomos. As técnicas atuais de teste e validação são apropriadas para testar sistemas com missões bem definidas, porém é uma tarefa complicada definir as especificações de comportamento de um sistema autônomo, pois esses podem proporcionar um vasto conjunto de comportamentos, muitos dos quais não são conhecidos no momento de projeto [21].

Segundo Macias [3], as abordagens tradicionais de teste se limitam a um único sistema com necessidades e requisitos preliminarmente considerados fixos na fase de projeto. Essas abordagens são preparadas para lidar com sistemas simples e complicados, mas não com sistemas complexos, como são definidos os sistemas autônomos. Entretanto, as técnicas tradicionais não devem ser simplesmente abandonadas, pois é evidente que se pode aprender muito das abordagens tradicionais de teste e, inclusive, reutilizar muitas delas [22].

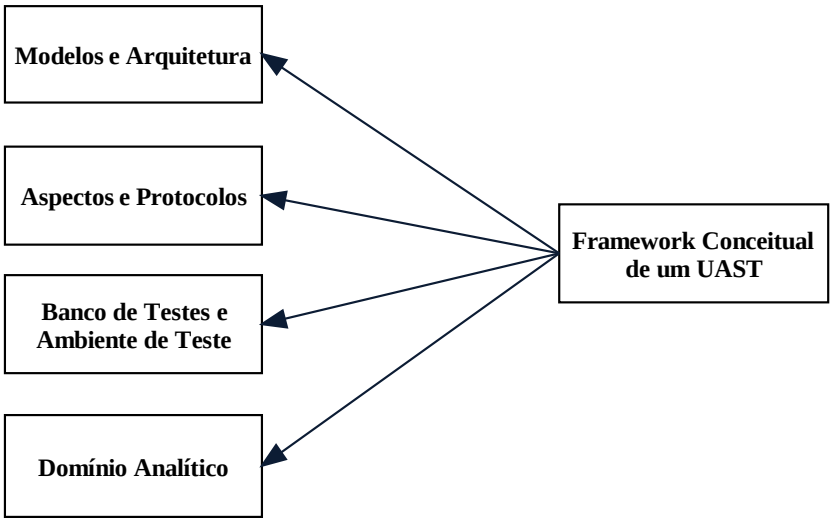

Figura 2.2: Framework para Teste de Sistemas Autônomos. Adaptado de [3].

Além dos pontos discutidos acima, várias outras discrepâncias entre as abordagens de teste tradicional e de sistemas autônomos são abordadas por Ferreira *et al* [23], as quais dizem respeito a uma análise mais aprofundada sobre esse tema como paradigmas de projeto e características evolutivas de sistemas, o que não é objeto desse trabalho.

2.3.1 Metodologias de Teste de Sistemas Autônomos

A partir dos conceitos apresentados anteriormente, pode-se iniciar a descrição de métodos utilizados na literatura para lidar com a obtenção de resultados utilizáveis de teste de sistemas autônomos.

No trabalho de Macias [3] verifica-se a descrição de um *framework* para teste de sistemas autônomos que consiste de 4 categorias: modelos e arquitetura, aspectos e protocolos, banco de testes e ambiente de testes, e domínio analítico, como mostra a figura 2.2. Nessa figura, a sigla UAST se refere ao Teste de Sistemas Autônomos, em inglês *Unmanned and Autonomous Systems Testing*.

A categoria de modelos e arquitetura para o teste de sistemas autônomos se refere à concepção de um *framework* que defina como organizar a estrutura associada à arquitetura utilizada e, além disso, define projeções complementares do modelo do sistema para lidar com a complexidade do mesmo [3].

Em seguida, tem-se a categoria de banco de testes e ambiente de testes de um *framework*, que deve ser composto de um conjunto de dados relativos ao ambiente e às tarefas a serem executadas nesse, de modo a fornecer informações sobre sensoriamento, aquisição e representação de conhecimento, planejamento e comportamento autônomo do sistema [3].

A categoria de aspectos e protocolos se baseia na proposição de que sistemas autônomos podem ser mais bem entendidos ao se utilizar análise multi-aspecto e protocolos que oferecem metodologia de procedimento no projeto e implementação de testes [3].

Por fim, na categoria de domínio analítico, o foco é a identificação e aquisição de informação para viabilizar que o sistema autônomo foque em efetividade das missões e finalização de tarefas [3].

Apesar de as definições feitas por Macias [3] se apresentarem de forma pouco precisa, muitas informações relevantes podem ser abstraídas para o desenvolvimento da metodologia de teste, como a utilização de um modelo para o sistema autônomo, a utilização de dados obtidos pelos testes no ambiente, a avaliação de efetividade das missões e a verificação de realização de tarefas.

Para utilizar as definições feitas por Macias [3], é necessário que se possa inferir do sistema autônomo um **modelo utilizável** para teste. Nos trabalhos de Nesnas *et al* [24] é feita a modelagem do sistema autônomo em duas camadas: a **camada funcional** e a **camada de decisão**.

A **camada funcional** diz respeito às partes de código (ou mesmo módulos) que fazem o interfaceamento com o *hardware*, incorporando todos os recursos do sistema. A **camada de decisão** se trata do planejador do sistema autônomo, onde se encontra a autonomia do sistema. O uso de recursos do sistema para uma operação é obtido por pedidos apropriados à camada funcional. Portanto, todo o modelo e informações de estado sobre o sistema físico residem apenas na camada funcional. Somente informações de estado sobre o planejamento e agendamento se encontram na camada de decisão. Nesse trabalho, os termos camada de decisão e camada de planejamento serão utilizados indistintamente.

Com a divisão da abordagem de teste nessas camadas se torna viável a validação completa da camada funcional e o acesso direto à camada de planejamento, na qual se encontram os desafios à validação do sistema autônomo [21].

2.3.2 Metodologias Auxiliares

Outro aspecto do teste de sistemas autônomos que pode ser abordado para se obter resultados confiáveis de teste é a criação de testes de caso. A definição feita por Binder [25] diz que um caso de teste especifica o estado e a configuração do ambiente antes do teste, as entradas e condições do sistema sendo testado e os resultados esperados. Resultados bastante favoráveis foram obtidos por [26] através da criação de testes de caso orientados a um objetivo do sistema autônomo e utilizando o teste contínuo. O teste contínuo consiste na execução do processo de teste de todas as funcionalidades desenvolvidas sempre que o sistema sofre uma modificação ou que novos casos de teste são desenvolvidos, o que será também estudado nesse trabalho.

Algumas abordagens auxiliares são discutidas na literatura como o método das missões e métodos que podem ser utilizados no estabelecimento de uma relação hierárquica para as capacidades de missões, as tarefas, a efetividade do sistema e os componentes do sistema [27]. Nessa abordagem os casos de teste são determinados pelas funcionalidades especificadas para o sistema autônomo e, a partir da especificação da missão, desenvolve-se o método que o sistema autônomo utilizará para

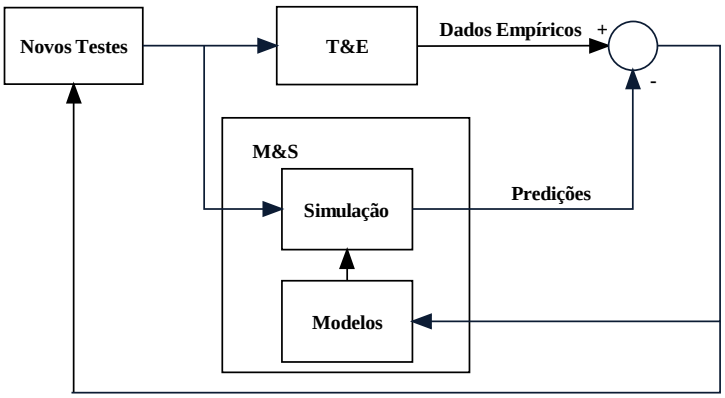

Figura 2.3: Filosofia de Teste de Modelos e Simulação. Adaptado de [4].

cumprir a missão.

No trabalho de Castillo-Effen *et al.* [4] uma relação direta entre o processo de modelagem do sistema autônomo e os testes de simulação são discutidos. A figura 2.3 representa a filosofia de teste proposta, na qual o processo de modelagem e simulação provê predições que dizem respeito às características do sistema, como adequabilidade, resistibilidade, efetividade e desempenho do sistema, enquanto o processo de teste e avaliação provê dados empíricos. A diferença entre esses dados é utilizada para refinar os modelos existentes além de auxiliar na geração de novos testes de caso.

Adicionalmente, Mikaelian [5] propõe a utilização da abordagem de opções reais como uma forma de tratar as incertezas do sistema autônomo. A abordagem de opções reais se preocupa como usar a flexibilidade no momento de projeto para lidar com as incertezas. A flexibilidade pode ser obtida tanto no projeto do sistema autônomo (por exemplo, a colocação de sensores sobressalentes que possam permitir melhor interpretação do ambiente) quanto do sistema de teste (por exemplo, novas tecnologias que permitem a execução de novos tipos de teste). Nessa abordagem existem quatro categorias de técnicas das opções reais: flexibilidade do sistema sob teste, projeto para testabilidade, teste para projeto e flexibilidade do teste. Essas categorias são mostradas na figura 2.4 utilizando as noções de tipo e mecanismo de opção real. Os **tipos** são opções reais que são capazes de lidar com incerteza, enquanto **mecanismos** são viabilizadores da opção real que estão relacionadas a uma flexibilidade identificada.

Assume-se nesse trabalho que as decisões de projeto do sistema autônomo já foram feitas, ou seja, que o sistema autônomo físico já existe e precisa ser testado. Portanto, só será tratada a categoria de **flexibilidade de teste**, que é descrita pela verificação tanto do mecanismo quanto do tipo de opção real no domínio de teste. Retomando o exemplo, novas tecnologias de testes (mecanismos) que possibilitam novas abordagens de teste (tipos).

Figura 2.4: Categorias Gerais para a Abordagem de Opções Reais. Adaptado de [5].

2.3.3 Avaliação da Literatura

Pode-se perceber que as discussões feitas na literatura exaltam a necessidade de novas abordagens de teste para sistemas autônomos. Entretanto, apesar de várias proposições em diversos domínios teóricos, não se verifica numerosas tentativas de implementação de um sistema de testes, inviabilizando para muitos dos casos a comparação de resultados obtidos.

Nesse trabalho serão utilizados os conceitos mencionados nessa seção como funcionalidades e propriedades almejadas da proposta de abordagem de teste a ser desenvolvida, na tentativa de obter resultados práticos que justifiquem a utilização de tais conceitos.

Capítulo 3

Desenvolvimento

As informações a respeito dos desafios de teste de sistemas autônomos e também a respeito de metodologias e conceitos utilizados já foram abordados. Nesse capítulo, será apresentada a proposta de abordagem de teste.

O capítulo seguinte é dividido em cinco seções. A seção 3.2 é dedicada à definição de alguns dos requisitos da metodologia de teste. Na seção 3.3, os elementos da estrutura básica da metodologia de testes são relacionados e a descrição funcional da mesma é realizada. As etapas do processo de teste que preparam o sistema autônomo para a realização dos testes é abordada na seção 3.4. Em seguida, a seção 3.5 desenvolve as características do teste da camada funcional, descrevendo os métodos apropriados para esse teste. Por fim, a seção 3.6 descreve o teste da camada de planejamento e aborda a realização da ferramenta de teste proposta.

3.1 Panorama da Proposta de Teste

O processo de teste de um sistema pode ser composto de vários estágios. A metodologia de teste deve levar sempre em conta os recursos disponíveis para teste e determinar uma estratégia para detectar as falhas do sistema e garantir que elas tenham sido corrigidas.

Com uma metodologia de teste, procura-se determinar as falhas de um sistema para que ela possa ser corrigida. Considerando um sistema autônomo com elevados níveis de autonomia e flexibilidade, o processo de teste não deve buscar testar todas as possibilidades do sistema como já foi dito no capítulo 2. Nesse sentido, a proposta aqui apresentada define o teste de **funcionalidades**. Já que não se pode testar completamente esse tipo de sistema, propõe-se testar as funcionalidades do sistema em um determinado ambiente, verificar as falhas nessa situação e validar as funcionalidades do sistema. A proposta de abordagem de teste desse trabalho é constituída de cinco estágios, como mostra a figura 3.1.

A metodologia proposta se inicia com a implementação de uma **nova funcionalidade** do sistema. A implementação dessa funcionalidade permite a **criação de casos de teste** que têm

Figura 3.1: Proposta de Processo de Teste

por objetivo testar a funcionalidade do sistema em determinados ambientes. Cada caso de teste é composto por arquivos que definem parâmetros internos (do sistema) e externos (do ambiente) além das tarefas que se espera que o sistema execute. Com os casos de teste criados, é realizado seu **teste em ambiente físico**, no qual o sistema com a nova funcionalidade é embarcado e os casos de teste executados nos ambientes apropriados. Nesse estágio, propõe-se gravar os dados de sensor do caso de teste sendo executado para que possa ser utilizado posteriormente. Além disso, nesse estágio é possível perceber operações inadequadas do sistema, que serão úteis na **criação dos arquivos de regras** que serão utilizadas para verificar se o caso de teste apresenta falhas. Em seguida, deve-se utilizar os arquivos de regras, os dados de sensor reais e os casos de teste para o processo de **detecção de falhas** do sistema autônomo. Os **resultados de teste** podem ser então utilizados por desenvolvedores para corrigir as falhas verificadas.

A proposta é que o processo de detecção de falhas se repita sempre que houver alguma modificação do sistema, assim verificando a correção das falhas detectadas, como mostra a realimentação da figura 3.1. No momento em que não se detecta falhas, os casos de teste podem ser então executados em ambiente físico para validar a correção de falhas.

Observando a figura 3.1, percebe-se o destaque dado aos três estágios de teste em que as falhas são detectadas para correção. Dentre esses estágios, a proposta tem foco no estágio de detecção de falhas, que será o objeto de estudo desse capítulo. A etapa de detecção de falhas é composta de uma estrutura básica de teste, etapas preparatórias e etapas de teste da camada funcional e de planejamento. Esses elementos são descritos nas seções subsequentes.

3.2 Requisitos de Desenvolvimento

A determinação de uma proposta de metodologia de testes de um sistema computacional depende naturalmente dos processos desse sistema a serem testados. Normalmente esses processos são implementados como módulos funcionais em se tratando de sistemas amplos. Nos sistemas não autônomos, mesmo que repletos de módulos e inúmeras funcionalidades, o teste dos módulos (entenda-se teste realizado diretamente no código do módulo) individualmente pode ser suficiente para se avaliar a confiabilidade do sistema como um todo, já que as circunstâncias em que o sistema opera são tipicamente conhecidas no momento de seu projeto [21]. Como já foi dito na seção 2.1, isso se deve ao fato de que sistemas não autônomos apresentam comportamento bem definido, permitindo que eles sejam completamente testados de utilizando abordagens de testes já bastante

conhecidas.

No caso de sistemas com elevados níveis de autonomia e flexiblidade, porém, o teste de cada módulo individualmente não é suficiente para atingir a validação do sistema completo. Isso se deve ao alto número de interações dos módulos entre si e dos mesmos com o ambiente, fazendo com que, na prática, seja impossível cobrir todas as possibilidades geradas durante a tomada de decisão e execução de um sistema autônomo [21]. É importante ressaltar que os testes dos módulos isoladamente não são dispensáveis por esse motivo. Os módulos devem ser individualmente testados e validados, utilizando em muitos casos técnicas tradicionais de teste, haja vista que, para tanto, os módulos devem possuir uma relação entrada-saída conhecida e, sobretudo, limitada.

Considerando a característica modular dos sistemas autônomos, idealiza-se utilizar uma metodologia de testes que seja capaz de testar tanto os módulos separadamente quando o sistema autônomo integrado. Para isso, propõe-se uma abordagem de teste composta de etapas funcionais independentes entre si, de modo que as mesmas possam ser acrescentadas ou suprimidas do processo de acordo com as necessidades específicas de cada módulo ou do sistema autônomo como um todo. Não se espera, no entanto, que os módulos e o sistema autônomo sejam testados no mesmo processo, já que os módulos são desenvolvidos e testados muitas vezes antes que o sistema autônomo esteja preparado para a realização dos testes. Com a divisão processo de teste em etapas funcionais independentes, o processo de teste pode conter as etapas adequadas para o teste de cada módulo e do sistema como um todo. Isso garante a flexibilidade do processo de teste, que é fundamental para teste de sistemas amplos com diversas funcionalidades. A definição de etapas do processo de teste será discutida na seção seguinte.

Fatores como autonomia e comportamento emergente fazem parte dos novos desafios à criação de uma metodologia eficaz de testes de sistemas autônomos [4]. Quando sistemas apresentam níveis elevados de autonomia, o processo de teste se torna uma tarefa complicada, pois o sistema pode possuir inúmeras variáveis que podem influenciar em seu comportamento, fazendo com que na prática analisar logicamente o código implementado no sistema seja uma prática insuficiente para se determinar o nível de confiabilidade do sistema autônomo. Além disso, o alto número de variáveis influenciando no comportamento se torna crítico durante o desenvolvimento do sistema autônomo, pois a modificação de um módulo pode influenciar o comportamento de vários outros, fatalmente gerando comportamentos inadequados se não houver um sistema de teste capaz de lidar com essa situação. Os métodos capazes de fazer frente a esse cenário são a geração de casos de teste e a análise de comportamento do sistema. A geração de casos de teste deve criar situações típicas de operação do sistema autônomo e definir o comportamento adequado do sistema de forma inteligível. A análise de comportamento do sistema consiste em utilizar as informações de comportamento adequado adquiridas juntamente com os casos de teste e verificar se o comportamento do sistema autônomo para os casos de teste é aceitável. Retomando o exemplo do robô Roomba, para testar o reconhecimento de desnível e o comportamento após esse reconhecimento formula-se um caso de teste que inclua desníveis no circuito a ser seguido pelo robô. Com isso, verifica-se o comportamento do robô e infere-se a necessidade de desenvolvimento ou aprimoramento de suas funcionalidades.

Além do que foi dito, há dois conceitos aplicáveis às práticas de testes: os **testes de desen-**

volvimento e os **testes de operação** [4]. Testes de desenvolvimento são atividades realizadas enquanto o sistema ainda está sendo desenvolvido, ao passo que testes de operação são aplicados para avaliar a efetividade operacional, a adequabilidade e a resistência do sistema testado para dar suporte às decisões a respeito da revisão da produção do sistema [20].

Em resumo, os requisitos de teste desejados são:

- Capacidade de testar tanto os módulos quanto o sistema integrado;
- Divisão do processo de teste em etapas funcionais independentes;
- Teste de comportamento a partir de casos de teste;
- Flexibilidade quanto às etapas utilizadas;
- Realização de testes de desenvolvimento e de operação;
- Testes Automáticos;

Os requisitos aqui apresentados para a abordagem de testes proposta mostram apenas características gerais do que se espera da metodologia implementada. Em outras palavras, existem outras características mais específicas que serão abordadas nas seções relativas aos seus desenvolvimentos. Nas seções seguintes, essas características serão descritas juntamente com a descrição de processos do sistema de teste.

3.3 Infraestrutura Básica para Testes

Inicia-se nesta seção a determinação da estrutura proposta para a metodologia de teste. O intuito dessa estrutura é criar uma base na qual as etapas de teste serão anexadas e organizadas. Podemos imaginar um sistema análogo, como uma linha de produção, onde há uma esteira e várias ferramentas, cada qual realiza um processo distinto. Nesse sistema, a esteira é análoga à estrutura da metodologia de testes, enquanto as ferramentas são as etapas do mesmo. Na analogia citada, a esteira tem basicamente uma função: levar a peça de um processo a outro. No caso do sistema supervisório, a estrutura deve possuir algumas outras funções, que serão descritas a seguir.

Ao se inserir as etapas na estrutura básica, a mesma deve ser capaz de organizar as etapas dentro do processo de teste. Isso implica em definição de alguns elementos, como a ordem das etapas, a possibilidade de paralelismo, a prioridade de processamento, entre outras. Além desses elementos, a estrutura deve garantir confiabilidade para o processamento das tarefas, fazendo com que o processamento inadequado de uma etapa seja facilmente detectado evitando que o sistema de testes seja interrompido por um período considerável. As duas funções descritas podem ser chamadas de organização e monitoramento do processo de teste e são as mais importantes funções da estrutura do sistema supervisório.

A construção das etapas de testes depende essencialmente das características das funcionalidades a serem testada. Observando os sistemas autônomos comerciais, tal qual o Care-o-bot 3 [28],

é possível perceber a abrangente gama de funcionalidades que podem ser incorporadas a tais sistemas, como detecção de objetos, mapeamento, manipulação de ferramentas, etc. Verifica-se, com isso, que há a necessidade de flexibilizar as possibilidades de definições das etapas de testes. Visando garantir a incorporação de diversas etapas distintas de teste, espera-se que haja flexibilidade também na estrutura básica de testes.

Sistemas de integração contínua, como Hudson [29] e Jenkins [30], são capazes de atender às características descritas acima de forma satisfatória, além de serem sistemas com inúmeras funcionalidades já desenvolvidas no campo de teste de sistemas computacionais.

Durante o desenvolvimento de um sistema computacional relativamente amplo - como são em sua maioria os sistemas autônomos - as várias partes do sistema são muitas vezes criadas simultaneamente por responsáveis especializados em um determinado módulo. Tal condição pode gerar grandes empecilhos no momento da integração do sistema. Alguns elementos potencialmente simplificam a integração, como os sistemas de controle de versão, tais quais Subversion [31] e CVS [32]. Nesses sistemas, o desenvolvimento da parte computacional de um sistema autônomo é mantido em um repositório que organiza e gerencia as atualizações durante o processo de desenvolvimento. A partir desses sistemas, a integração dos módulos do sistema pode ser realizada ainda durante o início do desenvolvimento, evitando que as incompatibilidades no código se acumulem. A associação desse sistema à estrutura básica de teste proporcionará a integração dos módulos do sistema autônomo sendo desenvolvido com a frequência conveniente para teste.

Cada vez que o código do sistema autônomo é atualizado, o sistema de controle de versão altera o índice da versão no repositório. Essa ação em nossa estrutura básica será utilizada como um disparador para iniciar os testes, garantindo que o sistema será testado sempre que houver uma modificação. Adicionalmente, a manutenção de um histórico por parte da estrutura básica de testes também se mostra importante para que seja possível identificar com facilidade as versões que geraram falhas durante os testes.

Considera-se uma estrutura básica como uma associação entre o sistema de controle de versão e a integração contínua, constituindo um repositório com todos os arquivos do sistema a ser testado, incluindo os casos de teste, diretamente ligado à cadeia de testes. A associação mencionada é uma prática conhecida e ambos os elementos possuem interfaceamentos que viabilizam essa prática.

Nas seções seguintes, serão tratados os aspectos relativos à incorporação de etapas no processo de testes.

3.4 Etapas de Teste Preparatórias

As considerações feitas na seção anterior definiram a plataforma e alguns conceitos que serão utilizados para a definição de etapas do processo de teste. A definição das etapas específicas de teste para porções do sistema autônomo, ou mesmo para o sistema completo, será o objeto desta seção.

No sistema de testes introduzido, a estrutura básica de testes deve ser capaz de gerenciar e

monitorar as etapas de teste nela incorporadas, como já foi dito na seção anterior. Com isso, ao se desenvolver uma nova etapa de testes, a mesma deve ser vinculada à estrutura de testes em posição adequada para o seu processamento. Serão consideradas nesse desenvolvimento duas etapas iniciais do processo de teste: *check out* e compilação. É intuitivo que para o sistema em desenvolvimento devem-se utilizar as versões mais atuais dos códigos de cada módulo e, para que o processamento ocorra adequadamente, os mesmos devem ser compilados de forma que operem de acordo com as especificações funcionais do sistema autônomo em desenvolvimento. As etapas preparatórias são descritas na sequência.

- *Check out* - a etapa de *check out* é responsável por disparar o sistema de testes. A cada vez que os códigos são modificados no repositório do projeto, essa etapa inicia seu processamento, que consiste em recuperar as versões mais atuais dos códigos de cada módulo. Em muitas situações, pode ser interessante que os casos de teste sejam obtidos também nessa etapa, e também que a inclusão ou modificação de um deles cause o disparo do procedimento de teste.
- Compilação - nessa etapa os códigos são compilados para que os casos de teste possam ser executados. É comum que a compilação possua configurações especiais para teste, normalmente com o intuito de se obter o máximo de informação relevante sobre a operação dos módulos durante o processamento dos casos de teste.

Essas etapas são as etapas preparatórias básicas. É possível que para determinados sistemas sejam necessárias etapas preparatórias adicionais, como por exemplo, a etapa de configuração do ambiente de testes no caso de sistemas em tempo real. As etapas preparatórias descritas são representadas na figura 3.2, que as mostra inseridas na estrutura de teste.

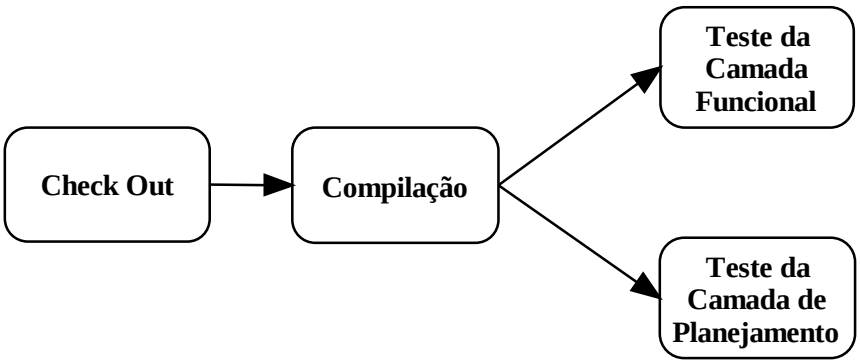

Figura 3.2: Etapas Preparatórias

A estrutura de testes apresentada na figura 3.2 mostra a sequência de etapas de teste. Inicialmente tem-se a etapa de *check out*, seguida da etapa de compilação e posteriormente as etapas de teste da camada funcional e da camada de planejamento. Esses testes, na realidade, são constituídos de várias etapas. Essas etapas serão discutidas mais adiante. É válido ressaltar que existem etapas de teste que não dependem da compilação, pois se tratam de testes estáticos. Em todo caso, a etapa de *check out* garante a presença do código para essas etapas.

As etapas de testes adicionais serão divididas em etapas relativas às camadas funcional e de planejamento presentes em um sistema autônomo, como já foi mostrado na seção 2.3. As duas seções seguintes abordam as características relevantes para o teste e a determinação das etapas de teste para cada camada.

3.5 Teste da Camada Funcional

A camada funcional consiste de códigos que implementam controladores para os dispositivos presentes no sistema autônomo na modelagem realizada por Nesnas *et al.* [24]. Os controladores podem ser simples, concentrados em um único módulo, porém eles podem ser combinações de vários componentes que interagem na implementação de controladores mais complicados, como por exemplo, o controlador de uma pinça de solda, o qual deve controlar a posição angular, a velocidade de fechamento da garra, a pressão máxima sobre a superfície, o tempo de pressão das pinças, etc. Essa série de tarefas faze parte do código do módulo funcional.

Os desafios para teste apresentados pela camada funcional são solucionáveis pelas abordagens tradicionais de teste, pois essa camada consiste de porções de código que executam tarefas bem definidas e sobretudo conhecidas, já que a parte inteligente do sistema autônomo se encontra principalmente na camada de decisão.

As características da camada funcional permitem que suas propriedades sejam verificadas através de alguns procedimentos de teste, como a **análise estática** e a **cobertura de código**. A análise estática é capaz de identificar erros sintáticos da implementação (os quais não são identificados pelo compilador), por exemplo, acesso a locais de memória impróprios ou uso de variáveis não definidas, chegando a determinar possiveis erros semânticos através de métodos matemáticos formais, como pode ser visto em [33]. Muitas ferramentas conhecidas de análise estática garantem a identificação de forma rápida e confiável (PolySpace [34], Coverity [35]), porém poucas são as que garantem essas funcionalidades aliando-as à cobertura completa do código. A cobertura de código é um procedimento simples que identifica o percentual do código que foi testado. É importante ressaltar que esse procedimento por si só não garante que o código foi corretamente testado e, portanto, deve estar em sinergia com outros métodos de teste que garantam confiabilidade e segurança. Caso haja necessidade de verificar as interdependências do sistema, relacionando entradas, saídas e estados internos, indica-se a utilização de **análise dinâmica**.

Na arquitetura de uma camada funcional são utilizados modelos lógicos funcionais [21]. Para lidar com a validação da implementação dos modelos, pode ser utilizada a **análise lógica**. A análise lógica é realizada por *softwares* de **verificação de modelo**, nos quais deve ser inserido o modelo lógico no formato de um autômato ou de uma rede de Petri. A tradução para uma linguagem de modelo (no caso um autômato ou rede de Petri) pode introduzir incertezas ao processo de teste, relativas ao nível de exatidão da tradução. Para contornar esse problema, a análise lógica pode ser aplicada diretamente no código sendo testado, sem que haja a necessidade de traduzir para uma linguagem de modelo. Até 2005 ainda havia uma restrição relativa à quantidade de código a ser testado, que era considerada baixa (da ordem de poucas dezenas de milhares de linhas de código)

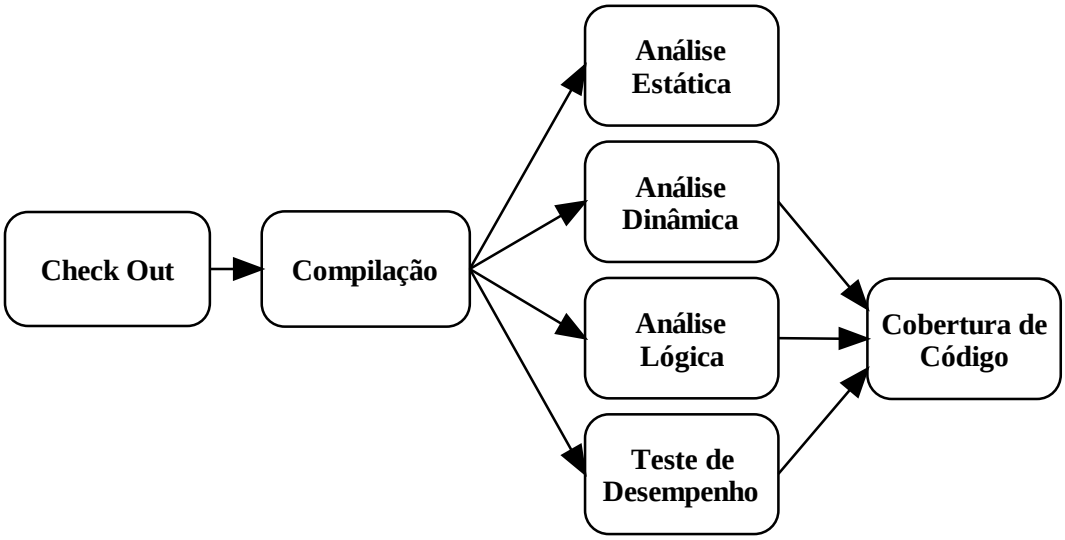

Figura 3.3: Etapas Preparatórias e de Teste da Camada Funcional

o que era suficiente para a aplicação na camada funcional da maior parte dos sistemas autônomos daquela época [21].

Além dos testes quanto à confiabilidade e adequação da camada funcional do sistema, pode ser útil a realização de **testes de desempenho** da camada funcional. Os testes de desempenho nesse caso podem ser obtidos a partir de simulação ou inferência matemática, já que as limitações de interfaceamento entre o código e o *hardware* são bem conhecidas.

Por fim, observa-se na Figura 3.3 a estrutura de teste com as etapas definidas para o teste da camada funcional. As etapas específicas de teste da camada funcional não são dependentes entre si, podendo ser processadas em paralelo, exceto a etapa de cobertura de código, que deve receber os dados de emulação dos casos de teste. Verifica-se que a análise estática não está ligada à cobertura de código, pois essa etapa não realiza a execução do código do sistema autônomo.

Mesmo com todas as etapas descritas acima, para a realização dos procedimentos experimentais, utilizou-se apenas a etapa de cobertura de código como implementação da camada funcional, como é mostrado no trabalho precedente [6]. A próxima seção se dedica a estudar o teste para a camada de planejamento.

3.6 Teste da Camada de Planejamento

A verificação dos módulos que realizam o planejamento é um dos maiores desafios para o desenvolvimento de uma abordagem de testes de sistemas autônomos. Nesse caso, é esperado que o processo de testes consiga obter resultados analisando um espaço de estados praticamente ilimitado, haja vista que os sistemas autônomos apresentam fatores como incerteza na operação, comportamento emergente e complexidade.

Até o momento, a literatura surge com uma série de proposições acerca de teste de sistemas autônomos, algumas delas estão descritas no capítulo 2. Muitos desses trabalhos estão focados em características auxiliares do sistema de teste. As propostas, no entanto, não costumam descrever ou mesmo mencionar a implementação das abordagens propostas, tampouco mostram resultados experimentais. Nessa seção, será desenvolvida uma abordagem de teste baseada no trabalho *Autonomous Systems Testing in Practice: a Systematic Approach* [6]. A mesma abordagem foi implementada e utilizada com sistemas autônomos reais, a saber: um robô de serviço *Care-O-bot® 3* e uma desempilhadeira automatizada. Os resultados para os testes sobre a desempilhadeira automatizada são mostrados na seção 4.2.

A proposta de abordagem de teste da camada de planejamento se inicia com a elaboração de um novo caso de teste, seguida da execução desse teste em um ambiente físico para coleta de dados de sensor e identificação de falhas a serem corrigidas. Assim que esses elementos são adquiridos, o sistema pode ser testado pela ferramenta de testes desenvolvida. Nesse ponto, o sistema é modificado, seja para a correção de falhas ou para a implementação de novas funcionalidades. Sempre que uma modificação é identificada, os testes de caso são emulados utilizando os dados reais de sensores que foram obtidos anteriormente e as informações sobre o ambiente de cada teste de caso. O resultado dessa emulação é comparado com o comportamento esperado pela ferramenta desenvolvida nesse trabalho, verificando se as falhas foram corrigidas. Por fim, quando a ferramenta de teste verifica que não há mais ocorrência de falhas, o sistema é atualizado e testado em ambiente físico para validação. Esses processos serão descritos com mais detalhes na sequência.

Para ilustrar o teste da camada funcional, considere a figura 3.4. Essa figura mostra as etapas de teste da camada de planejamento em conjunto com as etapas preparatórias. As etapas específicas da camada de planejamento são a etapa de emulação e a de verificação. É possível utilizar a etapa de cobertura de código como etapa auxiliar ao teste da camada de planejamento. O objetivo dessa etapa seria prover recursos para a formulação de novos casos de teste para a camada de planejamento. As descrições mais detalhadas dos **processos** presentes nas etapas de teste da camada de planejamento serão feitas mais adiante.

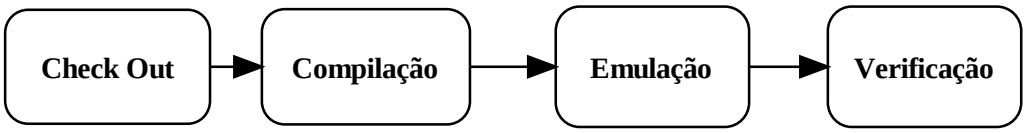

Figura 3.4: Etapas Preparatórias e de Teste da Camada de Planejamento

Durante o desenvolvimento de sistemas autônomos, são gerados inúmeros casos de teste utilizando diversas configurações relativas tanto às propriedades de operação do sistema autônomo, quanto do ambiente no qual o mesmo estará operando. Para que os testes utilizando a ferramenta de teste pudessem garantir repetibilidade, a criação de casos de teste se utilizou das seguintes estratégias:

- Para cada caso de teste criado e executado, são gerados arquivos computacionais que contem

todos os dados de sensores obtidos durante a execução do caso de teste;

- As configurações relativas tanto às configurações de operação do sistema autônomo, quanto ao ambiente no qual o mesmo estará operando, são vinculadas aos respectivos casos de teste através de um **arquivo de parâmetros**;
- Os resultados esperados (ou seja, as especificações de comportamento do sistema autônomo) são vinculados aos respectivos casos de teste através de **arquivos de regras**;

Com essas estratégias, é possível que posteriormente o sistema de teste execute todos os casos de teste com parâmetros de seus respectivos ambientes, com as configurações adequadas do sistema autônomo, utilizando dados reais de operação. A partir desses dados, os casos testes são **emulados** a cada modificação na implementação do sistema autônomo de maneira prática e automática, característica da estrutura básica utilizada. O resultado dessa emulação é o arquivo de registros da emulação do caso de teste, que será utilizado pela ferramenta de teste em conjunto com o arquivo de regras para determinar os resultados de teste. Essas informações são mostradas na figura 3.5.

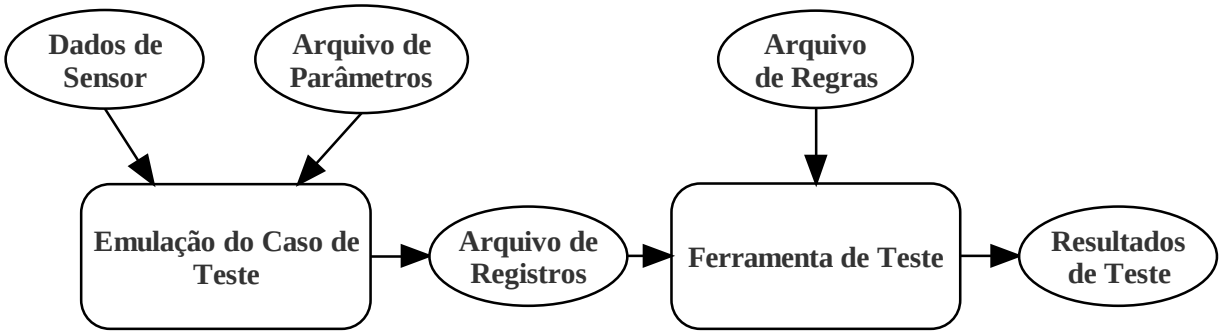

Figura 3.5: Processo de Teste da Camada de Planejamento

A figura 3.5 mostra dois procedimentos para a obtenção dos resultados de teste. O primeiro, já descrito, é a **emulação** do caso de teste, que tem como resultado o arquivo de registros. O segundo processo é a **verificação** do comportamento descrito no arquivo de registro utilizando a ferramenta de testes e o arquivo de regras. Os três elementos presentes no segundo processo da figura serão o sujeito das apresentações na sequência.

Os próximos passos consistem em definir como o sistema autônomo disponibiliza as informações de cada uma de suas configurações únicas, chamadas aqui de **estados**, de sua camada de planejamento durante a operação ou emulação, além de desenvolver a ferramenta de teste com as características e funcionalidades voltadas para o teste de sistemas autônomos. Essas descrições serão abordadas nas subseções seguintes.

3.6.1 Estados da Camada de Planejamento

Ao se executar um sistema autônomo, a camada de planejamento é responsável por utilizar os dados de sensor, a computação nela implementada e, para determinadas funções, informações sobre processamentos anteriores, com a finalidade de determinar o comportamento do sistema autônomo.

O processamento da camada de planejamento nada mais é do que o processamento de funções de planejamento implementadas em código, as quais podem ser dependentes entre si ou independentes. Essas funções, na prática, podem ser modeladas como máquinas de estados, mesmo que não tenham sido implementadas dessa forma. Isso se deve à característica de estados relativa ao processamento computacional aplicado a essas funções durante a sua execução.

Levando-se em conta a característica do processamento computacional, assume-se que a camada de planejamento se comporta como uma série de máquinas de estados durante a execução de um caso de teste do sistema autônomo.

Vale ressaltar que, em sua maioria, essas máquinas de estados apresentam infinitos estados possíveis na prática, o que impossibilitaria a verificação de todos os estados e a própria modelagem. Todavia, os casos de teste representam limitações no espaço de estados, as quais podem ser suficientes para se obter confiabilidade na execução do sistema autônomo em determinados ambientes. Para se obter esses resultados, considera-se a limitação do sistema autônomo por um determinado caso de teste. Com a validação do caso de teste gerado para uma determinada funcionalidade, se valida, por conseguinte, a respectiva funcionalidade para as configurações do sistema autônomo e do ambiente descritas no caso de teste. Após uma série de casos de teste, se obtem como resultado funcionalidades desenvolvidas e verificadas em determinadas situações.

Para ilustrar o processo descrito acima, imaginemos o exemplo já mencionado do robô aspirador Roomba. Para se validar suas funcionalidades de evitar obstáculos e reconhecimento de desnível, é gerado um determinado número de casos de teste. Esses casos de teste consideram ambientes de pisos uniformes de cor branca, cinza e preta. Estando os sistemas desenvolvidos e validados para esses testes de caso, pode-se garantir um nível de confiabilidade para essas funcionalidades nos ambientes citados.

Admitindo a modelagem do sistema como máquinas de estados, ainda é necessário definir a forma através da qual o sistema autônomo disponibilizará as informações sobre os estados para o sistema de teste. Para isso, será utilizada uma característica básica de sistemas computacionais: o registro de informações.

Os sistemas computacionais possuem a potencialidade de registrar as informações sobre seu processamento. Os dados de registro podem ser escritos em um arquivo ou mesmo mostrados em um display sempre que ocorrerem, possibilitando sua utilização para teste. A utilização de registro de dados é incontestavelmente uma das formas mais práticas de se disponibilizar os estados de um sistema.

Durante a emulação de um caso de teste, diversas informações podem ser obtidas e registradas. Assume-se nesse trabalho que no momento da implementação dos códigos foram tomadas as decisões a respeito das especificações sobre como e o que deve ser registrado pela camada de

planejamento. Essa hipótese é geralmente verdadeira em se tratando de sistemas autônomos reais [6] devido às especificações de arquitetura realizadas durante a fase de projeto.

Os registros ou estados que compõem o arquivo de registros de sistemas autônomos podem apresentar dados sensoriais, ações planejadas para o comportamento e estados internos de funcionamento do sistema. Diferenciar esses registros com relação à sua origem no sistema autônomo, ou seja, se ele é um dado sensorial, uma ação planejada ou um estado interno, não traz benefícios para o procedimento de teste, pois essa classificação é facilmente identificável pelo próprio formato do registro. No entanto, algumas classificações permitem analisar características como sequencialidade e estados indesejáveis.

Observando arquivos de registro de execução de sistemas autônomos, podem-se definir as seguintes categorias de estados:

- **Estado não sequencial**: estado que deve estar presente no arquivo de registros, independentemente da posição em que ele se encontra.
- **Estado não sequencial proibido**: estado indesejado no arquivo de registros, independentemente da posição em que ele se encontra.
- **Estado sequencial**: estado que deve estar presente no arquivo de registros, seguindo uma determinada sequência de estados durante a execução.
- **Estado sequencial proibido**: estado indesejado no arquivo de registros entre dois estados sequenciais. No caso em que há mais de um estado sequencial proibido entre dois estados sequenciais, a ordem em que eles aparecem no arquivo de registros é relevante.
- **Quadro de estados proibidos**: estados indesejados no arquivo de registros entre dois estados sequenciais. No caso em que há mais de um estado proibido no quadro entre dois estados sequenciais, a ordem em que eles aparecem no arquivo de registros **não** é relevante.

As duas últimas categorias representarão falha de operação somente quando todos os estados nelas descritos são verificados no arquivo de registros. No caso de um único estado indesejado causar falha, essas categorias são equivalentes e se fazem suficientes para a identificação da falha.

Para uma melhor compreensão das categorias descritas, observa-se o exemplo de execução de uma desempilhadeira automática, mostrado na figura 3.6, no qual há uma evidente redução do espaço de estados. Nesse exemplo, é esperado que a desempilhadeira trace o caminho a ser seguido, decida entre pegar ou não um objeto e percorra o caminho com segurança. A desempilhadeira é instrumentada com sensores a laser para detecção de objetos e sensores de distância para garantir segurança no percurso. Alguns elementos no percurso modificam o estado em que se encontra o sistema do robô. Por simplicidade, esses elementos serão ocultados.

No exemplo da figura 3.6 observa-se os estados de interesse em uma execução de um caso de teste. Vale ressaltar que existem inúmeros outros estados não mostrados que fazem parte da execução do sistema. Além disso, alguns estados que são sequenciais para um caso de teste podem ser não sequenciais para outro, dependendo da funcionalidade sendo testada. Verifica-se na figura

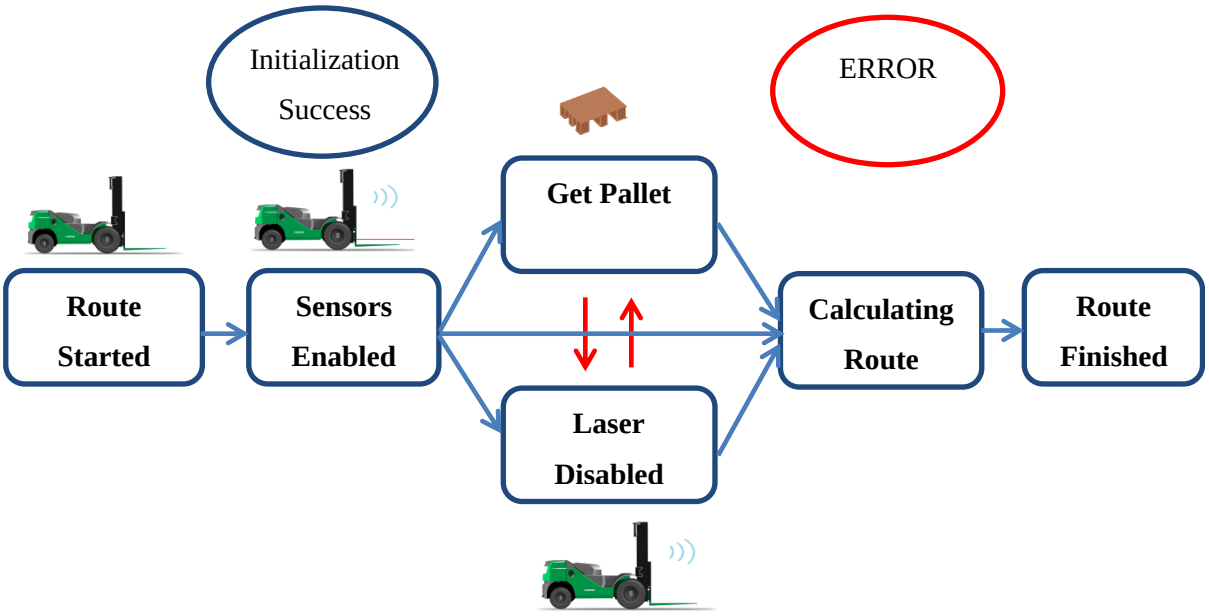

Figura 3.6: Exemplo para Caterogias de Estados. Adaptado de [6].

a presença do estado não sequencial *Initialization Success* e do estado não-sequencial proibido *ERROR*, que indica que a cadeia alfanumérica "ERROR" não pode estar presente em nenhum ponto do arquivo de registros. O importante nesse caso é que o primeiro registro esteja presente no arquivo de registro e que o segundo não esteja, caso contrario o teste apresentará falha. Os estados sequenciais são todos os estados alinhados horizontalmente. Os estados sequenciais não podem ser verificados em ordem diferente do especificado, por exemplo, caso o estado *Calculating Route* ocorra antes do estado *Sensors Enabled* no registro, o teste resultará em falha. Os estados do quadro de estados proibidos se encontram no centro da figura e estão conectados entre si. Isso significa que, caso ambos os estados *Get Pallet* e *Laser Disabled* sejam verificados no arquivo de registros entre os estados *Sensors Enabled* e *Calculating Route*, o teste falhará.

A categoria de estado sequencial proibido não foi exemplificada, mas pode ser verificada em casos reais, mais frequentemente com apenas um estado sequencial proibido entre estados sequenciais. Não é comum definir vários estados sequenciais proibidos, mesmo porque a definição dessa categoria de estado como regra de análise é bem menos intuitiva que as demais. Pode-se imaginar um exemplo para o robô aspirador Roomba: o estado "Desnível Perigoso Detectado" seguido do estado "Continuar Curso" poderia fazer com que o robô caísse de um degrau.

Após a apresentação das características dos estados associados à camada de planejamento, pode-se iniciar a apresentação dos elementos de teste. A verificação do arquivo que contem os registros para fins de teste é intrinsecamente dependente de três elementos: das características do arquivo de registros, das especificações de regras para teste e da ferramenta de teste de registros. Esses três elementos serão discutidos nas próximas subseções.

3.6.2 Arquivo de Registros

O primeiro elemento do teste da camada de planejamento a ser discutido é o arquivo de registro. Esse arquivo será responsável por manter as informações processadas durante a emulação de um caso de teste, tornando-as disponíveis para a ferramenta de teste.

Como foi dito na subseção anterior, o processamento da camada de planejamento pode ser modelado como um conjunto de máquinas de estado. O processamento dessas pode apresentar, na prática, infinitos estados devido ao conjunto ilimitado de circunstâncias que o sistema autônomo pode encontrar. Para que seja possível realizar os testes da camada de planejamento, a redução do espaço de estados ocorre naturalmente ao se criar um caso de teste finito.

Assim, durante a emulação de um teste de caso, os estados da camada de planejamento são colocados no arquivo de registros e podem ser classificados utilizando a nomenclatura desenvolvida na subseção anterior para fins de teste. Esse aspecto não é importante para a descrição do arquivo de registros em si, mas para a formulação das regras de teste, que será apresentada na próxima subseção.

O ponto relevante na caracterização do formato dos registros é a definição do formato quanto à variabilidade do registro. Verifica-se que os possíveis formatos dos registros são definidos de forma binária, ou seja, o registro pode ser *variável* ou *invariável*:

- **Registro Invariável**: a informação não varia de teste para teste ou, em outras palavras, o registro consiste de uma ou mais cadeias alfanuméricas conhecidas. Exemplo "Initialization Success".
- **Registro Variável**: o registro contem elementos variáveis que descrevem normalmente a interação do sistema autônomo com o ambiente, ou características internas do sistema autônomo. Exemplo: "Pallet at x:42.1 y:22.0".

Os registros variáveis podem apresentar variáveis tanto contínuas quanto discretas, além de relações lógicas entre as variáveis presentes no registro. Essa e outras definições serão mostradas na subseção seguinte.

Tanto os formatos dos registros quanto as categorias de estados são fundamentais para o desenvolvimento da sintaxe utilizada para a definição das regras de teste dos casos de teste. A próxima subseção aborda o desenvolvimento da sintaxe utilizada para definir o arquivo de regras.

3.6.3 Especificações de Sintaxe das Regras de Teste

Para a realização da verificação de comportamento a partir de registros, três elementos são necessários: o arquivo de registros, a ferramenta de testes e um arquivo de regras. O formato do arquivo de registros é determinado pelos desenvolvedores do sistema autônomo, porém, ainda resta definir a linguagem através da qual a ferramenta de teste de registro é informada pelo arquivo de regras sobre o que se espera no arquivo de registro. Esse procedimento de verificação é ilustrado

na figura 3.7. Nessa figura, o processo de verificação é representado pela ferramenta de testes, que tem como entradas o arquivo de registros e o arquivo de regras.

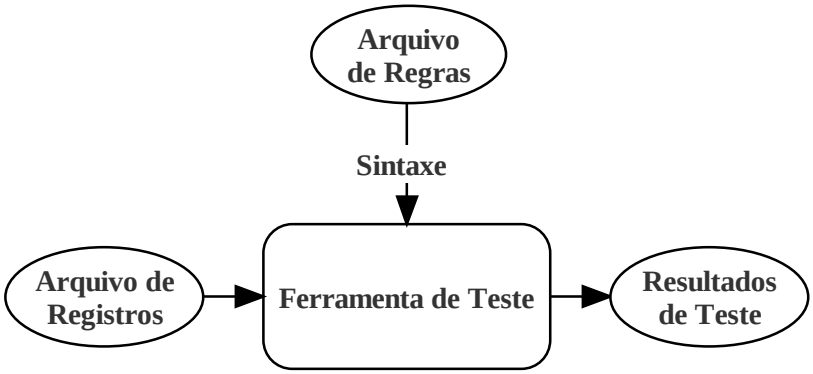

Figura 3.7: Processo de Verificação de Comportamento

O arquivo de regras é o arquivo no qual as informações inferidas de uma execução adequada do sistema autônomo estão contidas. As informações sobre o que se espera da execução do teste serão transmitidas do arquivo de regras para a ferramenta de teste através de dados que devem respeitar uma sintaxe que a ferramenta de testes possa utilizar, como mostra a figura 3.7. Essa subseção se dedica a desenvolver as normas sintáticas da linguagem utilizada para descrever as regras de teste.

Na subseção 3.6.2, foram definidos dois formatos de registros e cinco categorias de estados. Baseando-se nessa descrição, é preciso desenvolver normas sintáticas que sejam capazes de descrever completamente o conjunto de combinações entre as categorias e os formatos.

Considerando primeiramente os formatos de registros, pode-se determinar a sintaxe para exprimir a ideia de cada formato. Para fazer as atribuições de regras, será atribuído a cada formato uma **função**, ou seja, uma palavra que descreverá a ação da regra para o formato que a representa. Para o formato de registros nomeado **registro invariável**, os registros são meramente uma série conhecida de cadeias alfanuméricas que devem estar contidas no arquivo de registro. Para esse formato, escolheu-se a função **ContainsString**. Em se tratando do formato de registros intitulado **registro variável**, os registros são compostos tanto de palavras fixas quanto variáveis. Essas variáveis são descritas para teste através de restrições dos valores que a mesma poderá assumir durante a execução de um caso de teste. Para representar esse formato, determinou-se a função **StringConstraints**. Mais informações sobre essa função serão descritas adiante. Vale ressaltar que para a definição das funções relativas aos formatos dos registros não se considerou regras para registros indesejáveis. Essa discussão, assim como a discussão a respeito das categorias de estados, é mostrada na sequencia.

Ao se comparar os formatos de registros com as categorias de estados, percebe-se que a natureza desses é bastante diferente (o primeiro provém da variabilidade da informação, enquanto o segundo, da característica de processamento). Apesar desse fato, para a definição de uma regra de teste esses dois conceitos são extremamente importantes. Percebe-se que um registro de um

Tabela 3.1: Definições Sintáticas

	Registro invariável	**Registro variável**
Estado não sequencial	ContainsString	StringConstraints
Estado não sequencial proibido	Forbidden ContainsString	Forbidden StringConstraints
Estado sequencial	Sequential ContainsString	Sequential StringConstraints
Estado sequencial proibido	Forbidden Sequential ContainsString	Forbidden Sequential StringConstraints
Quadro de estados proibidos	ForbiddenWindow ContainsString	ForbiddenWindow StringConstraints

determinado formato pode representar qualquer categoria de estado e, por isso, a regra de teste deve ser modificada para que se represente a categoria de estado que está sendo testada. Portanto, para uma dada categoria, será atribuído um **modificador**, que será utilizado juntamente com as funções descritas para determinar as regras de teste. A primeira categoria de testes a ser considerada é a de **estado não sequencial**. Para simplificar a notação, essa categoria pode ser considerada a categoria padrão e nenhum modificador é atribuído. Assim, uma função sem modificador é entendida como pertencente a essa categoria. A próxima categoria se refere ao **estado não sequencial proibido**. Essa categoria deriva da categoria padrão, apenas adicionando a ideia de proibição, a qual será traduzida como o modificador **Forbidden**. Na sequência, tem-se o **estado sequencial**, que carrega a ideia de sequencialidade, perfeitamente descrita pelo modificador **Sequential**. Para a categoria de **estado sequencial proibido**, será empregada a mesma ideia de proibição anteriormente dita, representada pelo modificador **Forbidden**. Por fim, a categoria de **quadro de estados proibidos** será representada pelo modificador **ForbiddenWindow**. As definições de sintaxe feitas estão resumidas na tabela 3.1.

Utilizando as normas sintáticas da tabela 3.1, as regras de teste adequadas para testar a execução mostrada na figura 3.6 seriam:

ContainsString: "Initialization Success"
Forbidden ContainsString: "ERROR"
Sequential ContainsString: "Route Started"
Sequential ContainsString: "Sensors Enabled"
ForbiddenWindow ContainsString: "Laser Disabled"
ForbiddenWindow ContainsString: "Get Pallet"
Sequential ContainsString: "Calculating Route"
Sequential ContainsString: "Route Finished"

No exemplo de regras mostrado, é válido ressaltar que as regras não sequenciais, nesse caso as duas primeiras, podem ser descritas em qualquer posição dentro do arquivo de regras. Isso se deve ao fato de que o processamento dessas regras é diverso do processamento de regras sequenciais. Para uma regra não sequencial, o arquivo de registros será checado completamente, do primeiro ao último registro. Já para uma regra sequencial, o arquivo será checado a partir do primeiro registro após o registro relacionado à regra sequencial anterior. Discussões mais aprofundadas em se tratando da forma de checagem do arquivo de registros para as várias regras podem ser vistos

na subseção 3.6.4 com maiores detalhes.

Até o momento foram definidas as funções e os modificadores de uma regra. Todavia, a utilização da função **StringConstraints** para descrever uma regra de teste ainda não seria possível. Isso porque ainda deve-se definir a sintaxe para descrever as restrições de valores das variáveis que compõem essa regra. As variáveis a serem checadas podem assumir basicamente 3 formatos: números inteiros, números fracionários e frases. Para associar o tipo à variável, serão utilizados os especificadores de formato: **%d** para números inteiros, **%f** para números fracionários e **%s** para frases. Esses especificadores estarão presentes na frase principal da regra de teste, como segue:

StringConstraints: "Atuador:%s Velocidade:%f Aceleração:%d.%d"

Seguindo a ordem na qual os especificadores de formato aparecem na frase, serão definidos seus argumentos possíveis utilizando a palavra **Argument** seguida do número de sequência do especificador e, na sequência, as restrições da variável entre colchetes. É possível ainda definir uma lógica de teste para os argumentos, utilizando a palavra **Logic** seguida das expressões lógicas com os operadores e os números dos argumentos envolvidos. As possíveis operações lógicas e a sintaxe dos operadores são mostradas na tabela 3.2. As expressões lógicas podem ser construídas com o auxílio de parênteses. Ao se finalizar a definição dos argumentos e da lógica, utiliza-se a palavra **End** para indicar o encerramento da definição da regra.

StringConstraints: "Atuador:%s Velocidade:%f Aceleração:%d.%d"
Argument 1: [on]
Argument 2: [3.15, inf]
Argument 3: [0, 1]
Logic: (!1 & 2) | (1 & ! 2 & 3)
End

No exemplo mostrado, o primeiro argumento refere-se ao especificador de frase *%s*, e seu valor especificado é *on*. O segundo argumento se refere ao especificador de número fracionário *%f*, e seu valor especificado é entre 3.15 e infinito (podem ser utilizadas as palavras **inf** para infinito positivo e **ninf** para infinito negativo). O terceiro argumento se refere ao primeiro especificador de número inteiro *%d*, e seu valor esperado está entre 0 e 1. O quarto argumento é o segundo especificador de número inteiro, para o qual o argumento não é descrito, de onde se infere que esse argumento (que definiria a parte fracionária do valor da aceleração) não é importante para o teste. A logica implementada mostra duas operações entre parênteses, a primeira indica que o atuador não deve estar no estado *on* e a velocidade deve estar entre 3.15 e o valor máximo possível, enquanto a segunda indica que o atuador deve estar no estado on, a velocidade deve ser inferior a 3.15 e a aceleração deve estar entre 0 e 1. A regra de teste apresentará sucesso se qualquer dessas operações for verdadeira.

As definições sintáticas desenvolvidas nessa seção permitem a formulação de uma grande diversidade de regras para teste. A utilização dessas regras, em conjunto com os registros de sistemas autônomos, será discutida na subseção 3.6.4, na qual será apresentada a ferramenta de testes

Tabela 3.2: Operações Lógicas e Seus Operadores

Operação	Sintaxe do Operador
NÃO Lógico	!
E Lógico	&
OU Lógico	|

desenvolvida.

3.6.4 Ferramenta de Teste de Registros

O elemento responsável pela produção de resultados de teste da camada de planejamento é a ferramenta de teste. Ela se utiliza, para tanto, do arquivo de registros e de um arquivo com as regras para a análise da execução do caso de teste, podendo ser classificada como um analisador de registros (também conhecido, em inglês, como *log analyser*).

Os analisadores de registros são ferramentas geralmente utilizadas para aplicações *Web*, sendo que para analisar tais aplicações, essas ferramentas tem como característica a utilização de contadores e abordagens estatísticas, não sendo, assim, adequado para a abordagem que foi proposta nesse trabalho. Serão desenvolvidas nessa subseção as funcionalidades e características da ferramenta de teste proposta para desempenhar o teste de sistemas autônomos.

Com o objetivo de simplificar a descrição do modo de operação da ferramenta de testes de registros, será feita a divisão da descrição em três partes, referentes às regras de estados não sequenciais, às regras de estados sequenciais e às regras do quadro de estados proibidos. A estrutura de controle para cada uma das partes será descrita através de fluxogramas que utilizam o padrão de linguagem de modelamento unificado UML (*Unified Modeling Language*), na qual os blocos de contorno circular representam pontos de início e fim, os blocos retangulares são os processos e os blocos em forma de losango são estruturas de decisão. Para a construção dos fluxogramas foram realizadas simplificações para torná-los mais inteligíveis. Essas simplificações - de relativa importância para a implementação da estrutura de controle - serão discutidas como referências aos fluxogramas que seguem. Esses fluxogramas serão mostrados também no Anexo I de forma expandida para facilitar a visualização. Adicionalmente, o código desenvolvido para a verificação de comportamento, escrito em Ruby, é mostrado no anexo II.

O primeiro fluxograma, mostrado na figura 3.8, representa a execução da ferramenta de teste para dois tipos de regras: **regra de estado não-sequencial** e **regra de estado não-sequencial proibido**. Partindo do ponto de início (**Início**), no qual se presume que os arquivos de regras e de registros estão disponíveis, a primeira regra não verificada do arquivo de regras é lida (**Leitura da Próxima Regra**). Após se ler a regra, o ponteiro de leitura do arquivo é apontando para a primeira linha do arquivo de registros (**Início do Arquivo de Registros**) e em seguida, é iniciado o processo de leitura (**Leitura do Próximo Registro**). Continuando o processo, compara-se o registro lido com a regra (**Regra e Registro se Adequam?)** e se a regra e o registro se adequam podem ocorrer duas situações analisadas por (**Regra de Estado Proibido?**). A primeira situação ocorre se a regra for de estado proibido e, nesse caso, o teste resultaria em

falha, pois um estado indesejado estava presente no arquivo de registros. O teste seria então interrompido assim que ocorresse a falha (**Falha**). Na segunda situação, a regra não é de estado proibido. Sendo assim, resta saber se todas as regras não sequenciais foram testadas (**Fim das Regras Não Sequenciais?**) e caso essa situação ocorra, a execução do teste é finalizada com sucesso. Por outro lado, se ainda houver regras não sequenciais a serem testadas, retorna-se ao ponto de partida (**Leitura da Próxima Regra**). Considerando-se agora que a regra e o registro não se adequam, será necessário passar para o próximo registro e refazer o teste. Porém é necessário saber se ainda há mais registros (**Fim do Arquivo de Registros?**) e se for o caso, o próximo registro é lido (**Leitura do Próximo Registro**) e a verificação do registro é realizada novamente. Na situação oposta, todos os registros foram verificados e não se adequaram à regra e a verificação de regra de estado proibido é novamente realizada (**Regra de Estado Proibido?**). Se a regra atual não é de estado proibido, ocorreria uma falha, porque não seria detectado em todo o arquivo de registro um estado que era esperado. Por outro lado, se a regra for de estado proibido, ocorreria o esperado (ou seja, o arquivo de registros seria completamente verificado e nenhum estado proibido seria detectado). Nesse caso, o teste segue com a próxima regra de teste, caso exista.

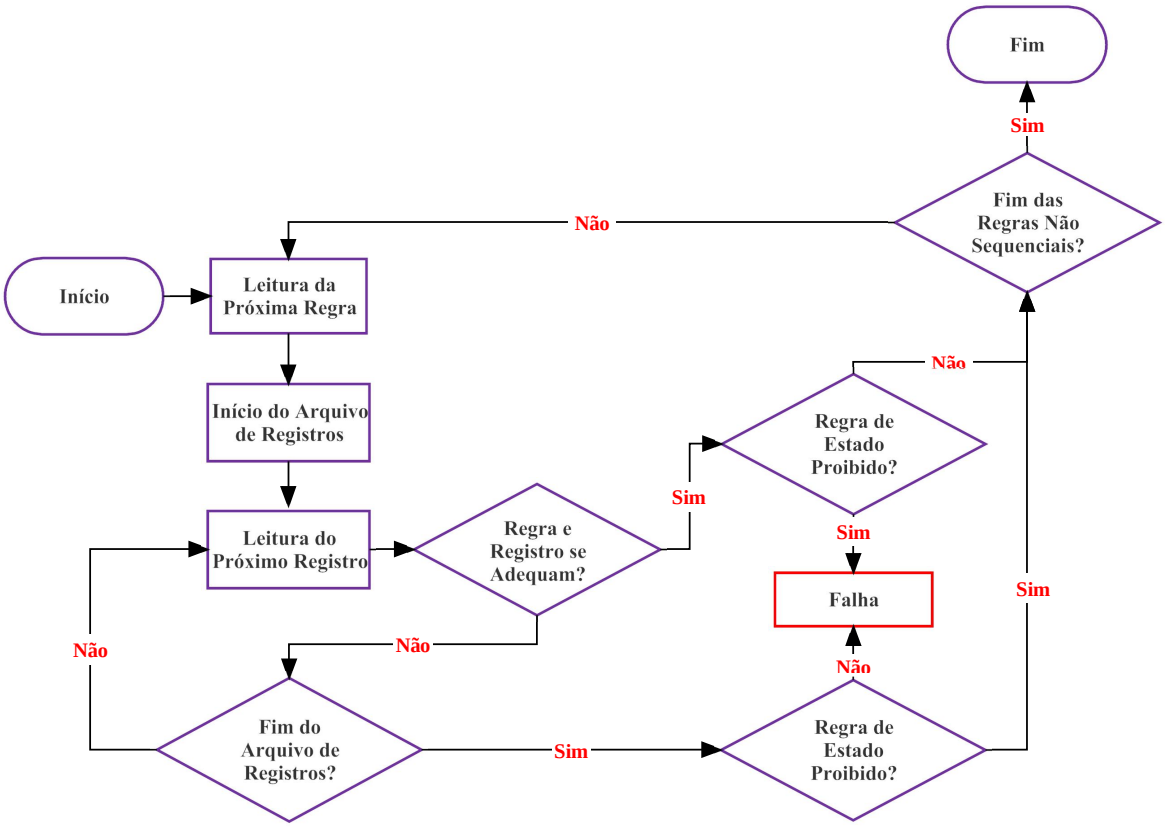

Figura 3.8: Fluxograma para Regras de Estados Não-Sequenciais

Com o intuito de simplificar a visualização, a execução da ferramenta de testes para regras de estados sequenciais será dividida em dois fluxogramas. O primeiro fluxograma, mostrado na figura 3.9, se refere ao procedimento de teste com **regras de estado sequencial** e **regras de estado sequencial proibido**, enquanto o segundo fluxograma, mostrado da figura 3.10, representa a execução das mesmas **regras de estado sequencial** e também das **regras do quadro de**

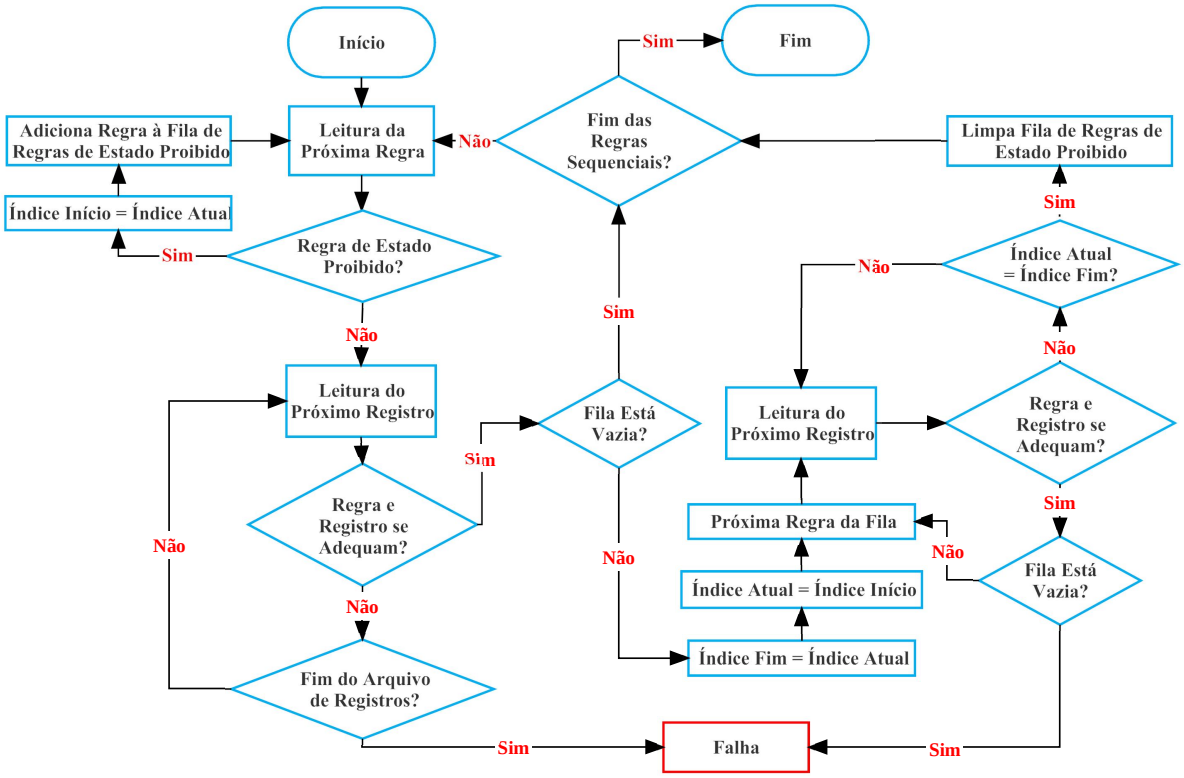

Figura 3.9: Fluxograma para Regras de Estados Sequenciais

estados proibidos.

O fluxograma da figura 3.9 se inicia com a leitura de uma regra de testes ainda não utilizada. Caso essa regra seja de estado proibido, se salva o índice atual de leitura do arquivo de registros como o registro de início para teste e adiciona-se a regra proibida a uma fila. Repete-se então o processo até que uma regra sequencial que não seja de estado proibido ocorra. Ao ocorrer essa regra, procede-se a leitura do próximo registro e a comparação com a regra. Se a regra e o registro não se adequarem e ainda houver mais registros, o próximo registro será lido e comparado com a regra. Porém, se a regra e o registro não se adequarem e não houver mais registros para comparação, ocorre uma falha, porque um estado que deveria estar contido na sequencia de estados desejados não é encontrado. Na situação em que o registro e a regra se adequam, se a fila de regras do estado proibido estiver vazia e não houver mais regras a ser lidas, é finalizado então, com sucesso, o processo de teste. Caso haja mais regras, o processo retorna ao ponto inicial. Uma situação em que uma regra e um registro se adequam e a fila de regras de estado proibido possui regras, indica que anteriormente àquela podem existir estados sequenciais proibidos ainda não verificados. Para a verificação de tais estados, define-se o índice atual do arquivo de registros como sendo o índice para o fim de verificação de estados proibidos. Então, o índice do arquivo de registros é retrocedido até o registro no qual as regras de estado sequencial proibido foram adicionadas à fila. Nesse ponto, se utiliza a próxima regra da fila e se compara ao próximo registro. Se todas as regras na fila se adequam a registros, a fila se esvazia e ocorre uma falha. Entretanto, ao se chegar ao registro de fim da verificação de estados proibidos e ainda houver regras na lista, não ocorreu a falha e a lista pode ser limpa. Desse ponto, se ainda houver mais regras a serem testadas, é

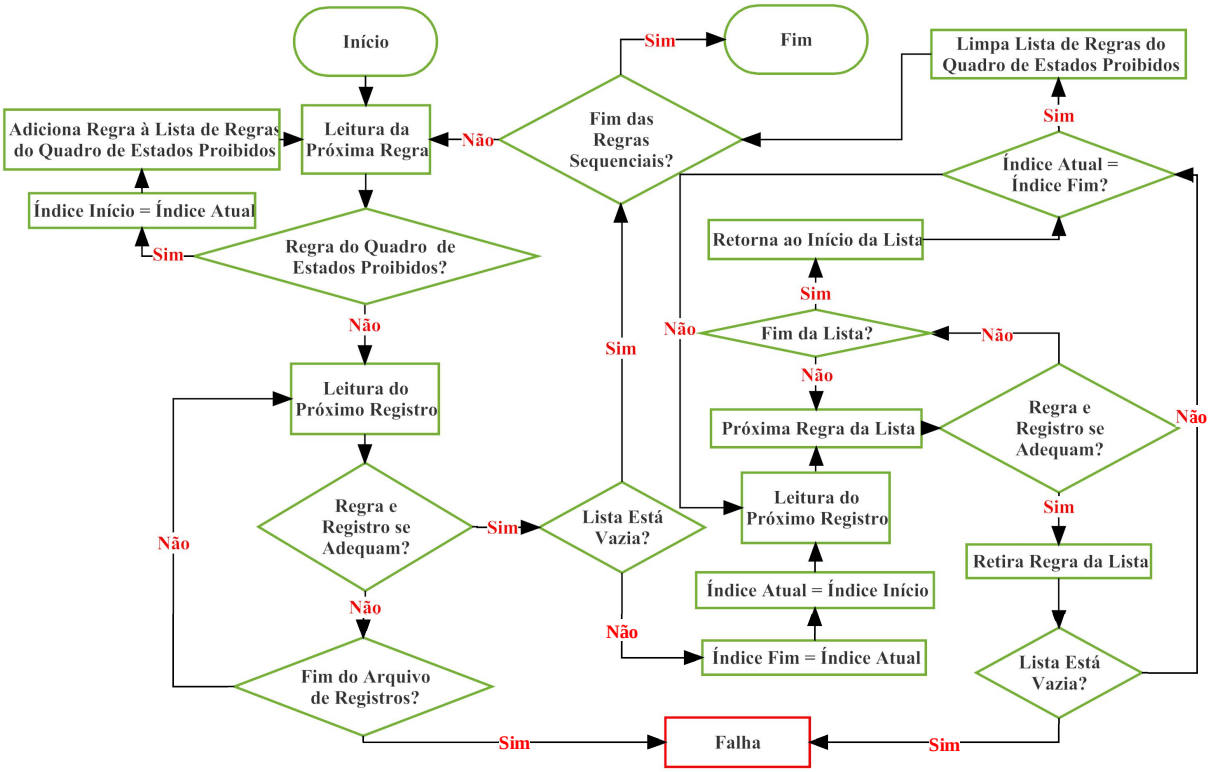

Figura 3.10: Fluxograma para Regras do Quadro Estados Proibidos

dado procedimento ao teste. Se não, o processo é finalizado com sucesso. É importante ressaltar que, para a construção do fluxograma, foi considerado que todas as regras de estado proibido são sucedidas de regras de estado não proibido. Sem essa consideração, o procedimento pode entrar em um laço colocando regras de estado proibido na fila e fatalmente para ao tentar ler uma nova regra que não existiria. Durante a implementação essa questão é solucionável ao se definir que as regras de estado proibido se referem aos estados definidos do índice atual do arquivo de registros até o final do mesmo arquivo.

Para o fluxograma da figura 3.10, a descrição da execução é basicamente a mesma na parte inicial, relativa à execução de regras de estado não proibido. A única ressalva é que as regras do quadro de estados proibidos são colocadas em uma lista, e não em uma fila. Por conseguinte, será descrito em seguida somente o procedimento para a verificação do quadro de estados proibidos. Primeiramente, assim como na descrição do fluxograma anterior, define-se o índice atual do arquivo de registros como sendo o índice para o fim de verificação de estados proibidos. Então, o índice do arquivo de registros é retrocedido até o registro no qual as regras do quadro de estados proibidos foram adicionadas à lista. Ocorre então a leitura do próximo registro e da próxima regra da lista. Diferentemente do caso anterior, para cada registro são comparadas todas as regras da lista. Se uma das regras da lista se adequar a um registro, essa regra é retirada da lista. Ao se obter uma lista vazia, ocorre uma falha, pois todas as regras proibidas do quadro ocorreram. Em uma situação em que nenhuma das regras da lista se adequou a um registro, retorna-se ao início da lista e realiza a leitura do próximo registro. Esse procedimento se repete até que se chegue ao registro que indica o fim do quadro de estados proibidos. Se esse ponto for atingido e ainda houver regras na lista,

é feita a limpeza da mesma e continua-se o procedimento de teste com a próxima regra de testes, caso exista. Para a construção desse diagrama, foram feitas as mesmas considerações citadas na descrição do diagrama anterior.

3.6.5 Considerações Finais

Nesse ponto, os três elementos de teste da camada de planejamento foram desenvolvidos. A utilização desses elementos é ilustrada a seguir utilizando o exemplo da desempilhadeira automatizada apresentado na figura 3.6.

O primeiro exemplo consiste em uma execução em que não foi detectada falha, como mostra a figura 3.11. Nessa figura, a coluna da esquerda representa o arquivo de regras e a da direita o arquivo de registros. Percebe-se que nesse exemplo todas as regras não proibidas se adequaram a registros, em particular as regras sequenciais se adequaram a registros também sequenciais. A regra do quadro proibido que se adequou ao registro não causou falha, pois para que isso ocorresse era necessário que todas as regras do quadro proibído se adequassem a registros entre os estados sequenciais que delimitam sua verificação.

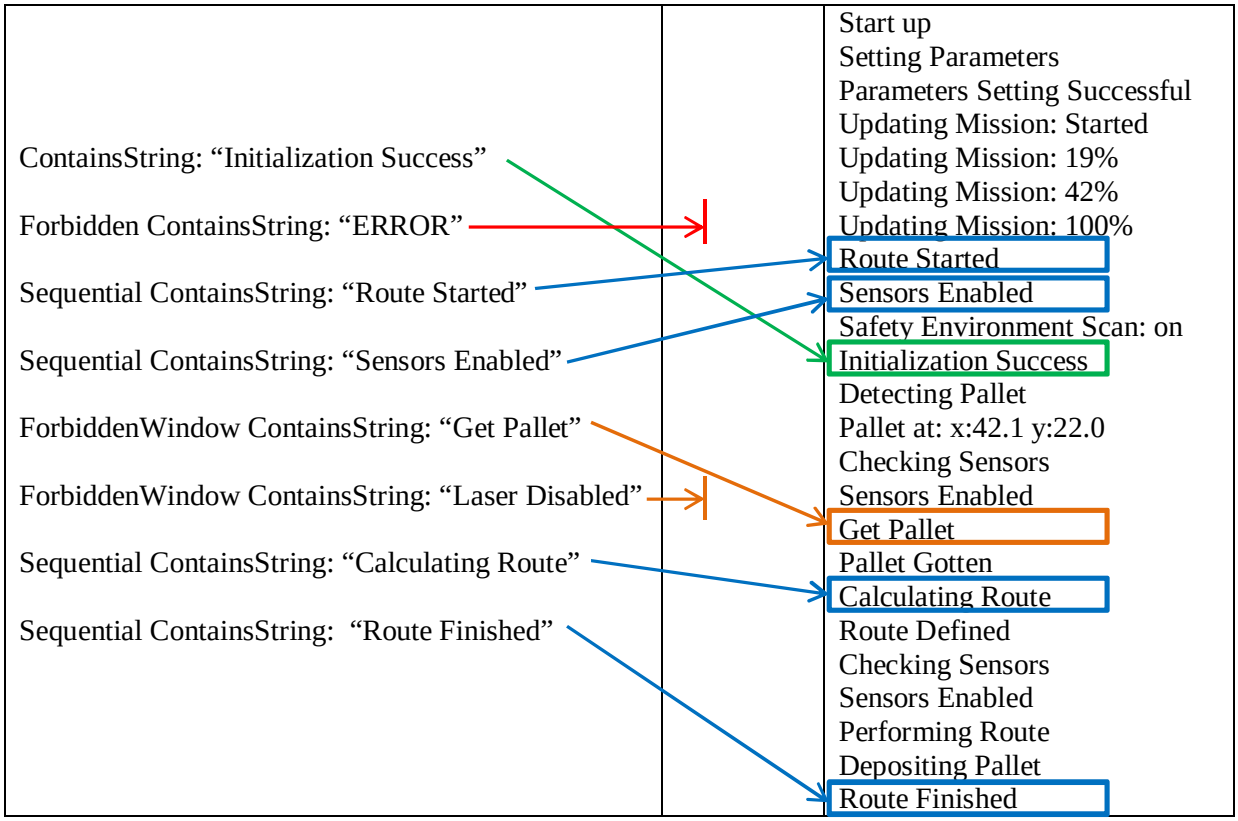

Figura 3.11: Exemplo de Sucesso da Verificação de Falhas

No segundo exemplo, mostrado na figura 3.12, verifica-se uma situação semelhante à anterior, exceto pela ocorrência de uma falha quando se completa o quadro de estados proibidos. É im-

portante reafirmar que a ordem em que as regras do quadro proibido são verificadas no arquivo de registro não é relevante. Nesse exemplo, por algum motivo o sensor de laser foi desabilitado, causando uma situação de risco. O que deve ser corrigido e posteriormente testado em ambiente físico para validação.

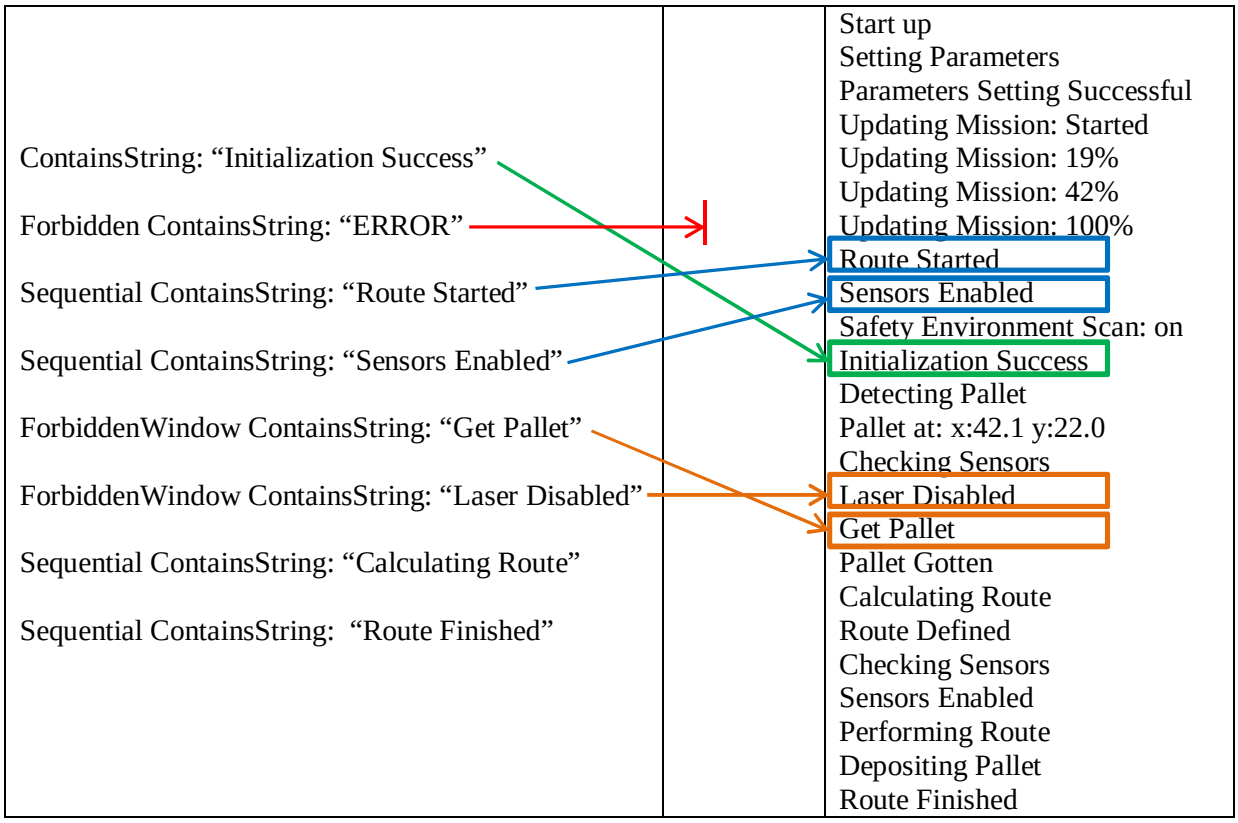

Figura 3.12: Exemplo de Falha Verificada

Os exemplos mostrados são simplificações do processo de verificação que, na prática, apresenta arquivos de registro com milhares de registros além de muitos arquivos de regras para a verificação do mesmo arquivo de registro.

No capítulo 4, a realização do processo de teste será abordada, esclarecendo a combinação desses elementos na prática para a obtenção de resultados de teste.

Capítulo 4

Simulações, Experimentos e Resultados

Percorreu-se até o presente momento toda a apresentação da proposta de abordagem de teste. Nesse capítulo, se examinará um método de definir os estágios da abordagem de teste para sua utilização no teste de sistemas reais, incluindo simulações para determinar a funcionalidade da ferramenta desenvolvida.

Na seção 4.1 é discutida a implementação da metodologia de teste utilizada, incluindo as respectivas ferramentas. Em seguida, os resultados de simulação obtidos, assim como as discussões realizadas são apresentados na seção 4.2.

4.1 Método de Implementação da Abordagem de Teste

Nessa seção são discutidos de forma detalhada todos os estágios de teste mostrados na figura 3.1, que foram utilizados de forma experimental de forma a demonstrar como se pode realizar a metodologia de teste proposta.

4.1.1 Implementação de Funcionalidade

O processo de teste proposto se inicia com a implementação de novas funcionalidades do sistema autônomo, como é mostrado na figura 4.1. Nesse processo, são verificadas as capacidades atuais do sistema autônomo, que se referem às funcionalidades já implementadas. A partir desse dado são definidas as próximas funcionalidades que devem ser desenvolvidas.

Imaginemos o sistema que foi modelado para os testes experimentais, que se trata de um robô móvel que tem como objetivo encontrar e organizar objetos. Nesse robô, as funcionalidades devem ser implementadas em uma determinada ordem para se otimizar o processo de teste. Por exemplo, a funcionalidade de **mover-se até o objeto** deve ser implementada após a implementação da funcionalidade **mover-se**.

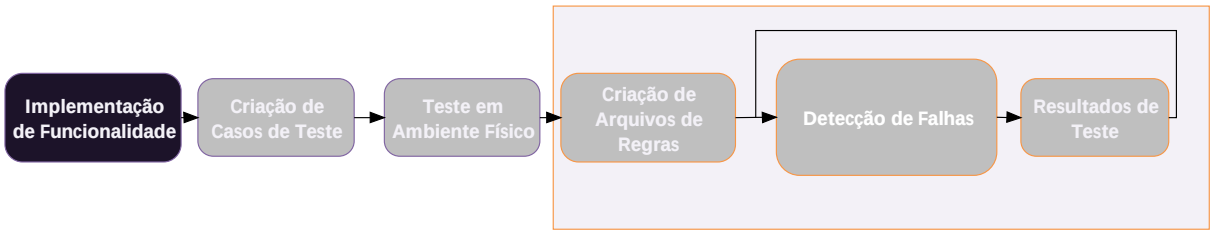

Figura 4.1: Estágio de Implementação de Funcionalidade

Para nosso exemplo foram elaboradas algumas funcionalidades. São elas a **calibração de sensores**, **iniciar tarefa**, **mover-se**, **detectar objeto**, **pegar objeto**, **calcular rota**, **desviar de obstáculos** e **depositar objeto**. Para a realização dos experimentos, é considerado que todas as funcionalidades foram desenvolvidas em uma instância, de modo que se possa descrever mais situações em que o sistema pode falhar.

Considerando as funcionalidades implementadas, passa-se então ao estágio de criação de casos de teste.

4.1.2 Criação de Casos de Teste

Ao se criar uma funcionalidade, é essencial verificar sua manifestação no sistema real. Dessa forma, é necessário criar situações que exijam do sistema a utilização da funcionalidade implementada em operações típicas, nas quais se espera que o sistema atue após seu desenvolvimento.

O estágio de criação de casos de teste, mostrado na figura 4.2, é o estágio na qual o desenvolvimento das situações mencionadas é realizado.

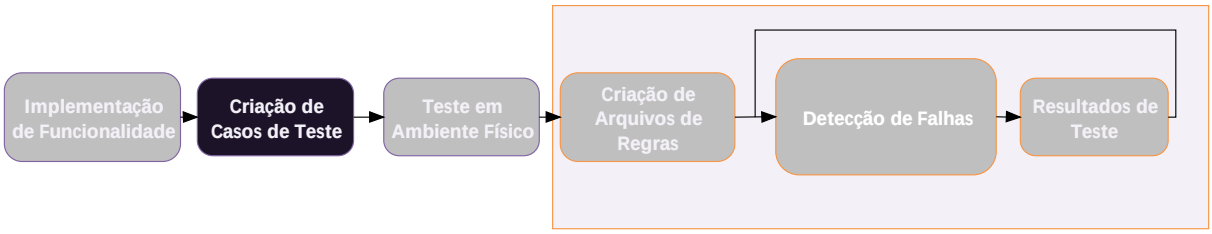

Figura 4.2: Estágio de Criação de Casos de Teste

Os arquivos dos casos de teste desenvolvidos nessa etapa são constituídos de três tipos de elementos: parâmetros internos, parâmetros externos e tarefas. **Parâmetros internos** dizem respeito a configurações do próprio sistema que definem como o sistema vai operar no teste da funcionalidade específica. **Parâmetros externos** são parâmetros que informam ao sistema as características do ambiente em que o sistema opera para o caso de teste. Esse parâmetro se torna muito útil no caso em que sistemas devem operar em vários ambientes distintos, como no exemplo do aspirador de pó robótico, a possibilidade de encontrar vários tipos de piso, tapetes etc. **Tarefas** são informações a respeito dos objetivos que devem ser alcançados na operação do sistema.

Para o sistema que será utilizado na simulação, considerou-se que o sistema já possuía os

parâmetros necessários para a execução do caso de teste. Com isso, foram definidas apenas as tarefas a serem executadas. O caso de teste apresenta as seguintes tarefas:

- Se inicializar corretamente;
- Calibrar sensores;
- Atualizar a tarefa;
- Procurar objeto;
- Detectar objeto;
- Pegar objeto;
- Depositar objeto;
- Desviar de obstáculos;
- Calcular rota;

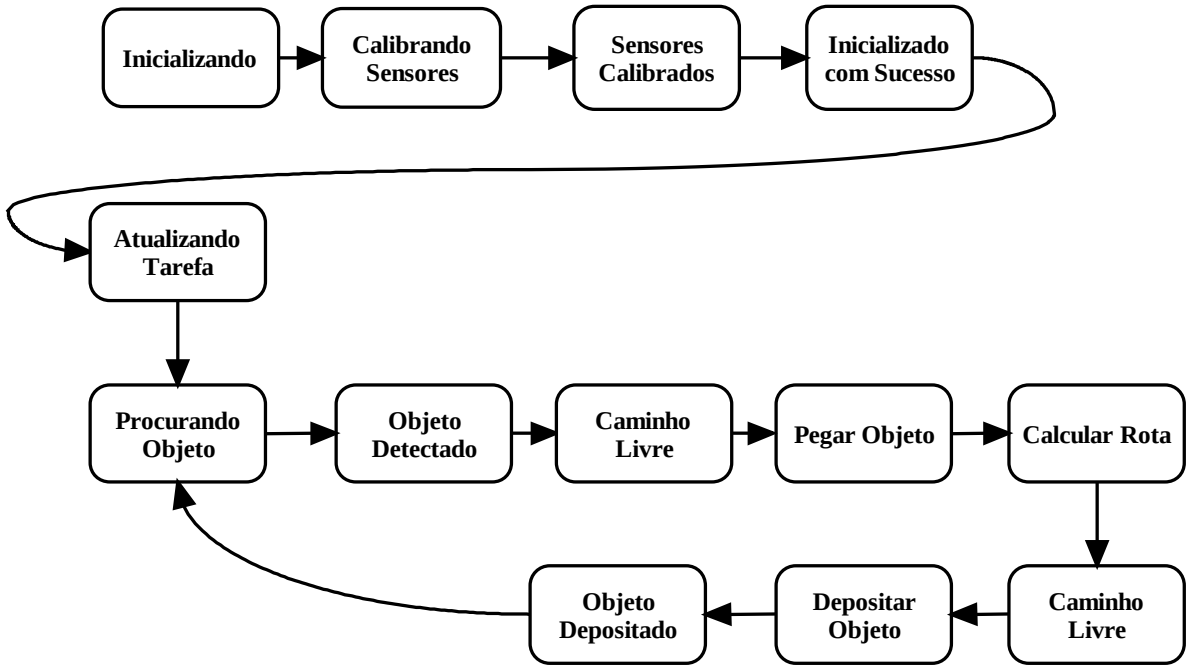

Figura 4.3: Estados Esperados na Execução do Caso de Teste

Com essas definições do caso de teste, é possível prever os estados que descrevem uma execução esperada do sistema. Para o nosso exemplo, esses estados são mostrados na figura 4.3. Nessa figura, percebe-se apenas estados esperados em uma execução sem a ocorrência de falhas do sistema. No caso de uma execução real, é normal a ocorrência de estados adicionais, alguns dos quais caracterizam a ocorrência de uma falha.

Realizada a criação do caso de teste, embarca-se o sistema com as funcionalidades desenvolvidas e procede-se os testes em ambiente físico.

4.1.3 Teste em Ambiente Físico

Uma vez que se tem os casos de teste definidos, inicia-se o estagio de teste em ambiente físico, como mostra a figura 4.4. Esse teste tem como objetivo investigar possíveis estados ou configurações de estados inadequados para a execução das funcionalidades sendo testadas.

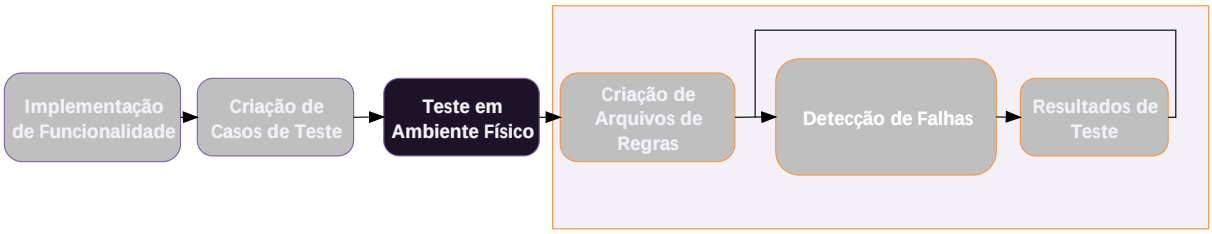

Figura 4.4: Estágio de Teste em Ambiente Físico

Em um teste em ambiente físico, o caso de teste é executado e monitorado, detectando-se os estados que são apropriados para a execução e, por conseguinte, os estados que não são adequados. Por exemplo, um estado onde seja detectado um obstáculo que não seja sucedido por um estado de evitar o obstáculo. Essas ocorrências são observadas em ambiente físico, possibilitando a criação de regras de verificação do comportamento do sistema.

Outra característica relevante do teste em ambiente físico é a gravação de dados de sensor. Os dados de sensor são gravados durante toda a operação do caso de teste. Dessa forma, o caso de teste pode ser emulado posteriormente utilizando dados reais de sensores para a detecção de falhas do sistema. Os dados de sensor são alocados juntamente com os arquivos do caso de teste que foi executado.

No exemplo do robô móvel, não será utilizado o teste em ambiente físico pela indisponibilidade de um sistema real que atenda as especificações. Para contornar esse problema, levando em conta que o objetivo aqui é verificar a funcionalidade da ferramenta de teste desenvolvida, realiza-se a simulação dos estados do sistema como uma máquina de estados finitos. Para realizar a simulação é necessário a utilização dos softwares Matlab [36], Simulink [37] e Stateflow [38].

Como já foi dito acima, o objetivo da simulação é determinar a capacidade da ferramenta de teste na detecção de falhas. Para isso, foram investigados estados que ocasionam falhas do sistema e seriam plausíveis em uma execução do sistema, como pode ser visto na figura 4.5 que é uma complementação da figura 4.3.

Dos estados adicionados na figura 4.5, muitos não acarretam em falhas diretamente, por exemplo, os estados **Nenhum Dado de Sensor**, **Objeto Perdido** e **Obstáculo**. Esses estados podem ocorrer durante a execução de um sistema sem que ocorra uma falha, pois o sistema pode se recuperar desses estados, no primeiro tentando reajustar os sensores, no segundo procurando novamente o objeto e no terceiro desviando do obstáculo. Esses estados podem acarretar em falhas caso não sejam corrigidos. Existem, por outro lado, estados como **Leitura Incorreta** e **Erro** que caracterizam diretamente uma falha do sistema. Essas questões são mais bem discutidas na subseção 4.1.4.

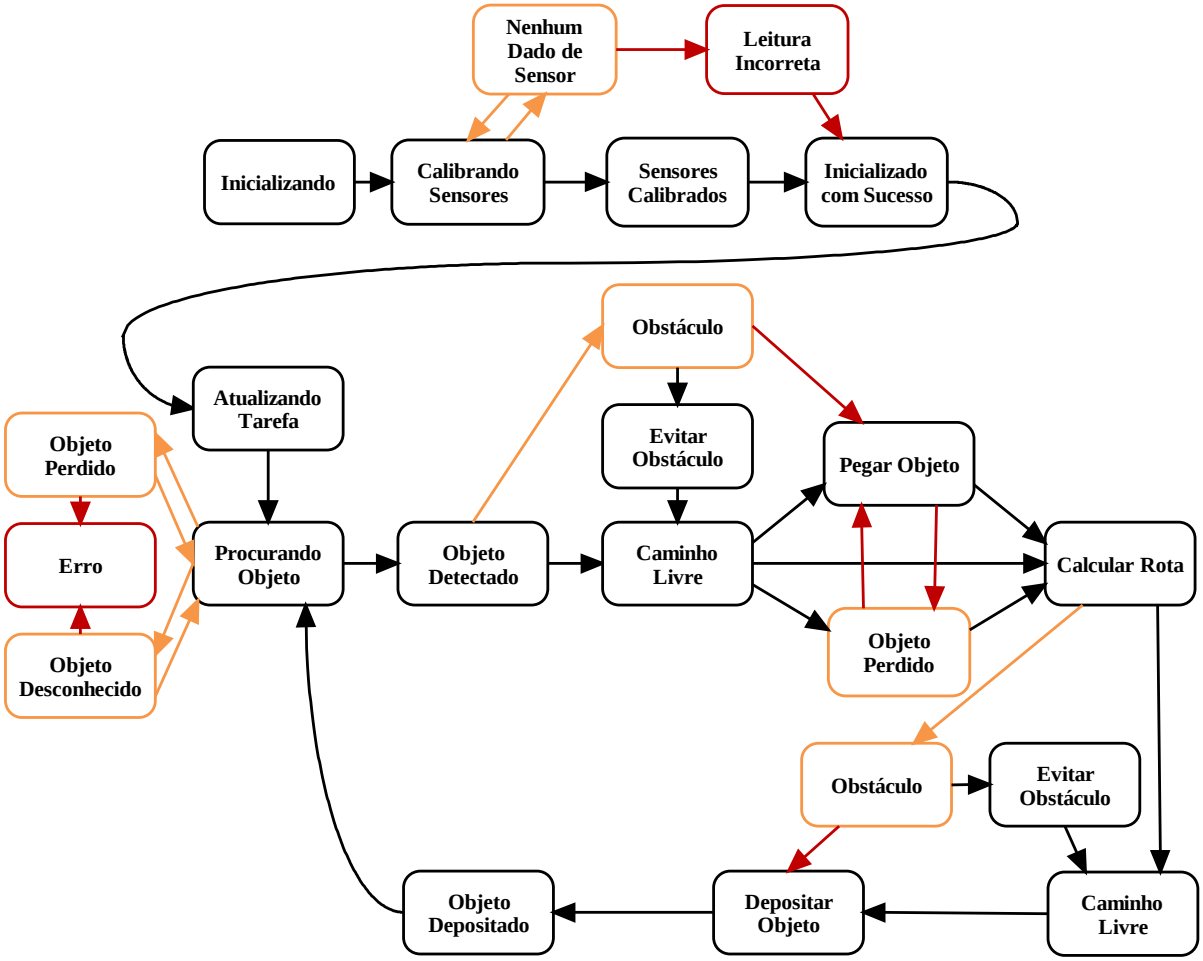

Figura 4.5: Estados na Execução do Caso de Teste Simulado

A simulação foi realizada no Simulink, no qual foi desenvolvido o diagrama mostrado na figura 4.6. Todo o sistema mostrado foi sincronizado para que tivessem o mesmo tempo de amostra. O diagrama apresenta o bloco *Input*, que representa a entrada do sistema. Para a simulação, a transição entre os estados foi modelada de forma probabilística por simplicidade. Assim, a entrada do sistema consiste em uma função que gera números aleatórios entre 0 e 1, que é representada pela variável T. O bloco *Sistema* representa o diagrama de estados da figura 4.5 implementados como estados do Stateflow. O diagrama original da simulação, com as probabilidades de transição, é apresentado no anexo I.

À medida que as entradas do bloco *Sistema* são geradas, os estados internos desse bloco se alteram. Para que se saiba em que estado o sistema se encontra, foi definida uma variável *s*, que representa um índice do estado. Essa variável é a saída do sistema, que é registrada em um arquivo representado pelo bloco *Arquivo de Índices*. Esse arquivo contém os índices de todos os estados pelo qual o sistema passou.

Após a produção do arquivo que contém os índices dos estados é possível gerar o arquivo de registros propriamente dito, com estados escritos de forma mais inteligível e verificável considerando as regras de verificação. Para isso, se utiliza o programa codificado em Matlab, mostrado no anexo

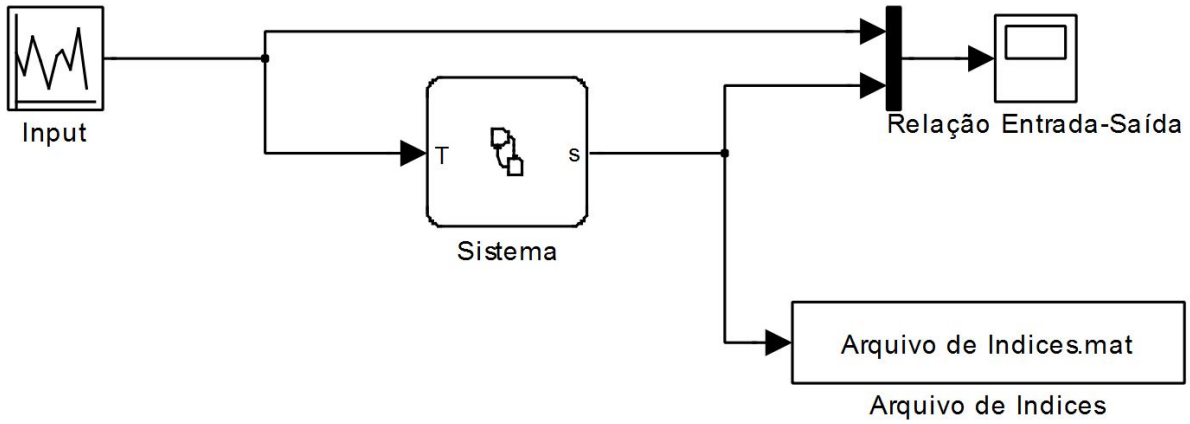

Figura 4.6: Diagrama de Simulação

II. Esse programa lê o arquivo que contém os índices dos estados do sistema e utiliza uma tabela de dispersão, em inglês *hash table*. Os índices são as chaves de pesquisa associadas a valores, que são os nomes dos estados. Com isso, é produzido o arquivo de registros com os nomes dos estados. Um exemplo do resultado é mostrado na figura 4.7.

Arquivo de Índices	Tabela de Dispersão	Arquivo de Registros
90 91 93 91 93 91 92 95 1 2 13 2	1→Atualizando Tarefa 2→Procurando Objeto 3→Objeto Detectado 5→Pegar Objeto 6→Calcular Rota 7→Caminho Livre 8→Obstaculo 9→Evitar Obstaculo 10→Depositar Objeto 11→Objeto Depositado 12→Objeto Perdido 13→Erro 14→Objeto Desconhecido 90→Inicializando Sistema 91→Calibrando Sensores 92→Sensores Calibrados 93→Nenhum Dado de Sensor 94→Leitura Incorreta 95→Inicializado com Sucesso	Inicializando Sistema Calibrando Sensores Nenhum Dado de Sensor Calibrando Sensores Nenhum Dado de Sensor Calibrando Sensores Sensores Calibrados Inicializado com Sucesso Atualizando Tarefa Procurando Objeto Objeto Perdido Procurando Objeto

Figura 4.7: Criação de Arquivo de Registros a partir do Arquivo de Índices

Utilizando a simulação apresentada, são obtidos vários arquivos de registros para a verificação do comportamento. O próximo passo é determinar as regras de verificação, que são objeto da próxima subseção.

4.1.4 Criação de Arquivo de Regras

Regras de verificação são estruturas que possibilitam a identificação de falhas na execução do sistema. Isso ocorre pois o arquivo de regras é gerado a partir das possibilidades de ocorrência de falha descobertas durante os testes em ambiente físico, dependência mostrada na figura 4.8.

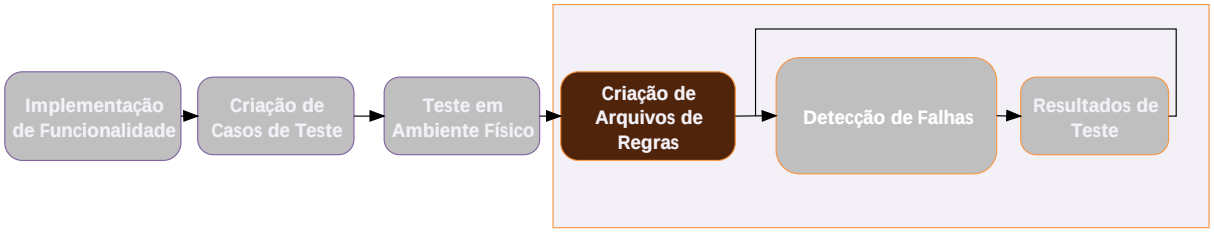

Figura 4.8: Estágio de Criação de Arquivos de Regras

Para facilitar a compreensão da formulação das regras de verificação, tomemos como base os estados mostrados na figura 4.5. O inicio da formulação se dá com as definições de estados não sequenciais. Para isso, deve-se verificar os estados que são desejados (ou proibidos) independentemente da ordem em que apareçam no sistema. Esse tipo de regra pode ser útil para tornar o teste mais eficiente e detectar falhas básicas, haja vista que são as primeiras regras a serem executadas. Pode-se considerar que o estado **Inicializado com Sucesso** seja um estado não sequencial, mesmo que no exemplo ele ocorra em uma sequência. Já o estado **Erro** pode ser considerado um estado não sequencial proibido. Essas considerações se dão devido à simplicidade do modelo, em que se tem apenas um estado por vez sendo processado. Em sistemas reais os módulos podem ser executados simultaneamente e apareceriam estados não sequenciais de fato. As regras para os estados não sequenciais do exemplo são mostrados a seguir.

ContainsString: "Inicializado com Sucesso"
Forbidden ContainsString: "Erro"

As próximas regras a serem definidas são as regras sequenciais. Para gerar as regras sequenciais, deve-se escolher o estado inicial e a partir dele escolher o estado de interesse. Os demais estados são criados por referenciamento ao último estado já criado. Considerando os estados que estão sendo utilizados, pode-se tomar o primeiro estado sequencial como sendo o estado **Calibrando Sensores**. A partir desse estado podem ocorrer das situações: na primeira, a execução passa pelo estado **Sensores Calibrados** e segue para **Inicializado com Sucesso**, enquanto que na segunda tem-se o estado **Nenhum Dado de Sensor**, que pode ocasionar uma falha se for sucedido pelo estado **Leitura Incorreta**, o que pode ser descrito para a verificação por regras sequenciais proibidas entre o estado **Calibrando Sensores** e o estado **Atualizando Tarefa**. A partir desse estado, deve-se passar pelo estado **Procurando Objeto**, do qual tem-se três possiblidades. Os estados **Objeto Perdido** e **Objeto Desconhecido** são situações que podem gerar erro ou voltar ao estado de busca de objeto, não sendo necessário a criação de regras para descrever esses estados, pois a simples regra não sequencial para o estado **Erro** é capaz de verificar a falha. Em seguida, espera-se o estado **Objeto Detectado** em uma determinada posição, que e descrita por uma

regra de registro variável. Esse estado pode ser sucedido por **Obstáculo** ou **Caminho Livre**. No caso do **Obstáculo**, o sistema pode ir para o estado **Evitar Obstáculo** e retornar ao estado **Caminho Livre** ou pode ir para o estado **Pegar Objeto**, gerando uma falha. Os problemas dessa parte são evitados simplesmente pela definição da regra sequencial do estado **Caminho Livre**, que obriga o sistema a passar por esse estado antes de pegar o objeto. Do estado **Caminho Livre** existem três alternativas: **Pegar Objeto**, **Objeto Perdido** e **Calcular Rota**. Esses estados constituem um quadro proibido, na medida em que a passagem do estado **Pegar Objeto** e **Objeto Perdido** constituem uma falha. Regras do quadro proibido para esses dois estados devem ser elaboradas entre o estado **Caminho Livre** e **Calcular Rota**, que constitui o próximo estado sequencial. Desse estado se obtem uma configuração de obstáculo ou caminho livre como ocorrido anteriormente e a solução para essa configuração é a mesma já utilizada, ou seja, uma regra sequencial para o estado **Caminho Livre**. O próximo estado é **Depositar Objeto**, que só chega ao estado **Objeto Depositado**, sendo necessária uma regra sequencial apenas para esse último. Por fim, retorna-se ao estado **Procurando Objeto**, que é a última regra sequencial desse ciclo, o qual pode se repetir dependendo do número de objetos a se tratar. As regras sequenciais são mostradas a seguir.

```
Sequencial ContainsString: "Calibrando Sensores"
Forbidden Sequencial ContainsString: "Nenhum Dado de Sensor"
Forbidden Sequencial ContainsString: "Falha de Leitura"
Sequencial ContainsString: "Atualizando Tarefa"
Sequencial ContainsString: "Procurando Objeto"
Sequencial StringConstraints: "Objeto Detectado: x%f y%f"
Sequencial ContainsString: "Caminho Livre"
ForbiddenWindow ContainsString: "Pegar Objeto"
ForbiddenWindow ContainsString: "Nenhum Dado de Sensor"
Sequencial ContainsString: "Calcular Rota"
Sequencial ContainsString: "Caminho Livre"
Sequencial ContainsString: "Objeto Depositado"
Sequencial ContainsString: "Procurando Objeto"
```

Assim, para o tratamento de um objeto, o arquivo de regras de verificação dos estados mostrados na figura 4.5, contendo regras sequenciais e não sequenciais, além de comentários, pode ser visto como segue.

4.1.5 Detecção de Falhas

Após a criação de arquivos de regras para a verificação de casos de teste, inicia-se o estágio de detecção de falhas, como é mostrado na figura 4.9.

Esse estágio do processo de teste é composto por duas etapas preparatórias e pelo teste das camadas funcional e de planejamento, como é mostrado na figura 4.10. É mostrado nessa subseção como esses elementos podem ser implementados na prática.

Antes de iniciar a discussão a respeitos das partes do sistema de teste, é necessário definir, como

```
#Arquivo de Regras de Verificação do Sistema de Detecção e Organização de Objetos
#Caso de Teste de Número 1
#Regras Não Sequenciais
ContainsString: "Inicializado com Sucesso"
Forbidden ContainsString: "Erro"
#Regras Sequenciais
Sequential ContainsString: "Calibrando Sensores"
Forbidden Sequential ContainsString: "Nenhum Dado de Sensor"
Forbidden Sequential ContainsString: "Leitura Incorreta"
Sequential ContainsString: "Atualizando Tarefa"
Sequential ContainsString: "Procurando Objeto
" Sequential StringConstraints: "Objeto Detectado: x%f y%f"
   Argument 1: [0,0.9]
   Argument 2: [0,0.9]
End
Sequential ContainsString: "Caminho Livre"
ForbiddenWindow ContainsString: "Pegar Objeto"
ForbiddenWindow ContainsString: "Objeto Perdido"
Sequential ContainsString: "Calcular Rota"
Sequential ContainsString: "Caminho Livre"
Sequential ContainsString: "Objeto Depositado"
```

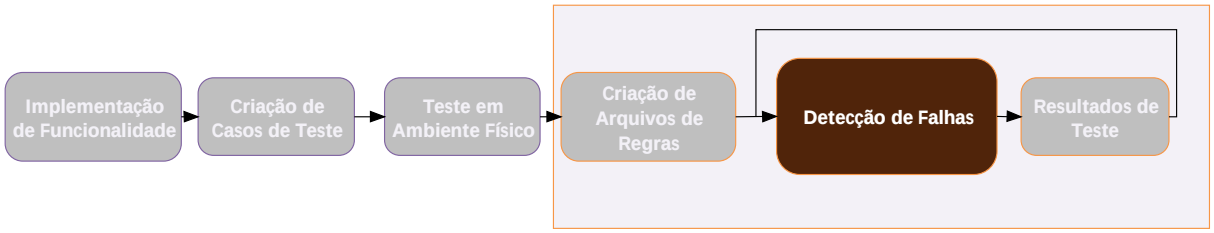

Figura 4.9: Estágio de Detecção de Falhas

foi mostrado na seção 3.3, a estrutura básica utilizada. Como foi dito na mesma seção, as principais características desejadas para a estrutura básica são a possibilidade de anexação e organização de etapas de teste, o monitoramento do processo de teste, a capacidade de armazenar os códigos do sistema e identificar modificações no mesmo e a manutenção de um histórico de testes.

Para atender a essas características, podem ser combinadas duas ferramentas bastante conhecidas na área de testes de sistemas computacionais. Essas ferramentas são descritas a seguir:

- *Subversion*: essa ferramenta consiste de um sistema de controle de versão centralizado. Essa ferramenta mantem em um repositório os arquivos relativos ao sistema autônomo (tanto os códigos quanto documentação e casos de teste) comtemplando a versão atual dos arquivos e o histórico de versões anteriores [31].

- *Jenkins*: programa que provê a integração contínua para o desenvolvimento de sistemas computacionais. Nesse programa, ocorre a anexação de tarefas (chamada *jobs*) que são executadas e monitoradas [30].

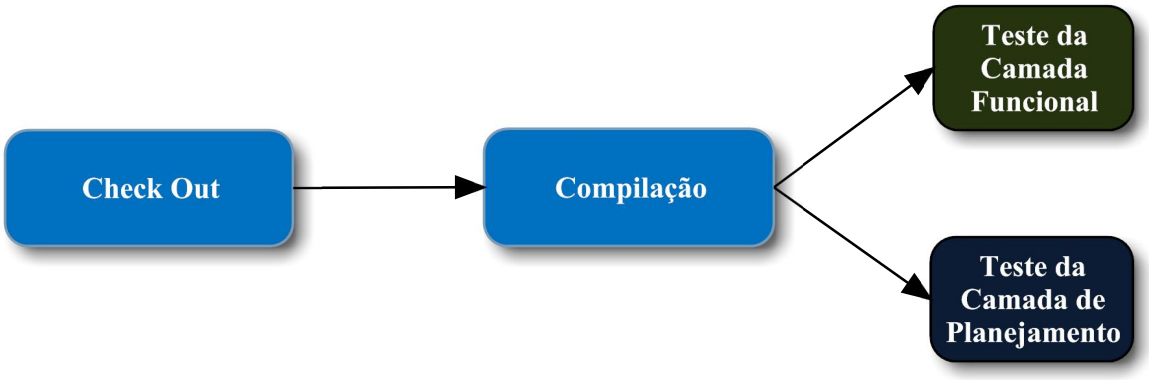

Figura 4.10: Estágio de Detecção de Falhas

A combinação dessas ferramentas traz grandes vantagens ao sistema de teste. Primeiramente porque são integradas, e podem ser utilizadas com simplicidade para a realização das etapas de teste. A primeira etapa preparatória definida foi o *check out*, que consiste em obter os arquivos do repositório do *Subversion*. Nessa etapa, o programa *Jenkins* verifica se ocorreu alguma modificação nos arquivos (o que implica em um novo número para a versão) e caso ocorra, os arquivos são adquiridos e inicia-se o processo de teste. A segunda etapa é a *compilação*, na qual o código do sistema autônomo é transformado em arquivos executáveis com as configurações adequadas para teste.

Essas etapas não foram implementadas na prática durante esse trabalho, devido à falta de um sistema para ser testado, o que não justificaria a criação de um repositório ou mesmo de definições de compilação.

4.1.5.1 Detecção de Falhas da Camada Funcional

O processo de teste da camada funcional é composto essencialmente das etapas de **análise estática**, **análise lógica**, **análise dinâmica**, **teste de desempenho** e **cobertura de código**.

Na etapa de análise estática, é realizada a detecção de erros sintáticos e semânticos, através de informações passadas à ferramenta de teste através de métodos formais. As ferramentas nesse caso dependem essencialmente da linguagem de programação utilizada, já que a análise ocorre diretamente no código, sem que haja de fato a execução do código. Ferramentas que podem ser usadas nessa análise são Lint para C/C++ e Jtest para Java.

Para a análise dinâmica, as falhas verificadas dizem respeito às falhas de processamento do sistema, gerados a partir de relações de entrada e saída. Dados de vulnerabilidade do sistema as entradas são geralmente obtidos com esse tipo de análise. Rational AppScan da IBM, Jtest da Parasoft e Parallel Inspector da Intel são exemplos de ferramentas que aplicam esse tipo de análise.

No caso da análise lógica, o código do sistema é traduzido em uma rede de Petri ou em um Autômato e aplicados a uma ferramenta de checagem de modelos. Essa técnica é utilizada pelas ferramentas CPAchecker da SoSy-Lab e BANDERA da SAnToS.

O teste de desempenho é realizado com a imposição de uma carga sobre o sistema, possibilitando inferir características como a utilização de recursos e o tempo de resposta. Essa técnica é aplicável através das ferramentas Visual Studio Team System da Microsoft, o LoadRunner da Hewlett Parckard e o JMeter da Apache.

Para a cobertura de código, podem ser utilizadas duas ferramentas. A primeira é a ferramenta *Gcovr* [39], uma ferramenta que se destina a gerar os dados de cobertura. Esses dados mostram o número de vezes que as linhas de código foram utilizadas durante a execução do sistema. Os dados gerados por essa ferramenta são armazenados em um relatório que contém referências ao código fonte e índices. A segunda consiste na ferramenta *Cobertura* [40], que utiliza dados do relatório produzido durante a execução e o código fonte para publicar os resultados do teste de cobertura. As informações obtidas a partir dessas ferramentas são réplicas do código fonte com marcações sobre sua utilização, assim como dados estatísticos.

O teste da camada funcional não foi implementado na prática, haja vista que não havia um sistema a ser testado e as ferramentas que realizam suas etapas são bem conhecidas no meio de teste, assim como os resultados que elas proporcionam.

4.1.5.2 Detecção de Falhas da Camada de Planejamento

O teste da camada de planejamento consiste de duas etapas: a **emulação** de casos de teste e a **verificação de comportamento**. Essas etapas são apresentadas de forma gráfica na figura 3.4.

A etapa de **emulação** consiste no processamento do caso de teste utilizando os dados de sensor gravados em meio físico e os arquivos do caso de teste, incluindo os arquivos de parâmetros que definem propriedades do sistema e do ambiente. Com isso, o sistema é processado de forma equivalente a uma execução em ambiente físico. Essa etapa tem como resultado o arquivo de registros.

Em seguida, procede-se a etapa de **verificação de comportamento**. A execução da verificação é realizada pela ferramenta de testes descrita na seção 3.6. Para isso se utiliza os algoritmos apresentados nas figuras 3.8, 3.9 e 3.10 como base para a ferramenta, que é mostrada no anexo II. Para a verificação do comportamento, a ferramenta se utiliza do arquivo de registros e do arquivo de regras de verificação. Essa etapa, assim como a etapa de emulação são mostradas na figura 3.5, que inclui os arquivos utilizados e gerados durante o processo.

A sintaxe desenvolvida na subseção 3.6.3 define as diretrizes de implementação da leitura do arquivo de regras. Cada formato de registro é representado por uma classe de regras no código, enquanto os modificadores são tratados por funções específicas. No caso do arquivo de registros, usa-se uma leitura simples. O resultado da leitura desses arquivos é armazenado em uma lista para a utilização no processo de verificação. O código foi implementado utilizando a linguagem de programação Ruby [41].

Considerando os experimentos realizados, não se fez necessária a implementação da etapa de emulação dos casos de teste, pois a simulação implementada se fazia suficiente para gerar os arquivos de registros que são utilizados na etapa de verificação de comportamento. Nessa etapa, é

utilizado o arquivo de regras gerado na subseção anterior e arquivos de registros obtidos no processo de simulação da subseção 4.1.3. Os resultados da simulação e da verificação de comportamento são mostrados na seção seguinte.

4.1.6 Resultados de Teste

O último estágio do processo de teste consiste no processo de apresentação de resultados de teste. Esse estágio ocorre logo após a detecção de falhas, como mostra a figura 4.11.

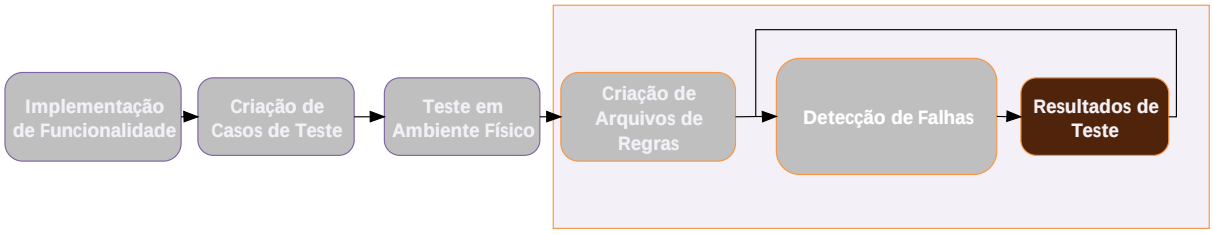

Figura 4.11: Estágio de Resultados de Teste

A apresentação de resultados de teste pode ser realizada através de relatórios de falhas que são enviados aos desenvolvedores do sistema assim que o processo de detecção é completamente executado. A própria ferramenta *Jenkins* possui meios de reportar as informações de erros utilizando-se inclusive de mensagens por correio eletrônico.

Objetiva-se, com isso, informar aos desenvolvedores das falhas existentes no sistema a cada vez que uma modificação do sistema é identificada. Quando um modificador modifica o código do sistema, o processo de detecção de falhas e de apresentação dos resultados de teste se repete para a nova versão do sistema sendo testado. Isso ocorre até que não haja mais falhas no sistema.

4.2 Resultados e Discussão

Nessa seção serão mostrados os resultados provenientes da simulação e da verificação de comportamento utilizando a ferramenta de teste, descritos respectivamente nas subseções 4.1.3 e 4.1.5.

A simulação foi realizada de modo a se obter vários arquivos de registros para o sistema mostrado na figura 4.5. Desses arquivos de registro, foram escolhidos os que melhores representavam as funcionalidades da ferramenta de teste. Além disso, foi realizada a fragmentação do arquivo de registro gerado em simulação, para utilizar somente sua porção de interesse para a análise de forma a tornar a apresentação mais clara.

Para a realização da verificação, utilizou-se a ferramenta de teste desenvolvida na subseção 3.6.4 e o arquivo de regras definido na subseção4.1.4. Esse arquivo de regras se refere à execução do caso de teste abordado na subseção 4.1.2.

No primeiro resultado, mostrado na figura 4.12, percebe-se uma verificação bem sucedida. Nesse caso, todas as regras positivas (entenda-se regras não proibidas) foram verificadas, especialmente as

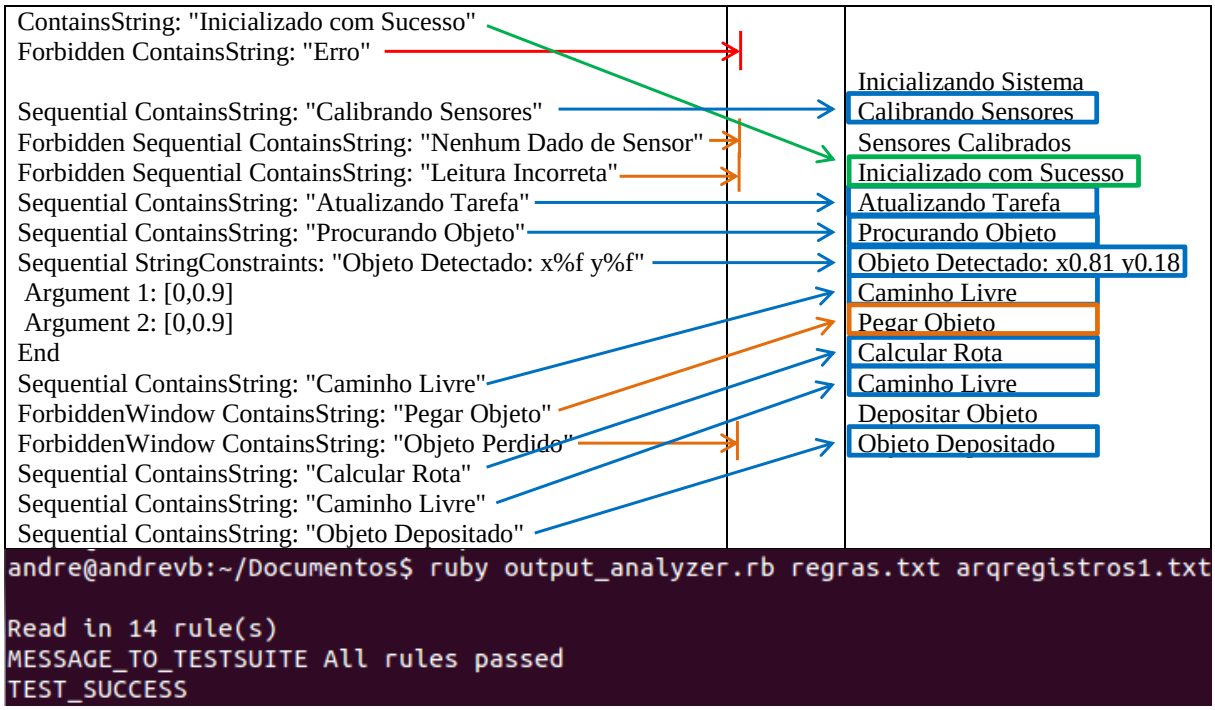

Figura 4.12: Verificação de Comportamento bem Sucedida

regras sequenciais que foram verificadas na ordem adequada. Houve uma regra do quadro proibido que estava presente no arquivo de registros, mas não foi preenchido o quadro, não representando uma falha.

Da figura 4.12, infere-se o comportamento do sistema durante a execução do caso de teste. A ferramenta de teste foi capaz de verificar os estados do sistema durante o processamento desse caso de teste, desde a inicialização do sistema à deposição do objeto.

Como a ferramenta foi desenvolvida para detectar falhas, os resultados foram selecionados de forma a mostrar como se procede a verificação realizada pela ferramenta, ao se deparar com estados que não seguem as regras de verificação.

O resultado mostrado na figura 4.13 representa uma falha em que a primeira regra do quadro de regras não é verificada no arquivo de registros. A regra em questão é uma regra não sequencial que se refere ao estado **Inicializado com Sucesso**. Mesmo que no diagrama 4.5 esse estado se apresente em uma sequência, muitas vezes no desenvolvimento esse tipo de estado pode aparecer em diferentes posições do arquivo de registros. Isso ocorre devido à divisão do sistema em módulos em seu desenvolvimento, fazendo com que muitos módulos sejam independentes e sejam processados em paralelo. Assim, as regras não sequenciais se mostram eficazes ao testar estados específicos que podem representar a confiabilidade de um módulo do sistema que não seja o objeto principal da verificação. Como, por exemplo, um módulo apenas para a inicialização com elevada confiabilidade pode ser validado por um estado não sequencial durante o teste de outro módulo, como o módulo de detecção de imagens.

Pode-se perceber que a regra não sequencial também torna a verificação mais eficiente, já que esse tipo de regra se aplica a estados imprescindíveis para a execução que são verificados anteri-

ContainsString: "Inicializado com Sucesso"		
Forbidden ContainsString: "Erro"		
		Inicializando Sistema
Sequential ContainsString: "Calibrando Sensores"		Calibrando Sensores
Forbidden Sequential ContainsString: "Nenhum Dado de Sensor"		Nenhum Dado de Sensor
Forbidden Sequential ContainsString: "Leitura Incorreta"		Calibrando Sensores
Sequential ContainsString: "Atualizando Tarefa"		Nenhum Dado de Sensor
Sequential ContainsString: "Procurando Objeto"		Calibrando Sensores
Sequential StringConstraints: "Objeto Detectado: x%f y%f"		Nenhum Dado de Sensor
Argument 1: [0,0.9]		Calibrando Sensores
Argument 2: [0,0.9]		Nenhum Dado de Sensor
End		Calibrando Sensores
Sequential ContainsString: "Caminho Livre"		Nenhum Dado de Sensor
ForbiddenWindow ContainsString: "Pegar Objeto"		Calibrando Sensores
ForbiddenWindow ContainsString: "Objeto Perdido"		Nenhum Dado de Sensor
Sequential ContainsString: "Calcular Rota"		Calibrando Sensores
Sequential ContainsString: "Caminho Livre"		Nenhum Dado de Sensor
Sequential ContainsString: "Objeto Depositado"		

```
andre@andrevb:~/Documentos$ ruby output_analyzer.rb regras.txt arqregistros2.txt

Read in 14 rule(s)
MESSAGE_TO_TESTSUITE Positive rule failed
Rule No. 1:'ContainsString: "Inicializado com Sucesso"'
could not be passed
no message!
TEST FAILURE
```

Figura 4.13: Verificação de Comportamento com Falha de Estado Não Sequencial

ormente aos demais estados. No caso de falha desse tipo de estado, a verificação é interrompida imediatamente.

Observando-se o arquivo de registros na figura 4.13, verifica-se uma situação que se assemelha a um *livelock*. Na ocorrência desse tipo de comportamento e assim como de *deadlock*, não se atinge os estados de interesse e a ferramenta é capaz de detectar a falha, que nesse caso pode não ser notada facilmente durante a execução do sistema.

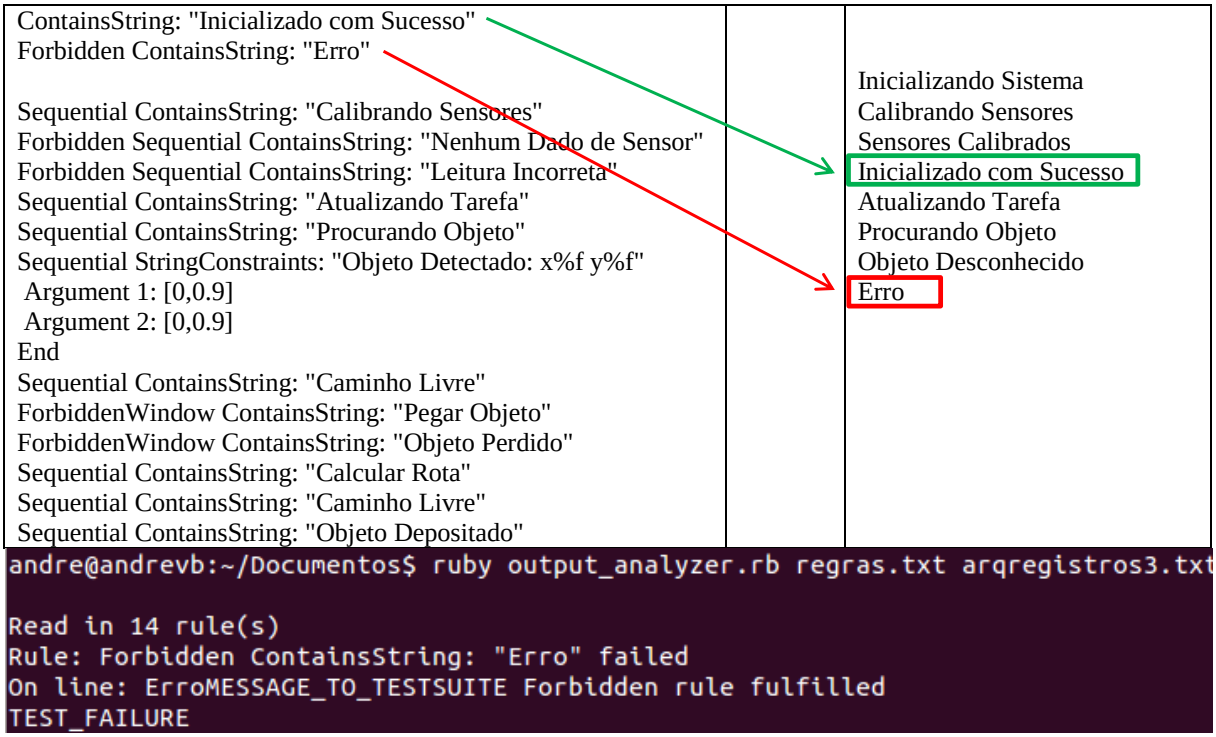

Figura 4.14: Verificação de Comportamento com Falha de Estado Não Sequencial Proibido

É apresentado na figura 4.14 o resultado de uma verificação na qual é caracterizado um erro devido à presença de um estado não sequencial proibido no arquivo de registros. Esse estado representa um *deadlock* do sistema. As regras não sequenciais proibidas são regras para a verificação de estados que são críticos para o sistema e acarretam em falha sempre que ocorrem. A presença desse tipo de estado é rapidamente verificada pela ferramenta.

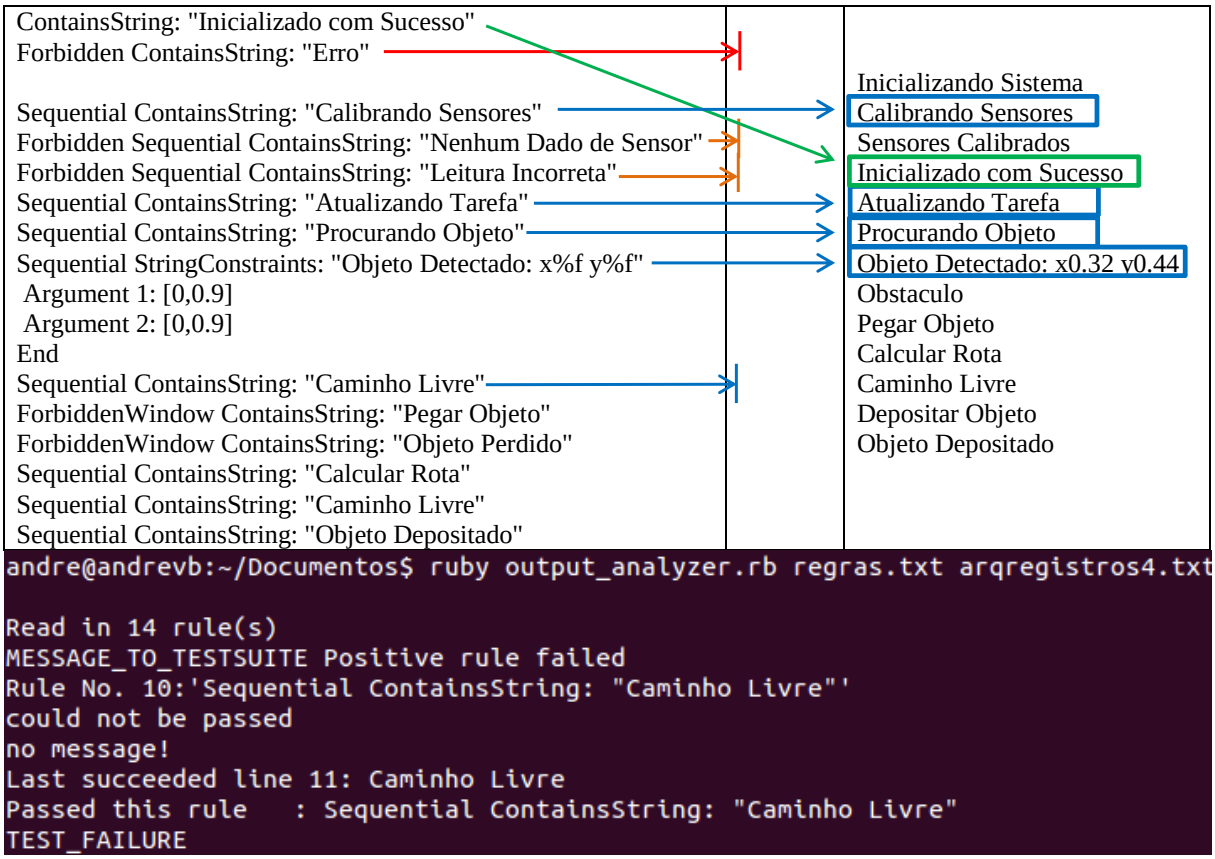

Figura 4.15: Verificação de Comportamento com Falha de Estado Sequencial

Inicia-se a partir desse ponto a discussão a respeito de estados sequenciais. O primeiro resultado relacionado a esse tipo de estado é mostrado na figura 4.16. A falha mostrada nessa figura se refere à não verificação de uma regra sequencial. Isso é perceptível no arquivo de registros, no qual, após a detecção do objeto, é esperado que haja o estado **Caminho Livre** antes do estado **Pegar Objeto**. A não ocorrência do estado **Caminho Livre** implica em uma falha, o que é compreensível intuitivamente, já que ao pegar o objeto o sistema pode colidir. A ferramenta de teste verifica a ausência desse estado e relata a falha como é mostrado.

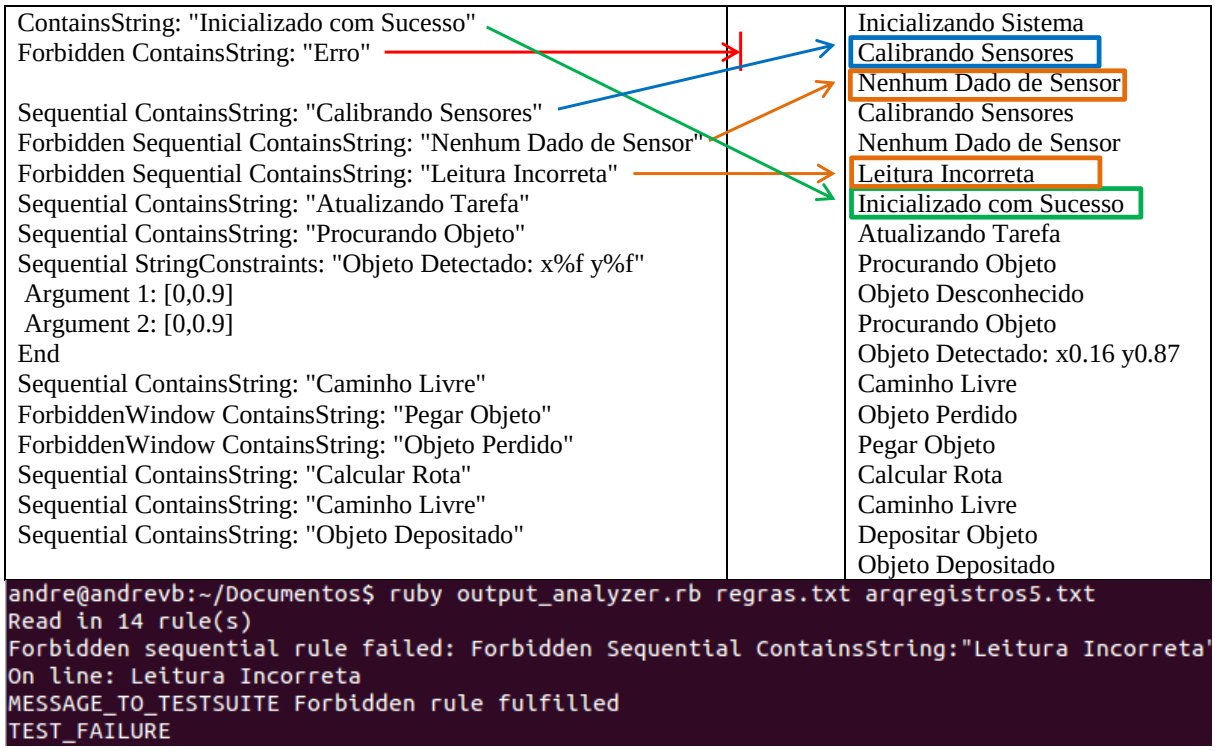

Figura 4.16: Verificação de Comportamento com Falha de Estado Sequencial Proibido

Na figura 4.16, tem-se uma falha causada por um quadro de estados sequenciais proibidos presentes no arquivo de registros. A ocorrência se deu no processo de calibração dos sensores do sistema, no qual houve o estado **Nenhum Dado de Sensor** seguido do estado **Leitura Incorreta**. Depois desses estados, o sistema continua a execução com os sensores calibrados inadequadamente, o que representa uma falha de operação. Com a definição das regras sequenciais proibidas, a ferramenta pode detectar configurações em que o sistema inicia uma falha em um ponto e, mesmo possuindo a capacidade de se recuperar, resulta em um estado que caracteriza falha.

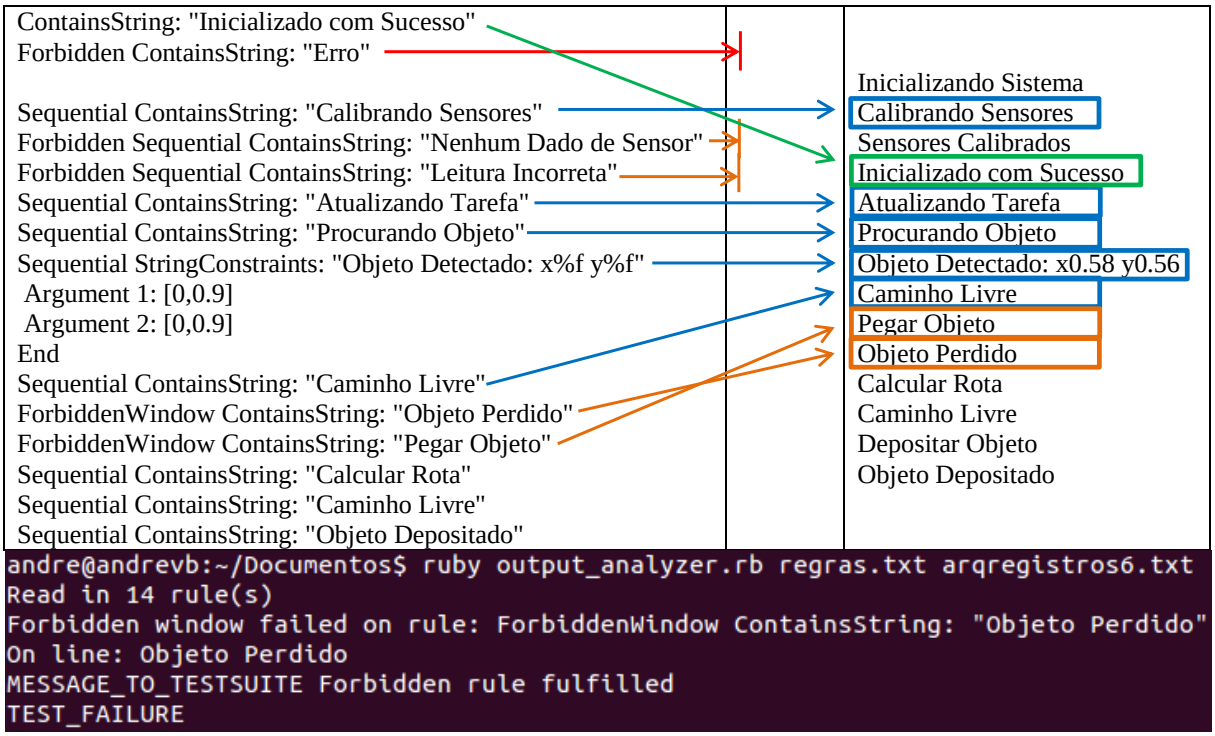

Figura 4.17: Verificação de Comportamento com Falha do Quadro de Estados Proibidos

O resultado apresentado na figura 4.17 implica em uma falha devido a um quadro de estados proibidos que se fez presente no arquivo de registros. Os estados presentes nesse arquivo de registros segue a execução apropriada inicialmente, passando pela calibração de sensores, inicialização e detecção de objeto. Após a detecção, tem-se o estado **Caminho Livre**, indicando que o sistema pode pegar o objeto sem risco de colisão. O que ocorre em seguida é que o sistema inicia o processo para pegar o objeto e perde as informações sobre o objeto, acarretando em riscos que ocorrem se o manipulador do sistema não convergir ao objeto. Por isso, se caracteriza uma falha, a qual dificilmente seria percebida pelo comportamento do sistema real, mas é facilmente identificada pela ferramenta de testes.

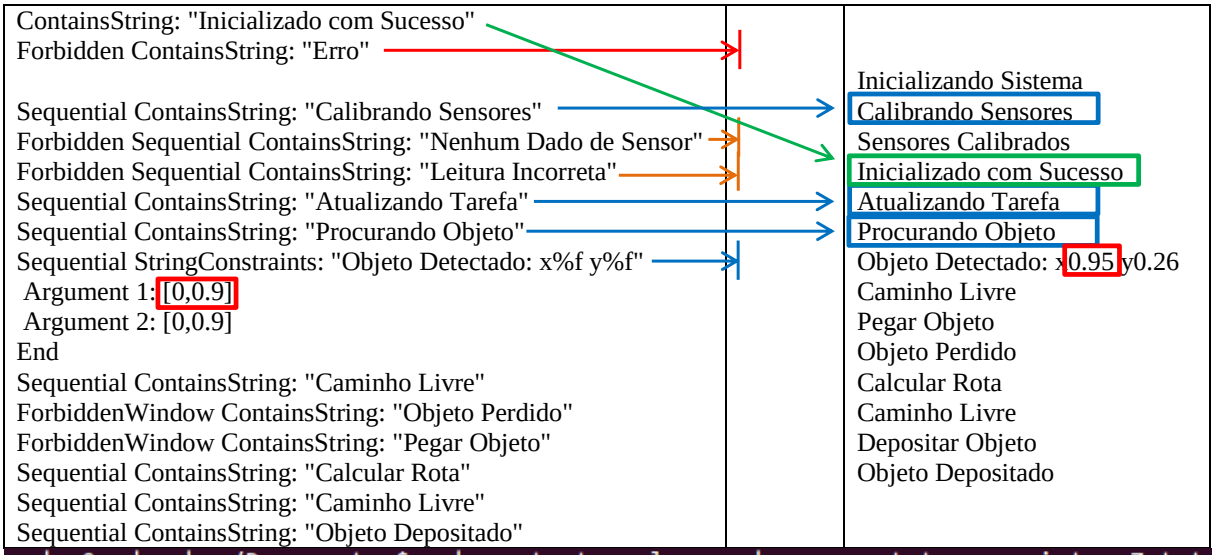

Figura 4.18: Verificação de Comportamento com Falha de Estado Variável

É interessante abordar como a ferramenta trata as falhas causadas por estados descritos por registros de formato variável. Na figura 4.18 é apresentado um resultado onde ocorre uma falha devido a um registro de formato variável que possui variáveis fora dos limites aceitáveis. O estado **Objeto Detectado: x0.95 y0.26** possui duas variáveis que são as posições de um sistema bidimensional. As restrições de valores de ambas são 0 como valor mínimo e 0.9 como valor máximo. Como a primeira variável apresenta valor superior ao valor máximo permitido, tem-se a caracterização de uma falha do sistema. A ferramenta de testes é apropriada para determinar a presença de variáveis que assumem valores inadequados e podem por em risco o comportamento e inclusive a integridade do sistema. Por exemplo, em sistema que possuem monitoramento de parâmetros elétricos, o nível de corrente elevado pode danificar os atuadores ou provocar atuações indesejadas.

Apesar de as funcionalidades da ferramenta de teste se mostrar capazes de verificar vários tipos de estados, podem existir situações com as quais a atual ferramenta de testes não está preparada para lidar. Um exemplo disso é quando estados de interesse se repetem durante o procedimento de teste. Pode-se imaginar um laço em que alguns estados são obtidos várias vezes. Nesse caso, mesmo que haja uma regra para identificar um estado de um laço, a ferramenta não detectaria a ausência desse estado, pois no próximo ciclo seria detectado um estado idêntico. Essa situação pode ser tratada internamente no sistema (por exemplo, adicionando um índice para cada ciclo, tornando os estados únicos) ou na própria ferramenta, implementando métodos que possam lidar

com esse comportamento (por exemplo, verificar mais de um estado sequencial ao mesmo tempo e tratar as lacunas de estados).

Um ponto que não depende da ferramenta, mas sim dos arquivos de regras, deve ser abordado. Como já foi dito, existem sistemas que possuem vários módulos, muitos deles independentes entre si. Durante o desenvolvimento de um sistema autônomo, esses módulos são modificados tanto para a correção de falhas quanto para atender a outros requisitos, como eficiência. Decorre disso que, sendo os módulos independentes, os estados de um módulo não necessariamente mantém a ordem de execução em relação a estados de outro módulo após uma modificação. Por isso, nesses casos deve-se optar pela criação de vários arquivos de registros que desvinculem as regras de verificação de módulos independentes. Com esse procedimento, pode-se evitar falsos positivos. Para ficar mais claro, considera-se a existência de dois módulos independentes. O módulo **Letras** possui os estados **A**, **B** e **C**, enquanto o estado **Números** os estados **1**, **2** e **3**. A figura 4.19 mostra uma falha detectada com a modificação desses módulos, que causou uma inversão nos estados. Isso evidencia um falso positivo da verificação, que é solucionável pela separação das regras em dois arquivos de regras de verificação.

Regras	Estados Originais	Estados Posteriores
Sequential ContainsString: “A”	**A**	**A**
Sequential ContainsString: “1”	**1**	**B**
Sequential ContainsString: “B”	**B**	**1**
Sequential ContainsString: “2”	**2**	**2**
Sequential ContainsString: “C”	**C**	**C**
Sequential ContainsString: “3”	**3**	**3**

Figura 4.19: Falha Causada por Inversão de Estados

Capítulo 5

Conclusões

5.1 Considerações Finais

O trabalho apresentado desenvolveu uma proposta de abordagem sistemática de teste para sistemas autônomos, a qual é potencialmente adequada para teste de sistemas autônomos para diversos propósitos. Os procedimentos de teste são realizados a partir da distinção das camadas funcional e de planejamento presentes nesses sistemas autônomos. Dessa forma, os resultados são alcançados através de testes específicos para cada camada permitindo analisar o sistema com foco sobre a validação e a avaliação do mesmo.

Para a camada funcional definiu-se métodos de teste tradicionais para teste de sistemas computacionais, a saber: a análise estática, a análise lógica, a análise dinâmica, a cobertura de código e o teste de desempenho. Muitos dos quais não puderam ser implementados e validados na prática. No caso da camada de planejamento, foco principal do trabalho, desenvolveu-se uma metodologia de teste que consiste de dois procedimentos: a emulação dos casos de teste e a verificação do comportamento. A emulação de testes de caso tem como objetivo produzir um arquivo de registros com o comportamento do sistema autônomo para a execução de um determinado caso de teste, ao passo que a verificação do comportamento usa os dados resultantes da emulação em conjunto com um arquivo de regras para detectar a ocorrência de falhas. A metodologia de testes desenvolvida se mostrou dirigida a funcionalidades, na medida em que casos de teste são criados para a validação de uma funcionalidade específica em diferentes ambientes.

A ferramenta de teste desenvolvida para a verificação de comportamento da camada de planejamento teve suas funcionalidades validadas através de testes em simulação. Com isso, a ocorrência de falhas pode ser verificada para todas as categorias de estados e formatos de registros. A existência de falhas não detectáveis pela ferramenta ainda pode ocorrer ocorrido devido a formulações inadequadas no arquivo de regras para análise ou por características da falha cuja detecção ainda não foi implementada na ferramenta de testes.

Ainda é necessária a implementação completa da metodologia desenvolvida, além de sua utilização com sistemas reais, para que se possa avaliar a efetividade do processo de teste de ambas as camadas.

Foram verificadas proposições de abordagem de teste de sistemas autônomos em um amplo contexto. Entretanto, o desenvolvimento de uma metodologia com uma abordagem sistemática de teste de sistemas autônomos que utilizasse os conceitos de teste por camadas não havia sido explorada na literatura.

5.2 Perspectivas Futuras

Feitas a definição e a descrição da abordagem de teste, se faz necessária sua implementação prática, assim como a utilização das abordagens de teste em sistemas reais. A partir dessa análise, pode-se verificar a efetividade de todas as etapas aqui apresentadas na detecção de falhas, assim como as carências do sistema.

É interessante a investigação a respeito da incorporação de novas técnicas de teste que possam ser úteis tanto para testes de desenvolvimento quanto para testes de operação. Essa investigação seria uma extensão da abordagem desse trabalho a diversos propósitos de teste, como verificações de adaptabilidade do sistema autônomo a diversos ambientes.

Em se tratando da camada de planejamento, a metodologia de testes avaliou apenas dados de entrada e saída do sistema autônomo, não se preocupando com fatores internos. Os próximos trabalhos devem se preocupar em definir estratégias para verificar informações relevantes sobre a interação entre os módulos do sistema autônomo. Como por exemplo, funcionalidades que compartilham recursos.

Uma das ferramentas que tem capacidade de lidar com a validação de sistemas autônomos é a geração de casos de teste. Trabalhos futuros devem estudar a incorporação da geração de casos de teste na abordagem proposta.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] PCWORLD. *Review: Epson WorkForce WF-2540 All-in-One Printer*. `http://www.pcworld.com/article/2023323/review-epson-workforce-wf-2540-all-in-one-printer.html`. Acessado em 5 de abril de 2013.

[2] TECHDIGEST. *Like Japanese food? Get the Motoman industrial robot to cook it for you*. `http://www.techdigest.tv/2008/11/like_japanese_f.html`. Acessado em 5 de abril de 2013.

[3] MACIAS, F. The test and evaluation of unmanned and autonomous systems. *ITEA Journal*, v. 29, p. 388–395, 2008.

[4] CASTILLO-EFFEN, M.; VISNEVSKI, N.; SUBBU, R. Modeling and simulation for unmanned and autonomous system test and evaluation. *PerMIS '09*, p. 86–92, 2009.

[5] MIKAELIAN, T. *An Integrated Real Options Framework for Model-based Identification and Valuation of Options under Uncertainty*. USA: Massachusetts Institute of Technology, 2009.

[6] CARVALHO, A. L. T. *Autonomous Systems Testing in Practice: a Systematic Approach*. France: Université de Strasbourg, 2012.

[7] IROBOT. *Will Roomba fall down steps or stairs?* `http://uksupport.irobot.com/app/answers/detail/a_id/825/~/will-roomba-fall-down-steps-or-stairs%3F/?cc=uk`. Acessado em 29 de março de 2013.

[8] HAWKINSON, J. *Guidelines for creation, selection, and registration of an Autonomous System (AS)*. `http://tools.ietf.org/html/rfc1930`. Acessado em 29 de março de 2013.

[9] BRAHA, D.; MINAI, A. A.; BAR-YAM, Y. *Complex Engineered Systems*. Berlin, Alemanha: Springer, 2006.

[10] KIRSHBAUM, D. *Complex Adaptive Systems*. `http://www.calresco.org/lucas/cas.htm`. Acessado em 12 de março de 2013.

[11] GOLDSTEIN, J. Emergence as a construct: History and issues. *Emergence : a Journal of Complexity Issues in Organizations and Manadgement. The New England Complex Systems Institute*, v. 1, n. 1, p. 49–72, 1999.

[12] COPELAND, L. *A Practitioner's Guide to Software Test Design*. Boston, USA: Artech House, 2004.

[13] WICHMANN, B. et al. Industrial perspective on static analysis. *Software Engineering Journal*, v. 10, n. 2, p. 69–75, 1995. ISSN 0268-6961.

[14] BALL, T. The concept of dynamic analysis. In: *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. London, UK, UK: Springer-Verlag, 1999. (ESEC/FSE-7), p. 216–234. ISBN 3-540-66538-2. Disponível em: <http://dl.acm.org/citation.cfm?id=318773.318944>.

[15] MOLYNEAUX, I. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. Sebastopol, USA: O'Reilly Media, 2009.

[16] CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 8, n. 2, p. 244–263, abr. 1986. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/5397.5399>.

[17] CORNETT, S. *Code Coverage Analysis*. `http://www.bullseye.com/coverage.html`. Acessado em 31 de março 2013.

[18] DEPARTMENT OF DEFENSE. *Interim Defense Acquisition Guidebook*. Arlington Countyl, USA: DoD, 2009.

[19] DEPARTMENT OF DEFENSE. *Modeling and Simulation (M&S) Verification, Validation, and Accreditation (VV&A)*. Arlington Countyl, USA: DoD Instruction 5000.61, 2003.

[20] DEFENSE AQUISITION UNIVERSITY. *Glossary of Defense Acquisition Acronyms and Terms*. 12th edition. ed. Arlington Countyl, USA: Defense Aquisition University Press, 2005.

[21] BRAT, G.; JONSSON, A. Challenges in verification and validation of autonomous systems for space exploration. In: *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*. Mountain View, USA: USRA/RIACS, 2005. v. 5, p. 2909 – 2914 vol. 5.

[22] PENTA, M. D. et al. Search-based testing of service level agreements. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2007. (GECCO '07), p. 1090–1097. ISBN 978-1-59593-697-4. Disponível em: <http://doi.acm.org/10.1145/1276958.1277174>.

[23] FERREIRA, S. et al. Unmanned and autonomous systems of systems test and evaluation: Challenges and opportunities. *5th IEEE Systems Conference*, 2010.

[24] NESNA, I. A. et al. CLARAty: an architecture for reusable robotic software. In: Gerhart, G. R.; Shoemaker, C. M.; Gage, D. W. (Ed.). *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*. New York, NY, USA: ACM, 2003. (Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, v. 5083), p. 253–264.

[25] BINDER, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston, USA: Addison-Wesley, 1999.

[26] NGUYEN, C.; PERINI, A.; TONELLA, P. Constraint-based evolutionary testing of autonomous distributed systems. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*. New York, NY, USA: ACM, 2008. p. 221–230.

[27] DJANG, P. A.; LOPEZ, F. Unmanned and autonomous systems mission based test and evaluation. In: *Proceedings of the 9th Workshop on Performance Metrics for Intelligent Systems*. New York, NY, USA: ACM, 2009. (PerMIS '09), p. 81–85. ISBN 978-1-60558-747-9. Disponível em: <http://doi.acm.org/10.1145/1865909.1865926>.

[28] IPA, F. *Care-O-bot Home - Fraunhofer IPA*. `http://www.care-o-bot.de/english/Care-O-bot_3.php`. Acessado em 31 de março de 2013.

[29] ORACLE. *Hudson Continuous Integration*. `http://hudson-ci.org/`. Acessado em 1 de março de 2013.

[30] JENKINS. *Meet Jenkins*. `https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins`. Acessado em 1 de março de 2013.

[31] APACHE. *Apache Subversion :Enterprise-class centralized version control for the masses*. `http://subversion.apache.org/`. Acessado em 1 de março de 2013.

[32] TEAM, C. *CVS - Concurrent Versions System*. `http://cvs.nongnu.org/`. Acessado em 31 de março de 2013.

[33] VENET, A.; BRAT, G. *Static Program Analysis using Abstract Interpretation*. `http://ti.arc.nasa.gov/m/tech/rse/publications/papers/cglobalsurveyor/abs_int_tutorial.ppt`. Acessado em 14 de abril de 2013.

[34] MATHWORKS. *PolySpace*. `http://www.mathworks.com/products/polyspace/`. Acessado em 31 de março de 2013.

[35] COVERITY. *Coverity Development Testing*. `http://www.coverity.com/`. Acessado em 31 de março de 2013.

[36] MATHWORKS. *MATLAB - The Language of Technical Computing*. `http://www.mathworks.com/products/matlab/`. Acessado em 17 de abril de 2013.

[37] MATHWORKS. *Simulink - Simulation and Model-Based Design*. `http://www.mathworks.com/products/simulink/`. Acessado em 17 de abril de 2013.

[38] MATHWORKS. *Stateflow - Model and simulate decision logic using state machines and flow charts*. `http://www.mathworks.com/products/stateflow/`. Acessado em 17 de abril de 2013.

[39] CORPORATION, S. *gcovr - FAST*. `https://software.sandia.gov/trac/fast/wiki/gcovr`. Acessado em 1 de abril de 2013.

[40] APACHE. *Cobertura*. `http://cobertura.sourceforge.net/`. Acessado em 1 de abril de 2013.

[41] MATSUMOTO, Y. *Ruby Programming Language*. `http://www.ruby-lang.org/`. Acessado em 1 de abril de 2013.

ANEXOS

I. DIAGRAMAS ESQUEMÁTICOS

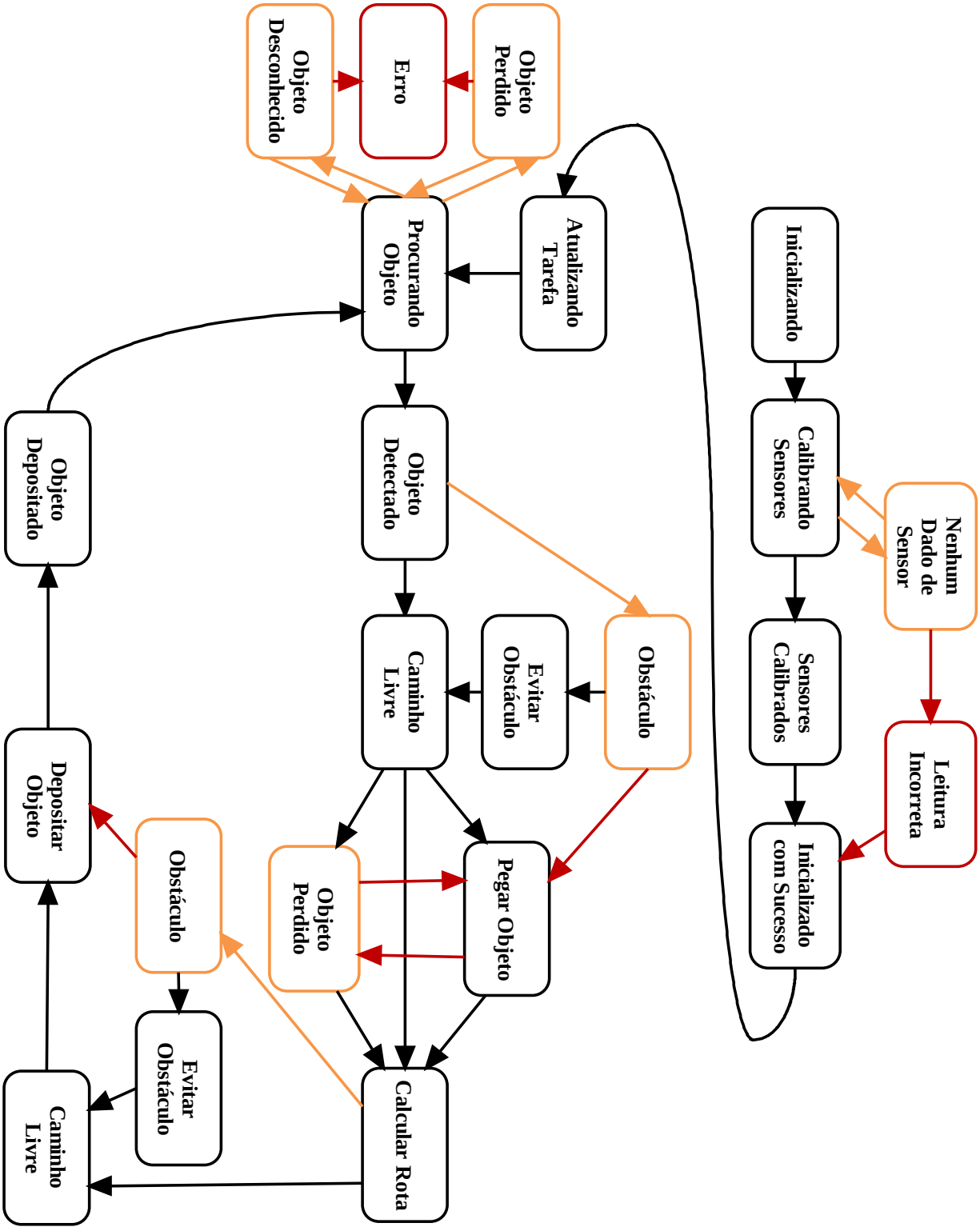

Figura I.1: Diagrama de Estados do Sistema para Simulação

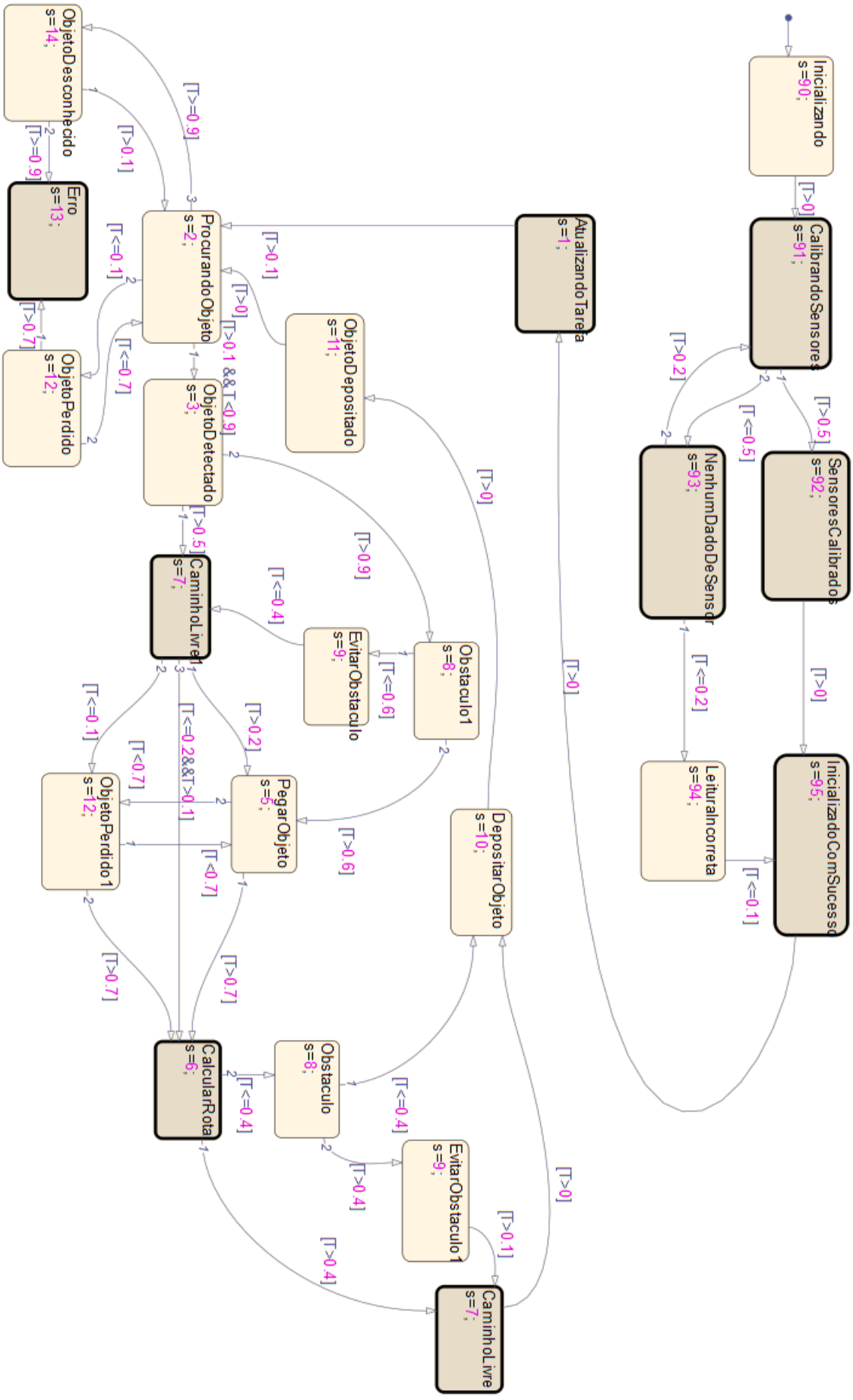

Figura I.2: Diagrama de Simulação com Probabilidades das Transições

II. CÓDIGOS FONTE

Código Fonte do Algoritmo de Verificação

```
def VerificaArquivodeRegistros(registros_stream, arquivo_regras)

  #rules is Array of Class Rule
  rules = LeituraRegras(arquivo_regras)
  if rules.length() == 0
    print "No rules read. Officially a success..."
    return true
  end

  puts "Read in #{rules.length()} rule(s)"

  #init stuff
  rule_count = rules.length()
  line_count = 1
  last_succeeded_line_index = 0
  last_succeeded_rule_index = 0
  last_succeeded_line = nil
  last_succeeded_rule = nil

  lines = []
  next_line = registros_stream.gets()
  #push all the logfile's lines into "lines" array
  while next_line
    if next_line.index(/\w/)
      lines = lines.push(next_line)
    end
    next_line = registros_stream.gets()
  end

  #setting arguments to the iterations in the logfile's lines
  line = lines[0]           #First line to be tested
  line_index=0              #Initial line index
  flag_loop=0               #Flag used to make the test of a non sequential
     rule for all the log lines: 1-Loop for all loglines
  rules_window=[]           #Array of rules for the forbidden sequential
     rules window
  quadro_proibido=[]
  non_sequencial_window=[]  #
  rulefile_end=0
  flag_quadro_proibido = 0
```

```
#iterate through the log file
while line_index < lines.length()+1

  line=lines[line_index]
  is_last_line = (line_index == lines.length()-1) #Test whether the line is
      the last log line

  #If the next rule to be tested is a negative rule, we must test this rule
      for all the log lines.
  #In this process we must save the line index to the following not "
     Forbidden" rule (reminder: the rules orders in this case is important)
  next_rule = rules[0]

  #Execution with no forbidden window
  if (quadro_proibido.length() == 0)

    if (flag_loop == 0) and !(next_rule.EhSequencial) and !(next_rule.
        EhRegradoQuadroProibido)
      saved_line_index = line_index
      line_index=0
      is_last_line = (line_index == lines.length()-1)
      flag_loop = 1
      line=lines[line_index]
    end

    #Making forbidden sequential rules window
    if next_rule.EhRegradoQuadroProibido
      i=0
      #Finding the next sequential non forbidden rule or the end of the
          rules
      while !(next_rule.EhSequencial and next_rule.EhRegraPositiva)
        #Sequential forbidden rules are added to an array
        if next_rule.EhRegradoQuadroProibido
          quadro_proibido = quadro_proibido.push(next_rule)
        elsif !next_rule.EhSequencial and next_rule != nil
          non_sequencial_window = non_sequencial_window.push(next_rule)
        end
        i=i+1                    #Go to the next rule in the file
        next_rule = rules[i]     #Set this rule for analysis
        if next_rule == nil      #In the end of the rulefile, it stops
            finding the sequential forbidden rules
          rulefile_end=1         #Flag to define the rulefile's end
          break
        end
      end
      quadro_proibido_length = quadro_proibido.length()
      next_rule_set = rules[i..-1] #Rules after the window
```

```
      window_fails = 0
      #The next sequential rule is add to the rules window if it exists
      if rules[i]
        next_rule = rules[i]
        quadro_proibido = quadro_proibido.push(next_rule)
        next_rule_set = next_rule_set[1..-1] #Rules after the sequential
            rule
      end
      rules = non_sequencial_window + next_rule_set  #Inserting the non
          sequential rules in the window into the rules array
      next_rule_set = rules
      non_sequencial_window = []     #Cleaning the non sequential rules
          array
    end #End of non sequential window creation
  end #End of execution with no forbidden sequential rules window
  #check the log line against the top rule
  #returns a rule_set with less entries if a match happened
  #For a forbidden window
  if (quadro_proibido != nil) and (quadro_proibido.length() != 0)
    i=0
    last_window_rule = quadro_proibido[-1]
    #Execution for the last rule  as a sequential one
    if last_window_rule.EhSequencial and last_window_rule.EhRegraPositiva
      last_rule_result = last_window_rule.VerificaLinha(line)
      #No match for the last sequential rule -> Verify all rules in the
          window
      if !last_rule_result
        current_window_rule = quadro_proibido[i]
        while current_window_rule.EhRegradoQuadroProibido
          current_rule_result = current_window_rule.VerificaLinha(line)
          #Match in the "ForbiddenWindow" rule -> Remove the rule from the
              window
          if current_rule_result
            quadro_proibido.delete_at(i)
            if quadro_proibido.length() == 1  #Fail when the last forbidden
                window rule match
              print "Forbidden window failed on rule: " +
                  current_window_rule.RuleDescription()
              print "On line: " + line
              next_rule_set = nil
              quadro_proibido = []
              break
            end
          else #No match in the "ForbiddenWindow" rule -> Verify next rule
            i=i+1
          end
          current_window_rule = quadro_proibido[i]
```

```
      end
    else #Match for the last sequential rule - Forbidden Window generates
        no failure
      quadro_proibido = []
      print "Forbidden Window passed: sequential rule matched.\n"
      next_rule_set = rules
    end
    #Execution for the last rule in the window as a "ForbiddenWindow"
       rule
  else
    if flag_quadro_proibido == 0
      beginning_of_quadro_proibido = line_index
      flag_quadro_proibido = 1
    end
    current_window_rule = quadro_proibido[i]
    while current_window_rule.EhRegradoQuadroProibido
      current_rule_result = current_window_rule.VerificaLinha(line)
      #Match in the "ForbiddenWindow" rule -> Remove the rule from the
         window
      if current_rule_result
        quadro_proibido.delete_at(i)
        if quadro_proibido.length() == 0  #Fail when the last forbidden
           window rule match
          print "Forbidden window rule failed on: " + current_window_rule
              .RuleDescription()
          print "On line: " + line
          next_rule_set = nil
          quadro_proibido = []
          break
        end
      else #No match in the "ForbiddenWindow" rule -> Verify next rule
        i=i+1
      end
      current_window_rule = quadro_proibido[i] # Take the new rule
      if !current_window_rule #If there is no rule, end of loop
        break
      end
    end
    # If is the logfile's end and there is any rule in the forbidden
       window that didn't match
    if is_last_line and (quadro_proibido.length() != 0)
      next_rule_set = rules
      if next_rule_set.length() != 0
        line_index = beginning_of_quadro_proibido
        is_last_line = (line_index == lines.length()-1)
        flag_quadro_proibido = 0
      end
```

```
        quadro_proibido = []
        print "Forbidden Window passed: end of logfile reached.\n"
      end
    end
  else #No forbidden window
    next_rule_set = CheckLogLine(line, rules, is_last_line)
  end
  #check for finish
  if (not next_rule_set or next_rule_set.length() == 0) and (
      quadro_proibido.length == 0) #No more rules and forbiddenwindow
    return AvaliaResultados(next_rule_set, last_succeeded_rule,
                            last_succeeded_rule_index,
                            last_succeeded_line, last_succeeded_line_index)
  elsif is_last_line and next_rule_set[0].EhRegraPositiva and next_rule.
      EhRegraPositiva
    return AvaliaResultados(next_rule_set, last_succeeded_rule,
                            last_succeeded_rule_index,
                            last_succeeded_line, last_succeeded_line_index)
  elsif (quadro_proibido.length == 0)
    # did we make a match?
    if next_rule_set.length() < rules.length()
      # yes => update status
      last_succeeded_line = line
      last_succeeded_rule = rules[0]
      last_succeeded_rule_index = (rule_count - rules.length()) + 1
      last_succeeded_line_index = line_index + 1
    end
    # move on to next ruleset if we had a match
    # keep the ruleset if we did not make a match
    rules = next_rule_set
    # index to the next line of the logfile
    line_count = line_count + 1
  end
  # move on to next ruleset if we had a match
  # keep the ruleset if we did not make a match

  line_index=line_index+1

  #In the end of the "Forbidden" rule checking of all the log lines, the
      line index must be recovered
  if (flag_loop != 0)
    if !next_rule.EhRegraPositiva and !next_rule.EhSequencial and
        is_last_line
      flag_loop = 0
      line_index=saved_line_index
    elsif next_rule.EhRegraPositiva and next_rule.VerificaLinha(line)
      flag_loop = 0
```

```
        line_index=saved_line_index
      end
    end

  end
  return AvaliaResultados(rules, last_succeeded_rule,
     last_succeeded_rule_index, last_succeeded_line,
     last_succeeded_line_index)
end

def CheckLogLine(log_line, rules, is_last_line, next_rule_index = 0)
  # get current rule
  front_rule = rules[next_rule_index]

  if front_rule == nil
    return nil
  end

  # check the rule against the line
  # true for a match, false if not
  # Caution!!! if this is a negative rule the result is inverted
  # negative Rule & match => false
  front_rule_result = front_rule.VerificaLinha(log_line)

  if front_rule_result == true
    if front_rule.EhRegraPositiva
      #match :-) => remove front_rule and return an array of the remaining
          rules
      return rules[next_rule_index + 1..-1]
    else
      #No Match && NegativeRule
      if is_last_line
        return rules[next_rule_index + 1..-1]
      else
        return rules
      end
    end
  else
    #  no match in this line
    if front_rule.EhRegraPositiva
      # return current array of rules
      return rules
    else
      # negative rule && false a match that should not happen!!
      print "Rule: " + front_rule.RuleDescription.strip() + " failed"+"\n"
      print "On line: " + log_line
```

```
      return nil
    end
  end
end

def VerificaLinha(log_line)
  if log_line.index(MessageToTestsuiteIdentifier)
    print log_line + "\n"
  end

  # ask the ruleFunction if this line matches
  line_distance = @RuleFunction.Evaluate(log_line)


  # result true if line_distance == 0 otherwise false
  result = (line_distance == 0)

  # save current Log Line to @BestLineYet
  # if @BestLineYet uninitialized (@BestLineDistance<0)
  # or
  # line_distance lower than the last saved @BestLineDistance
  # which will also be saved in both cases
  if line_distance < @BestLineDistance or @BestLineDistance < 0
    @BestLineYet = log_line.strip()
    @BestLineDistance = line_distance
  end

  if @PositiveRule
    return result
  else
    return (not result)
  end
end

def AvaliaResultados(rule_stack, last_succeeded_rule,
   last_succeeded_rule_index,
    last_succeeded_line,last_succeeded_line_index)
  if rule_stack and (not FirstPositiveRule(rule_stack))
    print "MESSAGE_TO_TESTSUITE All rules passed\n"
    return true
  elsif rule_stack
    next_positive_rule = FirstPositiveRule(rule_stack)
    print "MESSAGE_TO_TESTSUITE Positive rule failed\n"
    print "Rule No. " + (last_succeeded_rule_index + 1).to_s() \
    + ":'" + next_positive_rule.RuleDescription.strip() \
    + "'\ncould not be passed\n"
```

```
    print next_positive_rule.getFailureMessage

    if next_positive_rule.UseLineDistanceHeuristic()
      print  "Heuristically best candidate was: "\
      + next_positive_rule.BestLineYet + "\n"
    end

    if last_succeeded_line
      print "Last succeeded line " + last_succeeded_line_index.to_s + ": " \
      + last_succeeded_line.strip()\
      + "\n"
      print "Passed this rule   : " \
      + last_succeeded_rule.RuleDescription.strip() \
      + "\n"
    end
    return false
  else
    print "MESSAGE_TO_TESTSUITE Forbidden rule fulfilled\n"
    return false
  end
end
```

Script de Criação do Arquivo de Registros em Matlab

```
%Objetivo: Gerar um arquivo de registros com estados de operção simulados
%de um sistema autônomo.
clc
close all;
A = importdata('arquivo_de_registros.mat','Estado');
disp(A)

%Hashtable com os nomes dos estados
estados = java.util.Hashtable;
estados.put(1,'Atualizando_Tarefa');
estados.put(2,'Procurando_Objeto');
estados.put(3,'Objeto_Detectado');
estados.put(5,'Pegar_Objeto');
estados.put(6,'Calcular_Rota');
estados.put(7,'Caminho_Livre');
estados.put(8,'Obstaculo');
estados.put(9,'Evitar_Obstaculo');
estados.put(10,'Depositar_Objeto');
estados.put(11,'Objeto_Depositado');
estados.put(12,'Objeto_Perdido');
estados.put(13,'Erro');
estados.put(14,'Objeto_Desconhecido');
estados.put(90,'Inicializando_Sistema');
estados.put(91,'Calibrando_Sensores');
estados.put(92,'Sensores_Calibrados');
estados.put(93,'Nenhum_Dado_de_Sensor');
estados.put(94,'Falha_de_Leitura');
estados.put(95,'Inicializado_com_Sucesso');
estados.get(2)

%cria arquivo de log
logteste = fopen('logteste.txt', 'w');
fprintf(logteste, '%s\n',estados.get(A(2,1)));
for i = 2:31
    disp(estados.get(A(2,i)))
    if A(2,i)~= A(2,i-1)
        if A(2,i) == 3
            fprintf(logteste,'%s:_x%1.2f_y%1.2f\n',estados.get(A(2,i)), rand
                (1), rand(1));
        else
            fprintf(logteste,'%s\n',estados.get(A(2,i)));
        end
    end
end
fclose(logteste);
```

III. PROGRAMAS UTILIZADOS

Reunindo todas as referências obtidas alguns dos programas utilizados serão citados.

- Escrita do relatório:
  - Usou-se o Lyx como ambiente de desenvolvimento para criar o texto usando TEX. Ele permite que se use o TEX sem ter que se preocupar demais com coisas pequenas, como gerenciamento de rótulos e facilita, infinitamente, a criação de tabelas, matrizes, figuras etc.
  - Usou-se o SmartDraw para desenho dos diagramas de blocos. Para a inserir os diagramas no texto os mesmos foram exportados como .pdf.
  - Para gerenciamento da bibliografia, usou-se o BibTex. Ele também facilita a organização da sua biblioteca de referências.
- Ambiente de programação:
  - Linux: No Linux escreveu-se usando o Eclipse e compilação com makefiles.
  - Windows: Utilizou-se o programa Matlab para a programação de scripts e modelos utilizados nas simulações.