

**VERIFICAÇÃO DE PROJETOS DE CONTROLE DE VEÍCULOS
AÉREOS AUTÔNOMOS**

CLÁUDIO CAMPANHA FÉLIX

**VERIFICAÇÃO DE PROJETOS DE CONTROLE DE VEÍCULOS
AÉREOS AUTÔNOMOS**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: SÉRGIO VALE AGUIAR CAMPOS

COORIENTADOR: FABRÍCIO VIVAS ANDRADE

Belo Horizonte

29 de novembro de 2020

Félix, Cláudio Campanha.

F316v Verificação de projetos de controle de veículos aéreos autônomos [manuscrito] / Cláudio Campanha Félix. - 2020. xxv, 151 f. il.

Orientador: Sérgio Vale Aguiar Campos
Coorientador Fabrício Vivas Andrade

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f.97-100.

1. Computação – Teses. 2. Verificação formal – Teses. 3. SIMULINK (Software)– Teses 4. Software -Verificação - Teses. 5. Verossimilhança (Estatística) – Teses. I. Campos, Sérgio Vale Aguiar.II. Andrade, Fabrício Vivas. III.Universidade Federal de Minas Gerais; Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

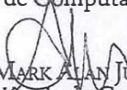
Verificação de projetos de controle de veículos aéreos autônomos


CLÁUDIO CAMPANHA FÉLIX

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. SÉRGIO VALE AGUIAR CAMPOS - Orientador
Departamento de Ciência da Computação - UFMG


PROF. FABRÍCIO VIVAS ANDRADE - Coorientador
Departamento de Computação - CEFET-MG


PROF. MARK ALAN JUNHO SONG
Departamento de Ciência da Computação - PUC Minas


PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG


PROF. KERLEY ALBERTO PEREIRA DE OLIVEIRA
Gestão Aeroespacial - Centro Universitário UNA

Belo Horizonte, 26 de Novembro de 2020.

Dedico este trabalho a quem esteja distante, como lembrete de que pode-se estar mais perto.

Agradecimentos

Agradeço à Fapemig pelo apoio à pesquisa.

Agradeço ao projeto PASARP pelo compartilhamento de conhecimentos, especialmente a Sérgio Campos pela orientação, Paulo Caires pela produção em parceria, Herbert Rausch pelas discussões sobre verificação formal, Ana Vilhena pela abertura de visão, Lincoln König pelos estudos sobre os modelos, Mário Oliveira pelas discussões sobre compiladores. Agradeço ao laboratório LUAR pelo acolhimento a esta pesquisa, abertura de portas e infraestrutura, especialmente a Alessandra Campos pelo encorajamento.

Agradeço ao CEFET-MG pela concessão de licença para o mestrado, especialmente a Satele Moreira por todo incentivo e apoio.

Agradeço ao DECOM/CEFET-MG pelo suporte através da coorientação de Fabrício Vivas, que mentora meus estudos desde a adolescência.

Agradeço ao Centro Universitário UNA pela parceria, através das colaborações de Kerley Oliveira sobre engenharia aeronáutica.

Agradeço à EMBRAER pela parceria, especialmente a Felipe Turetta pelo apoio no entendimento do problema abordado neste trabalho.

Agradeço a todas e todos que ainda, direta ou indiretamente, colaboraram com a construção deste trabalho.

“O que você alimentar.”
(Parábola)

Resumo

A verificação formal de sistemas de voo é necessária uma vez que sistemas de controle automático se tornaram amplamente aplicados, sendo complexos, interdisciplinares e possuindo componentes em alto grau de dependência. Falhas são responsáveis por prejuízos, de financeiros até vidas humanas. A expansão da aplicação de sistemas de controle automático gera possibilidades de mercado como o uso de veículos aéreos não tripulados (VANT) para o transporte de passageiros. A implantação comercial de VANT é restrita por barreiras comerciais que podem ser reduzidas ou removidas através da validação dos projetos relacionados. Isso impacta tanto na redução de acidentes quanto na apresentação de argumentos de confiança para os passageiros. Técnicas de validação de modelos comumente usadas como simulação e teste não são exaustivas e podem não detectar erros graves. A verificação formal de sistemas de voo, apesar de necessária para preenchimento de requisitos formais da indústria aeronáutica, é escassa devido à sua complexidade e custo computacional. Também são escassos modelos de aeronaves para compor conjuntos de testes de ferramentas de verificação de sistemas de voo. Este trabalho apresenta a implementação de um conjunto de modelos sintéticos em Simulink que são abstrações de elementos para sistemas de voo complexos. São apresentados 13 componentes classificados entre sistemas de alvo, rastreamento, orientação e *airframe* com composição modular para sistemas completos. Três composições de modelos completos são apresentadas com seus comportamentos simulados. Ele também realiza uma análise do problema da verificação de sistemas de aviação, partindo do estudo e desenvolvimento de modelos até a sua tradução e verificação. Para isso apresenta o SMCT, um tradutor de modelos descritos na linguagem Simulink para a linguagem XMV. O SMCT apresenta vantagens em relação a outras ferramentas que processam códigos Simulink por superar dificuldades apresentadas em outros trabalhos no que diz respeito à análise sintática/semântica de seu código-fonte textual. A partir dele traduz os modelos implementados e os verifica utilizando a ferramenta NuXMV, validando as funcionalidades nele modeladas. Além disso é verificado um conjunto de leis operacionais como exemplo de aplicação da técnica. Os resultados obtidos são um indicador positivo no sentido de se

verificar sistemas complexos como os de controle de voo autônomo, especialmente por associar técnicas automáticas de diferentes domínios, o que diminui a exigência multidisciplinar dos especialistas encarregados da detecção de falhas nos modelos. Este trabalho oferece um ferramental inicial para a redução das barreiras comerciais através da verificação formal de sistemas de aeronaves descritos em uma linguagem amplamente utilizada.

Palavras-chave: Aviação, Verificação Formal, Simulink.

Abstract

Flight systems formal checking became necessary once automatic control systems became widely applied, as they are complex, interdisciplinary and have highly dependent components. Failures are responsible from financial to human lives' losses. Expansion of automatic control systems application enables market innovations such as Unmanned Aerial Vehicles (UAV) for passenger transport. Commercial deployment of UAV gets restricted by commercial barriers which can be diminished or removed by validating related designs. This impacts in accident reduction as well as in presenting trust arguments for passengers. Commonly used techniques such as simulation and testing are proven incomplete and may not detect serious errors. Flight systems' formal checking, even though being necessary for fulfilling flight industries formal requirements, is scarce due to its complexity and computational cost. Flight system models for airflight systems verification tools benchmark sets are also scarce. This work presents a synthetic model set Simulink implementation built of complex airflight systems abstractions. Thirteen components are presented, classified as target, tracking, guidance and airframe systems having modular composition for complete systems building. Three complete model compositions are presented and simulated. This work also analyzes airflight systems' verification problem from the model's study and development to its translation and verification. For that, this work presents SMCT, a translator for models described in Simulink language to XMV language. SMCT presents improvements over other Simulink processing tools that overcome difficulties on textual source code syntax/semantics analysis. Applying SMCT, this work translates and verifies implemented models using NuXMV tool, validating modeled functionalities. An operational law set is also checked as an example of the technique application. Results are a positive pointer on complex systems' verification as autonomous flight, especially through different domain automatic techniques association, which reduces experts multidisciplinary requirements for models failure checking. This work offers a starting toolset for commercial barriers' reduction through airflight systems formal checking described in a widely used language.

Keywords: Flight systems, Formal methods, Simulink.

Lista de Figuras

| | | |
|------|---|----|
| 1.1 | Acidente envolvendo a aeronave do voo 447 da Air France em 2009 | 4 |
| 1.2 | Acidente envolvendo o foguete Ariane 5 em 1996 | 4 |
| 1.3 | Disparo de um míssil Patriot americano que falhou em interceptar um míssil iraquiano em 1991 | 5 |
| 2.1 | Mecanismos para fluxo de dados no Simulink. | 23 |
| 2.2 | Exemplo de um modelo que contém um erro <i>Write-After-Read</i> | 24 |
| 2.3 | Três leis operacionais de uma operação de pouso modeladas para a ferramenta NuXMV. | 35 |
| 2.4 | Propriedades a serem verificadas pela ferramenta NuXMV sobre as leis operacionais referidas na Figura 2.3 | 36 |
| 3.1 | Diagrama das etapas de geração à verificação de um sistema de voo | 37 |
| 3.2 | Visão geral do FGS | 39 |
| 3.3 | Diagrama geral de comando, controle, atuação e verificação de um sistema de voo | 41 |
| 3.4 | Modelo das dinâmicas da fuselagem de um projétil | 42 |
| 3.5 | Modelo Simulink contendo a referência (posição do alvo), o rastreador, o sistema de controle e a fuselagem autopilotada de um projétil. | 43 |
| 3.6 | Modelo do comportamento da <i>airframe</i> de um projétil (F3) [The Mathworks, 2014]. | 44 |
| 3.7 | Diagrama de transição de estados de um sistema de orientação modelado através do Matlab Simulink. | 45 |
| 3.8 | Uma implementação sobre uma estrutura de abstração dividida em Alvo, Rastreador, Orientação e <i>Airframe</i> | 46 |
| 3.9 | Modelo da <i>airframe</i> nos eixos cartesianos XZ considerando velocidade e atitude, com coordenadas GPS como saída (F1). | 46 |
| 3.10 | Modelo de um sistema rastreador de um míssil. | 47 |
| 3.11 | Modelo de um sistema rastreador em XZ sobre os eixos cartesianos. | 47 |

| | | |
|------|---|----|
| 3.12 | Modelo de um sistema de orientação de um míssil. | 48 |
| 3.13 | Modelo de um sistema de orientação baseado em atitude e velocidade. | 48 |
| 3.14 | Um modelo VANT com três graus de liberdade. | 49 |
| 3.15 | Modelo de um alvo estático com posição de destino (600, 300) nos eixos cartesianos XZ (A1). | 50 |
| 3.16 | Modelo de um alvo nos eixos cartesianos XZ iniciando em pouso, passando por decolagem, deslocamento e pouso. | 50 |
| 3.17 | Modelo de um alvo nos eixos cartesianos XZ com decolagem, deslocamento e pouso (A2). | 50 |
| 3.18 | Modelo de um alvo nos eixos cartesianos XZ controlado por coordenada polar e velocidade (A4) [The Mathworks, 2014] | 50 |
| 3.19 | Modelo para definição do comando direcional de um rastreador em XZ em quatro direções cartesianas (avanço, recuo, subida e descida). | 52 |
| 3.20 | Modelo para definição da velocidade em um rastreador em XZ em quatro direções cartesianas. | 52 |
| 3.21 | Modelo de um rastreador em X que considera o sensoriamento de obstáculos. | 53 |
| 3.22 | Um bloco para a orientação que, a partir de um comando que indique uma de quatro direções cartesianas, determina a atitude da aeronave. | 53 |
| 3.23 | Um bloco para orientação que, a partir de um comando que indique uma de quatro direções cartesianas e da altitude atual da aeronave, determina o estado da aeronave segundo leis operacionais que definem alturas de cruzeiro e de ligação dos motores horizontais. | 54 |
| 3.24 | Um bloco para a orientação que, a partir de uma velocidade comandada e do estado atual da aeronave, determina a sua velocidade operacional segundo leis operacionais que restringem a velocidade de cruzeiro, velocidade de subida vertical e altitude de cruzeiro. | 55 |
| 3.25 | Fragmento de uma orientação que determina a altitude referência para a <i>airframe</i> a partir da altitude relativa da aeronave em relação ao alvo e vetor de sensores para obstáculos. | 55 |
| 3.26 | Fragmento de uma orientação (O2) que determina o ângulo de inclinação referência para a <i>airframe</i> a partir do conjunto de comandos passados pelo sistema de rastreamento. | 56 |
| 3.27 | Modelo da <i>airframe</i> nos eixos cartesianos XZ considerando parâmetros físicos e ambiente | 57 |
| 3.28 | Simulações da aproximação da <i>airframe</i> em relação ao alvo utilizando o modelo apresentado na Figura 3.5 com ângulo de deslocamento do alvo em 3.1416 radianos e velocidade do alvo em 328 unidades por ciclo. | 57 |

| | | |
|------|---|----|
| 3.29 | Simulações da aproximação da <i>airframe</i> em relação ao alvo utilizando o modelo apresentado na Figura 3.8 com posicionamento do alvo em 5000 no eixo X a partir do tempo 10 e 100 no eixo Z a partir do tempo 5, ambos iniciados em 0. | 58 |
| 3.30 | Simulações da aproximação da <i>airframe</i> em relação ao alvo utilizando o modelo apresentado na Figura 3.14 com posicionamento do alvo em 5 no eixo X a partir do tempo 100 e 20 no eixo Z a do tempo 60 ao tempo 140, tendo início com X em 0 e Z em 10. | 59 |
| 4.1 | O algoritmo de tradução utiliza da relação entre blocos Simulink e módulos NuXMV, modelados intermediariamente como objetos de classes Java. De forma análoga, os sinais no Simulink são relacionados a variáveis NuXMV, modelados intermediariamente como atributos com seus valores assinalados nos objetos Java. | 65 |
| 5.1 | Trecho da saída da ferramenta NuXMV atestando o cumprimento das propriedades verificadas pelo modelo utilizando BMC até uma profundidade k=1000. | 76 |
| 5.2 | Modelo apresentado na Figura 3.19 traduzido na Seção 5.1. | 79 |
| 5.3 | Modelo apresentado na Figura 3.23 traduzido na Seção 5.2. | 83 |
| 5.4 | Modelo apresentado na Figura 3.25 traduzido na Seção 5.3. | 87 |
| 5.5 | Modelo apresentado na Figura 3.26 traduzido na Seção 5.3. | 88 |
| 5.6 | Fragmento do estado inicial do contraexemplo de saída que contém as variáveis que geram a violação da Lei Operacional 4 pelo modelo traduzido na Seção 5.2. | 90 |
| 5.7 | Fluxo de dados correspondente ao estado do contraexemplo apresentado na Figura 5.6, onde a entrada <i>Comando</i> com valor igual ao da constante <i>Comandosubir</i> habilita a constante <i>Decolagem</i> na saída <i>Estado</i> do sistema. | 90 |
| 5.8 | Fragmento do estado de erro do contraexemplo de saída que contém as variáveis que geram a violação da Lei Operacional 4 pelo modelo traduzido na Seção 5.2. | 90 |
| 5.9 | Fluxo de dados correspondente ao estado do contraexemplo apresentado na Figura 5.8, onde a entrada <i>Comando</i> com valor igual ao da constante <i>Comandoparadireita</i> e a entrada <i>Z</i> com valor igual a 340 habilitam a constante <i>Cruzeiro</i> na saída <i>Estado</i> do sistema. | 91 |

Lista de Tabelas

| | | |
|-----|---|----|
| 1.1 | Dados de fatalidades ocorridas nos Estados Unidos em 2013 para automóveis, linhas aéreas, táxis aéreos, motocicletas e aviação em geral. | 7 |
| 1.2 | Dados de fatalidades normalizadas em relação às ocorridas em automóveis nos Estados Unidos em 2013 para automóveis, linhas aéreas, táxis aéreos, motocicletas e aviação em geral. | 7 |
| 3.1 | Sistemas componentes dos modelos completos. | 40 |

Sumário

| | |
|---|-------------|
| Agradecimentos | ix |
| Resumo | xiii |
| Abstract | xv |
| Lista de Figuras | xvii |
| Lista de Tabelas | xxi |
| 1 Introdução | 1 |
| 1.1 Organização do trabalho | 2 |
| 1.2 Falhas em veículos aéreos autônomos | 3 |
| 1.3 Confiabilidade mercadológica sobre o uso de VANTs para transporte de passageiros | 6 |
| 1.4 Tecnologias recentes para veículos aéreos autônomos | 7 |
| 1.5 Teste e simulação de falhas em veículos aéreos autônomos | 11 |
| 1.6 Trabalhos relacionados a novas soluções para a detecção de falhas em veículos aéreos autônomos | 13 |
| 1.7 Tradução combinada à verificação formal como solução para a detecção de falhas em veículos aéreos autônomos | 15 |
| 1.8 Criação de um conjunto de modelos sintéticos de aeronaves autônomas | 18 |
| 2 Fundamentação Teórica | 21 |
| 2.1 Ambiente de projeto Matlab/Simulink | 21 |
| 2.2 Projeto de Veículos Aéreos Não Tripulados | 24 |
| 2.3 Detecção de falhas em modelos | 27 |
| 2.3.1 Formalismos lógicos | 28 |
| 2.3.2 Motores para verificação de modelos | 30 |

| | | |
|----------|--|-----------|
| 2.3.3 | Técnicas para a verificação de modelos | 32 |
| 2.3.4 | SMV e ferramentas derivadas para verificação de modelos | 33 |
| 3 | Geração de modelos de sistemas de aeronaves | 37 |
| 3.1 | Fluxo de informação do comando à atuação | 38 |
| 3.2 | Análise da estrutura de projeto de um modelo de controle de projétil em Simulink | 41 |
| 3.3 | Estrutura de abstração para modelos de aeronaves controladas em Simulink | 44 |
| 3.4 | Decisões de implementação de modelos de aeronaves controladas em Simulink | 49 |
| 3.4.1 | Implementação do bloco alvo | 49 |
| 3.4.2 | Implementação do bloco rastreador | 51 |
| 3.4.3 | Implementação do bloco orientação | 51 |
| 3.4.4 | Implementação do bloco <i>airframe</i> | 54 |
| 3.5 | Simulação de modelos implementados | 57 |
| 4 | Tradução automática de modelos Simulink para NuXMV | 61 |
| 4.1 | A ferramenta <i>Simulink to Model Checking Translator</i> | 62 |
| 4.2 | Algoritmo e Implementação da tradução automática de Simulink para NuXMV | 64 |
| 4.3 | Blocos Simulink traduzidos automaticamente pela SMCT | 66 |
| 4.3.1 | Seletor de Barramentos (Bus Selector) | 66 |
| 4.3.2 | Comparação a constante (Compare to Constant) | 67 |
| 4.3.3 | Constante (Constant) | 68 |
| 4.3.4 | Demultiplexador (Demux) | 68 |
| 4.3.5 | Ganho (Gain) | 68 |
| 4.3.6 | Entrada (Inport) | 69 |
| 4.3.7 | Integrador pelo método de somas discretas (Integrator) | 69 |
| 4.3.8 | Lógico (Logical Operator) | 70 |
| 4.3.9 | Memória (Memory) | 70 |
| 4.3.10 | Multiplexador (Mux) | 71 |
| 4.3.11 | Saída (Outport) | 71 |
| 4.3.12 | Controlador PID pelo método de Euler (PID Controller) | 71 |
| 4.3.13 | Produto (Product) | 72 |
| 4.3.14 | Operador relacional (Relational Operator) | 73 |
| 4.3.15 | Saturação (Saturation) | 73 |
| 4.3.16 | Soma (Sum) | 74 |

| | | |
|----------|--|------------|
| 4.3.17 | Comutador (Switch) | 74 |
| 5 | Tradução e verificação de projetos de sistemas autônomos | 75 |
| 5.1 | Tradução do subsistema de direção para o rastreador R1 | 76 |
| 5.2 | Tradução do subsistema de estado para a orientação O1 | 78 |
| 5.3 | Tradução do bloco de orientação O2 | 82 |
| 5.4 | Exemplo de validação de leis operacionais | 88 |
| 6 | Conclusão | 93 |
| | Referências Bibliográficas | 97 |
| | Apêndice A Detalhamento estrutural dos blocos Simulink traduzidos automaticamente pela SMCT | 101 |
| A.1 | Seletor de Barramentos (Bus Selector) | 101 |
| A.2 | Comparação a constante (Compare to Constant) | 103 |
| A.3 | Constante (Constant) | 105 |
| A.4 | Demultiplexador (Demux) | 106 |
| A.5 | Ganho (Gain) | 107 |
| A.6 | Entrada (Inport) | 108 |
| A.7 | Integrador pelo método de somas discretas (Integrator) | 110 |
| A.8 | Lógico (Logical Operator) | 112 |
| A.9 | Memória (Memory) | 113 |
| A.10 | Multiplexador (Mux) | 115 |
| A.11 | Saída (Outport) | 116 |
| A.12 | Controlador PID pelo método de Euler (PID Controller) | 118 |
| A.13 | Produto (Product) | 122 |
| A.14 | Operador relacional (Relational Operator) | 123 |
| A.15 | Saturação (Saturation) | 125 |
| A.16 | Soma (Sum) | 126 |
| A.17 | Comutador (Switch) | 128 |
| | Apêndice B Identificadores de estruturas do Simulink Comuns em Modelos VANT131 | |
| | Anexo A Códigos resumidos dos modelos-exemplo traduzidos | 137 |
| A.1 | Código do cálculo de direção | 137 |
| A.2 | Código do cálculo de orientação [Turetta, 2018] | 140 |
| A.3 | Código de cálculo do estado para a direção | 149 |

Capítulo 1

Introdução

Sistemas para aeronaves são complexos pelo seu número de componentes, interdisciplinaridade de seus módulos e interdependência entre eles. Quando essas são remotamente pilotadas, acrescenta-se a complexidade de uma rede crítica de tomadas de decisão. Seu desenvolvimento possui um custo elevado associado a um grande número de erros e à dificuldade de encontrá-los e corrigi-los. A importância de sistemas de aeronaves remotamente pilotadas tem crescido a cada dia, com aplicações desde militares até aplicações cotidianas como o transporte de passageiros, passando por aeronaves de monitoramento, reconhecimento, entrega de carga, dentre outras. Em diversas dessas aplicações a correção de seus sistemas é fundamental, com proteção a prejuízos que vão de financeiros até vidas humanas.

O desenvolvimento de novos projetos de aeronaves, cada vez mais complexos, aumenta o número de componentes susceptíveis a erros. Novos pontos de falha surgem a partir da evolução de projetos legados, utilização de processadores próprios, softwares de grande tamanho e de sensores e atuadores inteligentes com sistemas internos próprios cujo comportamento é difícil de prever. Cada componente utilizado possui difícil garantia de funcionamento correto, mesmo analisado individualmente. Enquanto interage no sistema, essa garantia se torna menor ainda pelo aumento da complexidade geral do sistema, além da probabilidade de erros na troca de mensagens. A ordem da complexidade refere-se ao número de comportamentos possíveis devido à interação de componentes complexos, que é exponencial em relação ao número de componentes.

O problema da interação de subsistemas dentro de um sistema é notavelmente agravado pela característica interdisciplinar dos projetos de aeronaves remotamente pilotadas [Alino, 2016]. Equipes de desenvolvimento possuem processos bem definidos internamente. No entanto a interface entre os vários departamentos responsáveis por diferentes tecnologias, que comumente pertencem a empresas distintas, é um desafio. Nesse

cenário fatores como prazos, multas e requisitos do projeto costumam ser muito discutidos e redefinidos. Assim, requisitos técnicos necessários para o correto desenvolvimento de hardware e software se mostram incompletos e muitas vezes falhos. São comuns problemas e retrabalhos devido à má comunicação entre as equipes e empresas ao longo do desenvolvimento e da fase de testes do projeto. Os custos decorrente de retrabalhos são significativos bem como de atrasos, incidentes e principalmente acidentes aeronáuticos. Por esse motivo, os casos de falência de indústrias aeronáuticas não são raros, dado que falhas de software e hardware podem ter consequências catastróficas em sistemas de controle de voo.

É importante ressaltar que os acidentes mencionados são causados pela falha nos projetos do sistema das aeronaves devido à complexidade dos sistemas sendo desenvolvidos atualmente, e não por falhas inerentes ao processo de desenvolvimento. Quando, contudo, estes problemas ocorrem, tornam-se necessários métodos de validação dos requisitos e de suas implementações mais poderosos do que os utilizados atualmente. A utilização de métodos formais para validação destes requisitos é uma alternativa eficiente para resolver este problema. O uso combinado de tecnologias consolidadas pode ser aplicado neste tipo de projeto, como o das propostas neste trabalho.

O tratamento de tais falhas se torna ainda mais urgente em um contexto em que Veículos Aéreos Não Tripulados (VANTs) começam a ser utilizados para transporte de passageiros. Sua implementação sugere a abertura de um novo mercado de grande valor para a sociedade. A aplicação de VANTs para transporte de passageiros remove o problema do trânsito terrestre e permite a aplicação urbana da *aviação sob demanda* que trata da disponibilização constante de aeronaves para transporte. O projeto Uber Elevate [Jeff Holden, 2016], por exemplo, expõe novas soluções para o problema da mobilidade urbana usando VANTs em viagens por médias e longas distâncias. Como impacto na mobilidade urbana, espera-se para as viagens: aumentar o nível de segurança, reduzir o tempo na ordem de dezenas de vezes e o custo financeiro no mínimo pela metade em longo prazo.

1.1 Organização do trabalho

Este trabalho organiza-se da seguinte forma: O presente capítulo trata sobre o contexto em que se encontra o problema da verificação de sistemas de veículos aéreos autônomos, apresentando as tecnologias recentes utilizadas nos mesmos, mecanismos de teste e simulação, novas soluções para a detecção de falhas e introduz a proposta da tradução combinada à verificação formal para a detecção das mesmas. O Capítulo 2 introduz os

principais conceitos utilizados no trabalho, situando os termos utilizados na literatura específica da área. O Capítulo 3 apresenta uma metodologia para a geração de modelos para ferramentas sobre projetos de sistemas autônomos com detalhamento de decisões e da implementação dos principais elementos de um modelo de aeronave em Simulink. Assim, apresenta um modelo adaptável a partir de 5 componentes implementados e mais 4 adaptados de um modelo completo para a estrutura de abstração utilizada. Ele também apresenta a simulação dos modelos implementados. O Capítulo 4 apresenta a ferramenta SMCT, utilizada para a tradução dos modelos Simulink em linguagem própria para sua verificação formal. O Capítulo 5 apresenta os resultados da tradução e verificação dos modelos implementados, tanto no que diz respeito à validação da tradução quanto à verificação de leis operacionais. O Capítulo 6 traz uma análise sobre as contribuições do trabalho sobre o escopo abordado, além de sugestões de continuidade. O Apêndice A traz o detalhamento estrutural dos blocos Simulink traduzidos pela ferramenta SMCT de modo a esclarecer as decisões tomadas no processo de tradução. O Apêndice B traz o principal conjunto de identificadores encontrados em modelos de VANT em Simulink para a orientação da leitura de arquivos de código-fonte em formato *.mdl*. O Anexo A traz exemplos de códigos *.mdl* resumidos para a referência sobre a natureza do código-fonte de entrada da ferramenta SMCT.

1.2 Falhas em veículos aéreos autônomos

Veículos aéreos autônomos consistem em sistemas críticos cujas falhas causam prejuízos não só na ordem econômica, mas também à vida humana. Exemplos de desastres provocados por falhas são abundantes como o do voo 447 da Air France em 2009 (Figura 1.1), cuja aeronave caiu devido ao tratamento incorreto da informação fornecida por seus múltiplos sensores de velocidade [Air France, 2011]. As informações conflitantes de seus sensores ocorreram devido a falhas simultâneas, o que levou a uma interpretação incorreta dos dados gerados. Problemas gerados por eventos combinados são difíceis de se identificar devido ao tamanho do conjunto de combinações possíveis.

Outro exemplo, em outro domínio, é o da explosão do foguete Ariane 5 (Figura 1.2) em 1996, 40 segundos após o seu lançamento tendo tido seu desenvolvimento por 10 anos e um investimento de 7 bilhões de dólares [Gleick, 1996, Lions, 1996]. Nesse caso o erro foi causado por um *overflow* na conversão de uma velocidade calculada em uma representação de 64 bits para uma representação de 16 bits. A redundância do cálculo através de duas unidades distintas foi inútil para o caso, uma vez que ambas executavam o mesmo software. Problemas dessa natureza, além de problemas de bloqueio, concor-



Figura 1.1: Acidente envolvendo a aeronave do voo 447 da Air France em 2009¹.



Figura 1.2: Acidente envolvendo o foguete Ariane 5 em 1996².

rência, falhas aritméticas e outros inerentes à integração hardware-software ocorrem com frequência e necessitam de tratamento correto.

No contexto militar, em 1991, um míssil Patriot americano falhou em interceptar um míssil Scud disparado pelas forças armadas iraquianas (Figura 1.3)[Subcommittee on Investigations and Oversight, 1992]. O projétil iraquiano acertou um acampamento da Coalisção, matando 28 soldados. Eventualmente constatou-se que a falha de interceptação deu-se devido a um erro de arredondamento de números de ponto flutuante. Um erro de precisão na medição do tempo foi gerado pelo seu

¹https://s.abcnews.com/images/WNT/abc_wn_france_110524_wg.jpg acesso em Maio de 2020.

²https://i.ytimg.com/vi/PK_yguLapgA/hqdefault.jpg acesso em Maio de 2020.



Figura 1.3: Disparo de um míssil Patriot americano que falhou em interceptar um míssil iraquiano em 1991³.

relógio que era mantido em décimos de segundo. Para a conversão para segundos, esse tempo era multiplicado por 1/10 que é representado em binário através de uma dízima periódica. Essa dízima periódica era arredondada a partir do bit 24, cujo erro, acumulado desde a operacionalização da bateria (aproximadamente 100 horas) gerou um erro de cerca de 0.34 segundos, tempo suficiente para que um foguete Scud percorra mais de meio quilômetro de distância. Problemas cuja natureza é facilmente identificável através de propriedades temporais, como os de propagação de erro através de ciclos, são negligenciados facilmente em testes e simulações.

Falhas como essas geraram diversos acidentes nos últimos anos. As principais causas de acidentes aéreos estão relacionadas à falta de informação como, por exemplo, falta de cobertura por radar, controle de tráfego aéreo assíncrono e má previsão do tempo e geram acidentes como voo controlado de encontro ao terreno, colisões no ar e perda de controle. Esses acidentes podem ser prevenidos através de formas de autonomia veicular que provejam controle suplementar vinculado a um conjunto mais completo de informações. Tais informações podem ser fornecidas através de mecanismos externos às aeronaves ou através da melhoria do sistema de sensoriamento. Sistemas de controle dos atuadores da aeronave podem então ser associados a sistemas de auxílio à pilotagem reduzindo os modos de falha atribuíveis a seus erros. Sistemas de ajuda à pilotagem podem evoluir então à autonomia completa, trazendo um grande impacto à segurança de voo. Relevante notar que a elevação dos níveis de segurança de voo está condicionada à corre-

³<https://www.iro.umontreal.ca/mignotte/IFT2425/image/patriot.jpg> acesso em Maio de 2020.

tude de tais sistemas e que ela pode ser garantida através da verificação de seus modelos. Os problemas de percepção das informações pelos pilotos são tais que cerca de metade dos acidentes estão essencialmente relacionados a má informação sobre clima e pilotos não sabendo onde estão. A adoção de auxílios avançados a pilotos e sistemas autônomos devem ser suficientes para tornar sistemas de voo tão seguros quanto sistemas automobilísticos [Jeff Holden, 2016].

1.3 Confiabilidade mercadológica sobre o uso de VANTs para transporte de passageiros

A expansão da aplicação de sistemas de controle autônomo gera possibilidades de mercado como o uso de veículos aéreos não tripulados para o transporte de passageiros. A necessidade de sistemas de controle de voo remotos torna ainda mais relevante a verificação dos modelos, evitando acidentes e apresentando um argumento de confiança para os passageiros. Técnicas de validação de modelos como simulação e teste, apesar de comumente utilizadas, possuem cobertura incompleta dos casos, gerando argumentos de menor força sobre a confiabilidade do sistema. Aplicação de técnicas para a verificação formal de tais sistemas são escassas devido à sua complexidade, interdisciplinaridade e dificuldade de geração de modelos cujo custo computacional seja razoável.

Barreiras para a aplicação de veículos aéreos não tripulados para transporte de passageiros no mercado se apresentam devido à novidade de sua aplicação. Elas se relacionam a tecnologias em desenvolvimento, registros legais, experiência de usuários e confiança da sociedade em relação ao sistema. Barreiras destacadas pelo projeto Uber Elevate [Jeff Holden, 2016] são: o processo de certificação junto às autoridades locais; tecnologias de energia para sustentação das aeronaves; eficiência energética das mesmas; confiabilidade e desempenho das aeronaves, o que diz respeito aos tempos de encontro com os passageiros e de viagem; controle de tráfego aéreo; custo; segurança; ruídos sonoros; emissão de poluentes; estrutura de pouso e treinamento de pilotos. Segundo critérios de segurança definidos no projeto, a população deve perceber uma maior segurança em viajar por um VANT do que por um automóvel. As melhorias de segurança em automóveis autônomos inserem uma responsabilidade ainda maior ao tratar deste quesito. Atualmente o uso de um táxi aéreo possui um nível de segurança duas vezes pior do que o uso de um automóvel, quando se trata da distância percorrida por passageiro. As Tabelas 1.1 e 1.2 apresentam dados de fatalidades absolutas e normalizadas em relação às ocorridas em automóveis nos Estados Unidos em 2013 para automóveis, linhas aéreas, táxis aéreos, motocicletas e aviação em geral.

| Tipos de veículo | Utilização anual da frota | | | Fatalidades (média anual) |
|------------------|--------------------------------|-----------------------------------|---------------------------------------|------------------------------|
| | Horas de veículo (1.000) | Milhas de veículo (milhões) | Milhas de passageiros (milhões) | |
| Carros | 50.300.000 | 1.510.000 | 2.340.000 | 14.701 |
| Linhas aéreas | 18.600 | 7.891 | 579.000 | 16 |
| Taxis aéreos | 2.100 | 375 | 1.500 | 18 |
| Motocicletas | 600.000 | 18.000 | 19.800 | 4.809 |
| Aviação geral | 22.400 | 3.370 | 6.740 | 511 |

Tabela 1.1: Dados de fatalidades corridas nos Estados Unidos em 2013 para automóveis, linhas aéreas, táxis aéreos, motocicletas e aviação em geral [Jeff Holden, 2016].

| Tipos de veículo | Taxas de fatalidade normalizadas | | |
|------------------|----------------------------------|---|--|
| | Por 100.000 horas de veículo | Por 100 milhões de milhas de veículo | Por 100 milhões de milhas de passageiro |
| Carros | 1X (0,030) | 1X | 1X (0,643) |
| Linhas aéreas | 2,9X | 0,208X | 0,004X |
| Taxis aéreos | 29,3X | 4,9X | 1,9X |
| Motocicletas | 27,4X | 27,4X | 38,7X |
| Aviação geral | 78,1X | 15,6X | 12,1X |

Tabela 1.2: Dados de fatalidades normalizadas em relação às ocorridas em automóveis nos Estados Unidos em 2013 para automóveis, linhas aéreas, táxis aéreos, motocicletas e aviação em geral [Jeff Holden, 2016].

Existe uma preocupação na indústria aeronáutica com o desenvolvimento e formalização de requisitos para sistemas de software e outros sistemas aeronáuticos. Documentos publicados detalham estes requisitos como os DO-178 [SC-205, RTCA, Inc, 2011] publicados pela RTCA (Radio Technical Commission for Aeronautics). Esses documentos incluem detalhamentos das necessidades e possibilidades de aplicação de métodos formais em sistemas aeronáuticos, como a DO-333 [Miller, 2014, Miller, , SC-205, RTCA, Inc, 2015], demonstrando a relevância da sua aplicação nesse contexto. Contudo, a aplicação destes métodos ainda é restrita.

1.4 Tecnologias recentes para veículos aéreos autônomos

Tecnologias têm sido empregadas com o intuito de reduzir as ocorrências de acidentes aeronáuticos. A automação das aeronaves apresenta relevância nesse domínio, uma vez que sua corretude pode ser elaborada ainda na etapa de projeto. Aplicação de redundân-

cia, controle eficiente das dinâmicas do sistema [Riddick, 2018], diminuição da responsabilidade humana na missão da aeronave, otimização do uso de recursos, otimização de rotas, prevenção de colisões, dentre outras técnicas têm sido desenvolvidas e evoluídas no sentido da autonomia total das aeronaves.

A substituição de helicópteros por aeronaves de pouso e decolagem verticais (VTOL, do inglês *Vertical Take-off Landing*) tem sido empregada na aviação urbana por apresentar vantagens significativas. Essas vantagens se evidenciam quando se realiza estudos sobre situações de falhas. O dimensionamento de tolerância a falhas em um motor, por exemplo, representa um desafio muito menor em um VTOL hexacóptero do que em um helicóptero de asas rotativas duplas. Enquanto helicópteros, diferentemente de aeronaves VTOL, são aptos a realizar uma autorrotação e realizar um pouso de emergência sem potência, a autorrotação não é bem operada em áreas urbanas densas com voos em baixa altura. O uso de aeronaves VTOL com múltiplos rotores de propulsão se torna ainda mais poderoso quando associado à autonomia veicular, de modo que se previna estados potencialmente danosos.

Güçlü [Güçlü, 2016] apresenta o projeto no ambiente Simulink de um veículo híbrido VTOL asa-fixa, modificado a partir de um modelo remotamente pilotado (*E-Flite Apprentice Model Plane*). Seu modelo possui três modos de operação principais: decolagem e pouso verticais; modo de transição; voo em asa fixa. Seu controle foi realizado a partir de controladores LADRC (Controlador de Rejeição Linear Ativa de Perturbações) e PID (Controlador Proporcional Integral Derivativo). Esse trabalho é especialmente interessante por condensar o modelo de voo em apenas três estágios e por utilizar um controlador com modelagem em verificação de modelos já encontrada na literatura [Vilhena, 2018].

O uso de múltiplos motores de propulsão através da propulsão distribuída provê, além de redundância, o potencial para robustez adicional no controle do sistema de aeronave de forma que seus componentes possam falhar sem comprometer o controle do pouso e decolagem. A complexidade do controle de múltiplos motores de propulsão deve ser considerada ao buscar a segurança de sistemas de voo. Uma mudança de tecnologia que substitua motores a combustão por motores elétricos corrobora com o controle autônomo dos mesmos. Problemas em motores a combustão são responsáveis por 18% dos acidentes em voo somados a problemas de administração de combustível. A substituição de motores a combustão por motores elétricos (DEP, *Distributed Electric Propulsion*) favorece a aplicação de sistemas de controle, que podem ser verificados. Em situações emergenciais, motores de turbinas e pistões são capazes de oferecer um aumento emergencial de potência entre 10 e 20%. Enquanto isso, motores elétricos são capazes de fornecer acima de 50% por 1 a 2 minutos até o sistema sobreaquecer, tempo sufici-

ente para a execução de manobras emergenciais. O uso de propulsão elétrica distribuída combinada com aeronaves autônomas permite a interação de sistemas digitalmente controlados através de sistemas digitais verificáveis sem interfaces complexas analógicas ou mecânicas. Dados digitais de cada elemento do sistema de propulsão são administrados através de controladores-mestre de voo redundantes, responsáveis por componentes que vão desde o estado de tensão das baterias até a temperatura dos motores.

O uso da tecnologia DEP se torna ainda mais interessante quando associada às técnicas de controle robusto. O controle robusto da aeronave provê a habilidade de lidar com incertezas ou distúrbios durante um voo como ventos fortes e temporais em ambientes como o urbano que gera distúrbios de fluxo locais. Voos verticais VTOL apresentam desafios adicionais em relação a voos tradicionais asa fixa. O controle robusto da propulsão elétrica distribuída mitiga a maior parte desses desafios. VTOLs utilizando a tecnologia DEP tendem a possuir uma maior velocidade de pouso, até 10 vezes mais rápida do que helicópteros com rotores típicos, que quando utilizada com múltiplos propulsores rotores auxilia a minorar condições indesejadas causadas por recirculação de ar (e.g. *vortex ring state*, que é uma condição aerodinâmica que surge durante o vôo do helicóptero, quando um sistema de anel de vórtice envolve o rotor, causando severa perda de sustentação.). Além disso, o controle robusto de DEP provê uma maior confiabilidade no caso emergencial de perda de motores, tanto por ser capaz de gerar uma maior potência por mais tempo em situações de emergência quanto por comumente ser aplicada em veículos com um maior número de motores, reduzindo o impacto da falha de um deles e permitindo um pouso controlado.

Pesquisa sobre tecnologias de controle de voo resiliente de aeronaves sob condições de voo adversas foi conduzida pelo NASA Aviation Safety Program [Riddick, 2018]. Com o intuito de reduzir a taxa de acidentes fatais de grandes aviões de transporte devido à perda de controle, conduziu uma série de avaliações de algoritmos de controle de voo utilizados durante operações de voo completas. Desse modo, diversas tecnologias de controle de voo foram avaliadas quanto à sua habilidade em prevenir perda de controle e estabilidade. As leis de controle do modelo utilizado incluem três metodologias de controle adaptativo, diversos projetos lineares multivariáveis, um projeto linear robusto, um sistema de melhoria de estabilidade linear e um modo de controle direto em malha aberta. Foi demonstrado então que o controle adaptativo apresenta melhores resultados de estabilidade, segundo as avaliações de Cooper-Harper⁴.

Operações de voo nos Estados Unidos da América seguem a documentação Part 135, com equivalentes em diversos países, que requerem dos pilotos uma licença comer-

⁴Avaliações de Cooper-Harper consistem na pontuação de diversos critérios subjetivos sobre a experiência do piloto ao conduzir uma aeronave.

cial, experiência mínima como piloto com condições visuais (VFR) de 500 horas e 1200 horas de voo sem condições visuais (IFR). Pilotos para aeronaves VTOL devem possuir ambas experiências em pilotagem de aeronaves asa-fixa e helicópteros, com tempo total para os requisitos sendo composto em qualquer proporção dentre elas. Complementarmente a esses requisitos, pilotos de aeronaves autônomas necessitam desenvolver habilidades relativas ao processo automático e seus ajustes. A autonomia nesse caso se refere à habilidade de o veículo realizar os ajustes de trajetória e parâmetros que forcem a rota ao caminho desejado, onde o piloto deve determinar apenas a trajetória desejada sem determinar como alcançá-la.

Sistemas mais seguros deveriam contar com pilotos apenas para o desvio visual de obstáculos e outras aeronaves [Jeff Holden, 2016]. Ao invés de comandar fisicamente a operação de motores e superfícies de controle, o piloto estabelece uma trajetória desejada a ser seguida pela aeronave. A redução no nível de interação do piloto possibilita a dedicação de uma maior parte de sua atenção a situações de emergência. A expectativa ao se utilizar sistemas com níveis mais elevados de autonomia é a redução dos requisitos de experiência dos pilotos, devido ao escopo reduzido de tarefas pelas quais eles são responsáveis. Deve-se considerar que um piloto particular típico investe de 8 a 10 horas aprendendo as manobras básicas de voo e o tempo restante aprendendo a lidar com situações excepcionais como *stalls*, más condições de voo, operações sob vento cruzado e falha de motores. Pilotos comerciais e instrumentais seguem a mesma estrutura, com aeronaves mais complexas e níveis mais elevados de precisão em equipamentos de navegação. Uma vez que esses modos de falha estejam contemplados pelos projetos de aeronaves autônomas o piloto não precisa tomar medidas corretivas para assegurar um voo seguro. Dessa forma um treinamento muito mais curto pode ser utilizado para atingir operações seguras sobre todos os potenciais (e conhecidos) modos de falha da aeronave.

O controle autônomo de voo é capaz de prover melhores perfis de trajetória de voo que minimizam a potência extra necessária para controle utilizando a combinação de parâmetros ótimos como velocidade, ângulo de subida, ângulo de ataque e ângulos de inclinação de asa/propulsão durante a transição estado de suspensão ao estado de voo em cruzeiro. O ganho de segurança através do uso do controle autônomo de voo e sua percepção pela sociedade é importante fator na adoção inicial da utilização de aeronaves para mobilidade urbana. Para redução de danos, tecnologias como modos de emergência equivalentes à utilização de acostamentos no transporte rodoviário, emprego de paraquedas como recurso para trazer a aeronave em segurança ao solo e *airbags* aplicados à aeronave inteira têm sido projetadas. Sistemas autônomos de prevenção de colisão com o solo são cada vez mais eficientes e já possuem aplicação em caças F-16. Novas soluções proveem segurança adicional para praticamente todas as condições de operação do

veículo. Grandes volumes de dados de operações de mundo real têm sido aplicadas aos projetos, de modo que operações utilizando aeronaves VTOL alcancem a confiabilidade da aviação das linhas aéreas. Aeronaves VTOL podem utilizar sistemas digitais *fly-by-wire*, que é um tipo de controle das superfícies móveis de um avião por um computador, permitindo que qualquer modificação da direção e do sentido de uma aeronave feita pelo piloto seja processada e repassada para as superfícies móveis da aeronave, adaptando esses sistemas através da inclusão de assistência e reduzindo significativamente falhas relativas a erros de pilotagem. Essa assistência tende a evoluir na direção da autonomia completa, representando um marco na segurança de voo.

Ainda assim, é esperado que um pequeno conjunto de fatores seja responsável por uma grande melhoria nos sistemas de segurança uma vez que o maior número de acidentes é causado por fatores em comum, que são poucos. Encontrar falhas na associação de diferentes tecnologias para veículos aéreos autônomos é um desafio. As falhas mais comuns são tipicamente simuladas nos projetos e são realizados testes de diversas naturezas. Isso previne que projetos com erros clássicos se tornem produtos e gerem acidentes. Ainda assim, erros menos comuns passam despercebidos por testes e simulações, gerando infrequentes, mas numerosos casos de acidentes envolvendo veículos aéreos.

1.5 Teste e simulação de falhas em veículos aéreos autônomos

Sistemas embarcados são frequentemente projetados através da definição de modelos executáveis. As ferramentas mais utilizadas para essa definição são o Matlab/Simulink e o Scilab/Xcos. Modelos Simulink são comumente utilizados tanto no meio acadêmico quanto em grandes empresas de aviação. Eles não só representam uma documentação do projeto mas também são utilizados para teste e simulação. Possuindo papel nuclear no processo de desenvolvimento, sua correção é fundamental. Soluções distintas existem para a definição de testes apropriados para avaliar esses modelos, mas apenas algumas soluções parciais abordam uma qualidade desejada dos resultados das simulações, definindo oráculos adequados. Baresi [Nardi, 2017] propõe uma linguagem formal para a especificação de oráculos e para relacioná-los a modelos existentes. Também apresenta Apolom, uma ferramenta para a verificação de resultados de simulações perante oráculos declarados, obtendo resultados interessantes em termos de efetividade, eficiência e custo computacional.

Testes e simulações são importantes para que se entenda o comportamento geral de uma sistema antes da construção de protótipos, apresentando ganhos significativos

em tempo e outros recursos. Falhas conhecidas podem ser então simuladas de modo a perceber o comportamento do modelo sob tais circunstâncias. À medida em que a complexidade do projeto aumenta, contudo, aumentam-se as dificuldades em garantir sua correteza. Métodos tradicionais de análise que incluem testes e simulações possuem uma cobertura de falhas muito baixa, uma vez que o número de comportamentos possíveis em um sistema aumenta exponencialmente enquanto seu tamanho aumenta linearmente [Campos, 1996]. O Simulink incorpora a ferramenta Simulink Design Verifier que gera testes de sistema com o objetivo de maximizar a cobertura de falhas com o menor número de testes e simulações. Esta ferramenta e outras semelhantes, contudo, são limitadas por não serem exaustivas e não garantem que todas as falhas sejam encontradas.

Testes são feitos postulando-se hipóteses sobre a execução do sistema e simulando seu comportamento em um cenário no qual tal hipótese ocorra, verificando o comportamento do sistema em relação esperado. Nessa metodologia as hipóteses sendo postuladas necessitam ser explicitamente especificadas. Como o número de comportamentos possível é muito grande, essas hipóteses comumente cobrem um percentual pequeno das possibilidades de execução: as que são executadas mais comumente. Por exemplo, os projetistas do Ariane 5 [Gleick, 1996, Lions, 1996] proveram o sistema de tolerância à falhas de hardware através de módulos duplicados de cálculo, mas não previram que se o software causasse o erro o mesmo seria calculado por ambos módulos ao mesmo tempo, tornando a redundância inútil.

Especialmente em sistemas críticos a demonstração de que todos os requisitos tenham sido contemplados por um conjunto de testes baseados neles é comumente mandado por processos internos ou padrões externos. Métodos tradicionais de cobertura, entretanto, não assinalam esse mandato porque eles medem apenas a cobertura do projeto através da determinação de quais caminhos no projeto foram executados. Determinar se um conjunto dado de vetores de testes cobrem os requisitos de projetos, em oposição a simplesmente cobrir o projeto, é um desafio. Friedman [Friedman, 2014] apresenta uma abordagem para a geração de vetores de teste baseados em requisitos que habilita engenheiros a determinar se seus projetos são suficientemente cobertos por um conjunto de vetores de teste baseados em requisitos. Ainda, apresenta que partes do projeto não cobertas carecem de investigações mais elaboradas para determinar se os elementos de projeto deveriam existir ou se há um problema com os requisitos.

Enquanto técnicas de teste e simulação são facilmente computáveis e cobrem os estados de operação mais frequentes, os casos excepcionais ainda ocorrem. Por ocorrerem em situações muito específicas são difíceis de serem descritos e especificados, sendo mais facilmente atingidos por técnicas com cobertura exaustiva. Essas técnicas, entretanto, comumente possuem custos computacionais elevados. Trabalhos que apresentam

soluções que as aplicam são apresentados na próxima seção como forma de estabelecer um ponto razoável entre cobrir tais estados em um tempo factível.

1.6 Trabalhos relacionados a novas soluções para a detecção de falhas em veículos aéreos autônomos

A utilização de técnicas que sejam capazes de analisar exaustivamente os estados possíveis de um sistema complexo se faz necessária onde haja interação entre diversos componentes em casos como o do Voo 447, situações de compatibilidade entre componentes em hardware e software como o caso do foguete Ariane 5 [Gleick, 1996, Lions, 1996] e problemas causados por relações temporais, como a falha da interceptação do míssil Patriot [Subcommittee on Investigations and Oversight, 1992]. Métodos formais têm como característica explorar exaustivamente todas as possíveis execuções de um sistema sendo avaliado. Desta forma, os resultados obtidos são garantidos, ao contrário de testes e simulações, que exploram somente uma parcela dos comportamentos possíveis. Algoritmos e estruturas de dados eficientes garantem que as ferramentas modernas sejam capazes de analisar sistemas de complexidade industrial.

Métodos formais ainda não são suficientemente explorados na indústria aeronáutica. As ferramentas existentes frequentemente têm uso difícil, ou utilizam linguagens específicas e tradutores para estas linguagens ainda não estão amplamente disponíveis. Desta forma a indústria aeronáutica ainda não se aproveita do potencial que métodos formais têm de reduzir tempo e custos no desenvolvimento de sistemas identificando erros de forma eficiente ainda durante a etapa de projeto. Estudos como o NASA/CR-2014-218244 [Miller, 2014, Miller,] apresentam o uso de métodos formais como nucleares na verificação de projetos de sistemas aeronáuticos.

A escolha do conjunto de métodos formais a serem aplicados para a verificação de um projeto requer conhecimentos específicos da área, que é pouco explorada entre os projetistas. Diversos métodos formais diferentes com capacidades e utilizações diferentes podem ser aplicados. Os métodos formais são diversos como os provadores de teorema, extremamente poderosos e de utilização fundamentalmente manual, de forma que se exija do projetista conhecimentos específicos da ferramenta. Outras técnicas menos poderosas, mas automáticas na execução como a interpretação abstrata ou a avaliação simbólica de trajetórias exigem menor conhecimento específico, mas possuem também menor poder na capacidade de expressão das propriedades a serem verificadas. Atenção

especial dedica-se ao método da verificação de modelos, uma das técnicas de verificação formal mais utilizadas atualmente por ser extremamente eficiente e automática, exigindo menos conhecimento do projetista sobre o processo de verificação e ainda assim possuindo uma capacidade completa de cobertura dos estados de um sistema. Uma das ferramentas mais utilizadas no mundo desta técnica é NuSMV [Cimatti, 2002] e sua extensão NuXMV [Bozzano, 2020], desenvolvida na Itália a partir da implementação original da ferramenta SMV de Carnegie Mellon [Mcmillan, 2000].

A modelagem e verificação de um sistema aeronáutico utilizando verificação de modelos, que consiste em diversos processos concorrentes controlando um avião utilizando SMV foi apresentado por Campos [Campos, 1996]. Neste trabalho as propriedades a serem verificadas versam sobre o prazo de execução das tarefas concorrentes, que em hipótese alguma podem exceder o tempo previsto. Alino [Alino, 2016] gerou uma unidade de software para missões aeronáuticas complexas com diretrizes baseadas em métodos formais. Sua ênfase foi na fase de pouso devido à natureza crítica da reação a falhas. Esse trabalho demonstrou a possibilidade de se descrever leis operacionais em linguagens de verificação formal, especificamente para a NuXMV. O estudo destaca que resultados, entre os métodos de prova de teorema, verificação de modelos e interpretação abstrata, possuem melhor desempenho computacional a partir do uso de verificação de modelos. Ainda utilizando o NuXMV sobre outro domínio, Vilhena [Vilhena, 2018] apresenta uma solução para verificação de modelos de controle de sistemas dinâmicos, modelando e verificando um controlador PID. Seus resultados demonstram a cobertura exhaustiva do modelo a partir da discretização da tripla das constantes de controle. Outra contribuição foi a demonstração da capacidade da modelagem em encontrar valores de tais constantes relacionadas a um comportamento esperado do controlador a partir de contraexemplos.

Morzenti [Morzenti, 1991] apresenta um estudo de especificações formais executáveis no suporte da validação e prototipação de sistemas de tempo real. Através da lógica temporal de primeira ordem TRIO, ele apresenta dois algoritmos que decidem a satisfabilidade de fórmulas que consideram distância em tempo e duração de intervalos de tempo a partir de verificação de modelos. Ainda utilizando a lógica TRIO, Morzenti [Morzenti, 2003] utiliza a ferramenta SPIN [Spinroot.com, 2020] para o problema de *Railway Crossing* [Newrailwaymodellers.co.uk, 2020], apresentando cobertura completa do problema clássico na área de verificação de modelos. Ferreira [Ferreira, 2016] utiliza a Verificação Probabilística de Modelos (PMC, do inglês *Probabilistic Model Checking*) para a verificação de modelos de tráfego de veículos autônomos integrados a uma rede de semáforos. Ele considera o fluxo de tráfego, redes computacionais e propagação a rádio abordando o não-determinismo do espalhamento das mensagens. Ele apresenta instruções para a construção de ferramentas e interfaces para a automação da criação de tais

modelos e demonstra seu funcionamento através de um modelo simplificado.

Outro método formal de interesse é a análise estática. Análise estática via interpretação abstrata consiste em um conjunto de técnicas que podem ser usadas para responder perguntas sobre o comportamento de um programa. A análise estática de código é um mecanismo de verificação formal menos preciso que a verificação de modelos, porém muito mais rápido. Um exemplo de propriedade que a interpretação abstrata verifica é a possibilidade de perda de precisão em operações aritméticas, além da verificação de violações como a divisão por zero, acesso fora de memória alocada, dereferência de ponteiro nulo e vazamento de informação sigilosa.

Outra vantagem advinda da interpretação abstrata é a possibilidade de aplicá-la sobre código binário, de baixo nível. Existem diversas ferramentas que se prestam a esse propósito. Por exemplo, *Astreé*, um interpretador abstrato, foi utilizado para verificar o código binário produzido para vários sistemas da empresa francesa Air Bus [Cousot, 2009]. Além disso, *CompCert* [Leroy, 2009], um gerador de código Francês, é capaz de provar formalmente a corretude de programas binários que ele produz a partir de código C. *Astreé* e *CompCert* são dois exemplos de uma direção comum para onde marcha a comunidade de verificação formal. Ferramentas de interpretação abstrata que já foram disponibilizadas por pesquisadores da UFMG como *FlowTracker*[Rodrigues, 2016, Rodrigues, 2020], *Psyche-c*[Melo, 2018, Melo, 2020], *DawnCC* e *SIoT1*[Guimarães, 2017, Guimarães, 2020]. Todas estas ferramentas exploram técnicas de interpretação abstrata para detectar erros e vulnerabilidades em programas.

Enquanto encontramos diversos métodos para verificação dos sistemas de controle aeronáutico de forma completa, a interface entre as áreas de engenharia e de verificação formal se mostra muito pouco desenvolvida. Emerge então a necessidade de ferramentas que, de forma automática e que não requeiram do projetista conhecimentos específicos em verificadores, sejam desenvolvidas. Na próxima seção propomos uma técnica para se traduzir automaticamente projetos em modelos verificáveis, realizando de forma encapsulada a transição entre os domínios.

1.7 Tradução combinada à verificação formal como solução para a detecção de falhas em veículos aéreos autônomos

Os problemas apresentados motivam a realização deste trabalho uma vez que demonstram que mesmo havendo ferramentas relacionadas à verificação formal de sistemas

autônomos como apresentado por Nardi et al [Nardi, 2017], que gera oráculos de teste para modelos qual Simulink e por Tripakis [Dragomir, 2018] que realiza verificação, o problema continua aberto. Este trabalho desenvolve uma metodologia baseada na aplicação de métodos formais para analisar sistemas de aeronaves remotamente pilotadas. Nesse campo, escolhemos realizar a tradução de projetos gerados na linguagem de descrição de modelos executáveis Simulink, por seu extenso uso no meio acadêmico e industrial. Tradutores automáticos de modelos Simulink são escassos e os poucos disponíveis são de propriedade de empresas possuem custo elevado como o Simulink Design Verifier e soluções da Rockwell Collins, motivando o desenvolvimento de uma ferramenta de código aberto.

Modelos para verificação formal criados manualmente a partir de modelos executáveis de aeronaves autônomas possuem desvantagens em relação a modelos criados automaticamente. A principal desvantagem é que a modelagem manual exige do projetista um conhecimento das ferramentas de verificação, o que a maioria dos projetistas aeronáuticos não possui. Um aspecto extremamente vantajoso da verificação automática em uma ferramenta de código aberto é o uso do mesmo motor de tradução para diferentes linguagens de entrada e saída, adaptáveis de acordo com o interesse da comunidade.

Este trabalho apresenta um esquema automático de tradução de modelos de aeronaves autônomas na linguagem Simulink com correspondência verossímil em uma linguagem de verificação formal. Especial atenção dedica-se ao módulo de leis operacionais da aeronave. Essa ênfase se dá devido a resultados desencorajadores relativos ao tempo de execução da verificação de módulos mesmo pequenos de Leis de Controle [Vilhena, 2018], explicada por sua operação no domínio dos números reais. Além disso, especialistas são capazes de avaliar satisfatoriamente o comportamento do controle de uma aeronave através de simulação, por não ser necessária a sua cobertura completa. Uma demonstração da eficácia do controle das dinâmicas de uma aeronave é apresentada por [Riddick, 2018], onde a falha induzida de atuadores não é sequer percebida pelo piloto devido ao controle robusto. Entretanto para as leis operacionais de aeronaves é necessária e possível a realização de sua verificação com cobertura completa, uma vez que seus parâmetros reais podem ser reduzidos a variáveis de estado discretas. Ainda, a verificação dos comandos gerados através do piloto autônomo pode ser realizada isoladamente, aspecto coberto neste texto pelo tratamento de tais comandos pelas Leis Operacionais. De forma geral, os aspectos da linguagem Simulink não cobertos no processo de tradução são:

1. Não foi realizada prova de adequação do código traduzido. Este trabalho objetiva a tradução funcional dos blocos Simulink como um passo inicial para tradutores

completos;

2. O processo de tradução ignora possíveis problemas de sincronização. Os sistemas de Leis Operacionais são compostos por elementos lógicos e de controle que, utilizados com as configurações padrão do Simulink não geram problemas dessa natureza;
3. A funcionalidade dos blocos foi traduzida considerando seu comportamento esperado e não a sua implementação na ferramenta Simulink. Enquanto isso reduz a adequação do código traduzido ao código fonte, temos a ferramenta Simulink com código fechado, não possuindo acesso à sua implementação para uma tradução exata;
4. O quantitativo de blocos traduzidos é limitado aos contidos nos projetos utilizados neste trabalho. A tradução de todas as bibliotecas Simulink é impraticável, devido a sua extensão ampla e crescente. Como solução oferecemos contato aos usuários no sentido de acolhermos demandas de novas implementações;
5. A falha de componentes como sensores e falhas de comunicação não são consideradas. Elementos estocásticos podem ser modelados em trabalhos futuros;
6. Dinâmicas físicas e elementos contínuos de controle não foram modelados uma vez que não há publicações que apresentem a verificação de modelos como uma vantagem em relação a técnicas clássicas de ajustes de tais comportamentos.

Para o processo de tradução, a ferramenta deve abstrair uma estrutura intermediária conversível em diversas linguagens a partir da análise sintática e semântica do modelo, com sua verificação léxica não sendo realizada uma vez que tais modelos são construídos diretamente na ferramenta Matlab/Simulink, sendo isentos de erros léxicos. A relação de correspondência entre o código Simulink e a linguagem de verificação formal apresentada considera modelos pré construídos dos blocos mais comuns em Simulink para a representação de sistemas de controle VANT, como os apresentados no Apêndice A. Um exemplo de bloco de controle PID pré construído modelado em código objeto NuXmv foi apresentado por [Vilhena, 2018]. Uma ferramenta que se proponha a verificar a corretude de modelos de sistemas dinâmicos deve ter a capacidade de se mostrar assertiva em seus resultados positivos (aqueles em que declaram o modelo como correto) e na detecção de erros. Além disso, o tempo de verificação e o espaço de alocação de memória devem ser praticáveis. O método de validação da ferramenta é apresentado no Capítulo 4.

Este trabalho desenvolve a tradução de Leis Operacionais em sistemas de aeronaves e a construção de modelos genéricos e parametrizáveis de blocos de Leis Operacionais em

Simulink que possam ser utilizados para testes de ferramentas para verificação de sistemas de aeronaves. Realizamos um estudo das leis operacionais mais comuns aplicáveis a um voo de média distância em um veículo híbrido VTOL/Asa fixa de modo a atender às demandas de projetos de mobilidade urbana através de VANTs como o Uber Elevate [Jeff Holden, 2016]. Contemplamos leis operacionais aplicáveis tanto ao manual de voo de aeronaves especificadas de modo parametrizável para que se seja capaz de modelar projetos distintos quanto a leis operacionais aplicáveis ao conforto do passageiro e a especificações de um plano de voo singular. Para tanto, os modelos utilizam uma estrutura com inserção facilitada de regras e com uma documentação didática de sua parametrização. Para cada regra descrita é gerada sua modelagem em uma linguagem de verificação formal para a verificação de especificações sobre os modelos construídos como tradução de blocos Simulink.

No que toca à ferramenta de verificação formal escolhida, adotamos a NuXMV para a verificação de modelos. A técnica de verificação de modelos foi adotada por possuir uma cobertura compatível com problemas dessa natureza e possuir um grande poder de verificação dentre as técnicas totalmente automáticas. Ela possui a capacidade de descrever propriedades de software que permitem expressar a grande maioria das propriedades necessárias para uma verificação completa de um sistema complexo. Através da verificação dos sistemas de controle de aeronaves desenvolvemos soluções para os Problemas de Segurança (Seção 1.2) no que diz respeito à prova do funcionamento correto dos sistemas de controle.

1.8 Criação de um conjunto de modelos sintéticos de aeronaves autônomas

Ao gerar ferramentas para verificação de modelos de aeronaves autônomas percebemos a escassez de modelos para testes das mesmas. Modelos sintéticos são usualmente muito generalistas e necessitam de um nível alto de parametrização, o que requer certa especialização. Por possuírem alto nível de parametrização, possuem em seu projeto diversos aspectos redundantes para se adaptarem a diferentes usos. Isso faz com que boa parte da descrição dos modelos seja código volumoso e inútil para o seu comportamento. Modelos reais, de aplicação industrial são protegidos por direitos de propriedade intelectual, não sendo facilmente acessíveis. Este trabalho também apresenta no Capítulo 3 um esquema geral para a construção de modelos sintéticos específicos com aplicabilidade em ferramentas de verificação de modelos de aeronaves autônomas. Nele, apresentamos um conjunto de três modelos completos: Um modelo de um míssil utilizado como referência

ouro; Um modelo gerado a partir da estrutura de abstração da referência ouro utilizada; Um modelo gerado por um especialista, adaptado à estrutura de abstração desenvolvida. Os dois últimos modelos trazem 10 blocos intercambiáveis, o que permite a montagem de diversos outros modelos a partir de sua combinação. O desenvolvimento desse conjunto de modelos tornou possível o estudo e desenvolvimento da ferramenta de tradução e é elemento nuclear deste trabalho. O conjunto de modelos implementados não representa os seguintes aspectos de projetos reais de aeronaves:

1. A implementação das dinâmicas das aeronaves foi simplificada descartando elementos importantes de controle e físicos para que se limitasse a simulações verossímeis. Isso pode ser feito uma vez que tais dinâmicas sequer são traduzidas pela ferramenta implementada;
2. Os modelos não implementam situações de falha e outros elementos estocásticos, uma vez que esse aspecto não foi o foco da abordagem inicial deste trabalho;
3. Os modelos não utilizam processadores para máquinas de estado implícitas como o apresentado no Capítulo 3.

Capítulo 2

Fundamentação Teórica

Este capítulo traz as bases teóricas nas quais se fundamenta este trabalho. Os conceitos apresentados pertencem a três grandes áreas. A primeira refere-se ao projeto de sistemas em ferramentas de blocos executáveis, mais especificamente o Matlab/Simulink. Nela analisamos a sintaxe utilizada, sua aplicação na indústria, como as informações de projeto são processadas pela ferramenta e quais os recursos mais utilizados em projetos de controle de sistemas aeronáuticos. A segunda refere-se ao projeto de veículos aéreos não tripulados, trazendo os principais conceitos associados, a definição de como ocorre o fluxo de informação do comando à atuação e como leis de controle e leis operacionais impactam em seu projeto. A terceira e última área refere-se à detecção de falhas em modelos, apresentando os formalismos lógicos utilizados nos principais trabalhos relacionados, motores para verificação de modelos, técnicas que os utilizam e por fim as ferramentas que foram utilizadas neste trabalho.

2.1 Ambiente de projeto Matlab/Simulink

O Simulink é uma ferramenta comercial pertencente à Mathworks, sendo um ambiente de desenvolvimento de projetos baseado em modelos de amplo uso tanto no meio acadêmico quanto em grandes empresas de aviação. Sua popularidade se deve à amplitude de domínios disponíveis em suas bibliotecas que representam sistemas desde elétricos, hidráulicos, mecânicos até máquinas de estado e controle de sistemas. Sua interface oferece uma linguagem de programação visual, que se tornou um método comum para o desenvolvimento de software embarcado, especialmente para sistemas de controle [Postma, 2015]. A construção de um sistema completo através de subsistemas torna o modelo como um todo mais compreensível, manutenível, hierárquico e modular. No Simulink os blocos representam esses sistemas integrados assim como funções básicas de-

sempenhadas pelo sistema. Os blocos são conectados através de sinais que representam os dados. Através dessa linguagem de programação visual, o Simulink oferece recursos para teste e simulação atrelados à geração automática de código facilitada pela disponibilidade de um grande número de ferramentas disponíveis. Ele é aplicado no desenvolvimento de sistemas embarcados de diversos domínios [Wassyng, 2018]. Outra ferramenta que possui função similar e que é não comercial é a Scilab/Xcos, pertencente a The Scilab Consortium. Ambas as ferramentas definem um padrão para o projeto e simulação de sistemas embarcados em diversos domínios como aviação, automobilística e telecomunicações. Entretanto o Simulink é comumente utilizado devido a melhor suporte e maior extensão das bibliotecas de blocos.

A linguagem do Simulink se estrutura através de blocos que representam entidades do fenômeno que se está modelando. Esses blocos são tratados como funções que transformam sinais recebidos através de linhas conectadas à saída de outros blocos e sincronizados a partir de uma contagem de ciclos. As conexões das entradas de um bloco são representadas por setas que apontam para um conjunto de posições ordenadas, que transportam os valores de sinal para o processamento do bloco. De forma análoga, as conexões das saídas de um bloco são representadas por setas que possuem como origem a posição ordenada daquela saída do processamento do bloco. Esse processamento é realizado tanto por funções descritas na linguagem nativa do Matlab quanto por subsistemas Simulink.

Um bloco de subsistema, *Subsystem*, é utilizado para representar sistemas dentro de sistemas, permitindo uma modelagem hierárquica [Wassyng, 2018, Postma, 2015]. Um bloco *Subsystem* possui portas de entradas, *Inports*, e de saídas, *Outports*, que representam ligações explícitas de entrada e saída do bloco. As dependências de dados internos ao subsistema como *Data Stores* e pares *Goto/From* são denominadas interfaces implícitas. O uso de pares *Goto/From* quebra o fluxo de dados, que é feito principalmente através de linhas, gerando problemas de referência na documentação do projeto por não tornar explícito o caminho de dados utilizado (também para analisadores semânticos, como tratado no Capítulo 4). Os símbolos esquemáticos dos blocos que representam os mecanismos para fluxo de dados no Simulink são apresentados na Figura 2.1.

Data Stores são unidades análogas a variáveis em linguagens de programação tradicionais, servindo como unidades de memória [Wassyng, 2018, Postma, 2015]. Elas são importantes para o compartilhamento de dados entre subsistemas e modelos referenciados através do bloco *Model*. Uma unidade *Data Store* é definida utilizando o bloco *Data Store Memory* e é referenciada para leitura através de blocos *Data Read Blocks* e para escrita através de blocos *Data Write Blocks*. Seu escopo é o subsistema que a contém, incluindo todos os subsistemas a ele hierarquicamente inferiores (excluindo modelos re-

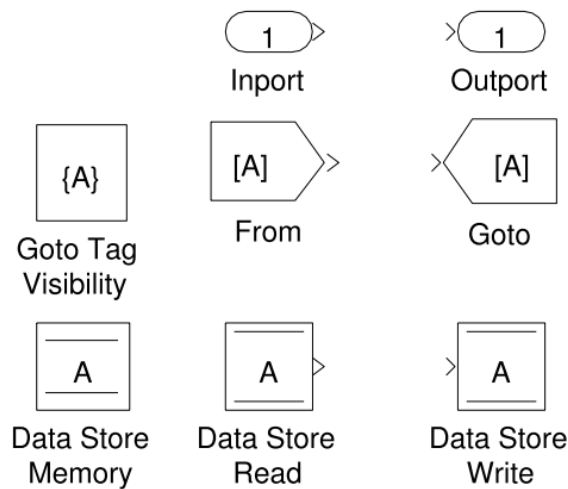


Figura 2.1: Mecanismos para fluxo de dados no Simulink [Postma, 2015].

ferenciados). Uma unidade *Data Store* definida no escopo base é chamada de global e pode ser acessada em qualquer lugar do modelo, inclusive modelos referenciados. O sincronismo entre os blocos é garantido através da contagem de ciclos pelo Simulink. Latências de sinal, tempos de leitura e escrita e outras características temporais são definidas diretamente em cada bloco. Diversas leituras e escritas podem ocorrer em uma mesma unidade *Data Store*, gerando problemas de concorrência como:

Read-Before-Write: Uma leitura ocorre num passo de tempo antes de uma escrita ter ocorrido, gerando problema de latência pela leitura do dado antigo;

Write-After-Read: Uma escrita acontece depois de uma leitura ter ocorrido em um passo de tempo, gerando problema de ambiguidade quanto ao valor correto requerido para a execução (Figura 2.2);

Write-After-Write: Uma escrita ocorre duas vezes em um mesmo passo de tempo sem haver leitura, gerando problema de perda de dados.

Com relação ao desvio de fluxo implícito de dados, que se refere a caminhos de dados não explícitos através de linhas, o Simulink utiliza o mecanismo *Goto/From* [Wassyng, 2018, Postma, 2015]. O dado inserido em um bloco *Goto* é transmitido aos blocos *From* correspondentes (aqueles que possuem o mesmo rótulo, *Tag*) sem o uso de uma linha de sinal entre eles (*Line*). O escopo de um bloco *Goto* é determinado por seu parâmetro *Tag Visibility*, que pode assumir os valores:

Local: Os blocos *Goto* e *From* de mesmo rótulo se encontram no mesmo subsistema;

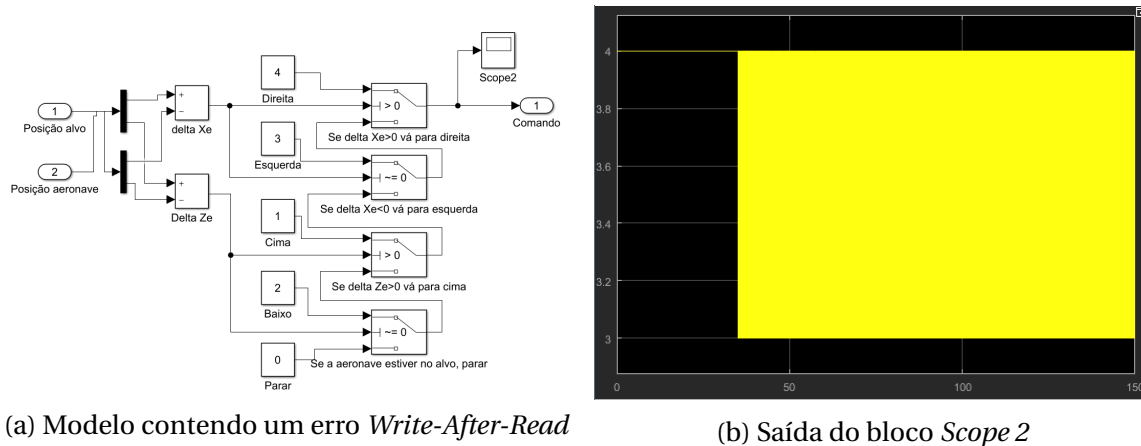


Figura 2.2: Exemplo de um modelo que contém um erro *Write-After-Read*. Nele, a variável *Comando* é utilizada para determinar as dinâmicas da aeronave que determinam o novo estado da variável *Posição aeronave*. Como não existe garantia de sincronismo entre o cálculo do novo estado e o cálculo de *Delta Xe* e *Delta Ze*, os deltas são sempre calculados com o valor antigo da *Posição aeronave*, gerando uma inconsistência na geração do comando para a aeronave, apresentado na visualização do *Scope2*. Uma solução para o problema é apresentada na Seção 3.4 do Capítulo 3 (Figura 3.19).

Scoped: O escopo do bloco *Goto* é aquele de seu bloco *Goto Tag Visibility* e subsistemas hierarquicamente inferiores;

Global: Os blocos *Goto* e *From* possuem todo o sistema como escopo.

Entendendo o fluxo de dados no Simulink podemos entender como as variáveis são geradas e transformadas através de funções (blocos) de modo a converter seu código em outras linguagens de propósitos distintos. Esse é o primeiro passo para que se possa verificar, através do método de verificação de modelos, projetos como os do controle de veículos aéreos não tripulados de forma automática. Os Capítulos 3 e 4 apresentam respectivamente a implementação de modelos Simulink seguindo os padrões de projeto recomendados e um método automático para a sua tradução.

2.2 Projeto de Veículos Aéreos Não Tripulados

Um Veículo Aéreo não Tripulado (VANT), no inglês Unmanned Aerial Vehicle (UAV), é uma aeronave tipicamente militar que é guiada autonomamente ou por controle remoto e que carrega sensores, designações de alvo, ordens ofensivas ou transmissores eletrônicos projetados para interferir ou destruir alvos inimigos [Britannica Academic, 2015]. É considerada uma arma importante desde a década de 1980 por seu tamanho reduzido e motores silenciosos. Livre de tripulação, sistemas de apoio à vida e projetos de segurança

específicos a aeronaves tripuladas, oferecem maior eficiência e alcance quando comparados a sistemas tripulados equivalentes. São a evolução de veículos remotamente pilotados (Remotely Piloted Vehicles - RPVs) que foram amplamente utilizados nas décadas posteriores à Segunda Guerra Mundial. Ozsoy [Ozsoy, 2017] também apresenta aplicações militares como sombreamento de tropas inimigas; captura de mísseis pela emissão de sinais artificiais; retransmissão de sinais de rádio; proteção de portos de ataques do alto mar; situações de guerra; reconhecimento; vigilância da atividade inimiga; monitoramento de contaminação nuclear; biológica ou química; designação e monitoramento de alvos; localização e destruição de minas terrestres; eliminação de bombas ativas; sabotagem e destruição de sistemas de radar; segurança e avaliação de danos de bases aéreas.

Aplicado ao contexto civil, VANTs são definidos como veículos aéreos energizados sustentados em voo por aerodinâmica e guiados sem um piloto a bordo [Ozsoy, 2017]. Eles podem ser descartáveis ou recuperáveis e podem voar de forma autônoma ou através de um piloto remoto. VANTs são comumente utilizados no meio civil para fotografias aéreas; monitoramento de rebanhos; serviços de contenção e detecção de incêndios; vigilância de tráfico ilegal de produtos; inspeção de redes elétricas de distribuição; monitoramento de trânsito; busca de pessoas e monitoramento de recursos hídricos.

Um VANT totalmente autônomo é aquele que consegue voar sem intervenção de um operador da decolagem até o pouso. Alguns tipos de aeronaves são operadas por controle remoto com constante envolvimento de um piloto externo. Para alcançar requisitos de controle, diferentes estratégias de controle tem sido aplicadas, incluindo o controlador proporcional-integral-derivativo (PID), redes neurais adaptativas, lógica difusa e controladores de ordem fracionária. Um estudo da aplicação de diferentes técnicas de controle de sistemas dinâmicos em um projeto de aeronave foi apresentado por [Riddick, 2018]. O primeiro tipo de controle, PID, foi o escolhido para modelagem neste trabalho devido à sua ampla aplicação nos modelos de controle de aeronave encontrados, sendo utilizado na maior parte dos pilotos automáticos comerciais [Ozsoy, 2017].

O projeto de VANTs consiste em um trabalho interdisciplinar complexo que possui um grande número de requisitos que se comportam de forma integrada e que são potencialmente conflitantes. Não raramente esses requisitos são obtidos por engenharia reversa ou através de projetos legados que podem conter informações incompletas. Outros requisitos são constantemente atualizados ao longo do projeto, inclusive em etapas finais. A mudança em um parâmetro como o peso da aeronave pela adição ou remoção de um componente, por exemplo, impacta no projeto como um todo por interferir em todo o conjunto de suas dinâmicas físicas e equações relacionadas.

Ambientes de projetos que cobrem diversos domínios através da descrição de blocos executáveis para simulação são amplamente utilizados em projetos de veículos autô-

nomos. Eles concentram as diversas decisões de implementação, simulações multifísica, a organização do projeto de uma forma geral e a leitura detalhada de seus subsistemas. Elementos importantes a serem modelados são as referências de comando do piloto, seja ele embarcado ou autônomo, modeladas como Alvo; o funcionamento físico da Aeronave (com a modelagem de seus sensores e atuadores); as Leis Operacionais do voo, que são aquelas que dizem sobre restrições da operação da aeronave em alto nível, como as descritas no manual de voo; as Leis de Controle dos atuadores como sistemas dinâmicos.

De forma geral, a representação de uma Aeronave (referenciada comumente por *airframe* quando diz respeito às dinâmicas de sua fuselagem) através de um diagrama de blocos contempla seus atuadores (como motores, superfícies de controle, portas automáticas, válvulas dentre outros) e conjunto de sensores (sensores de posição, ângulo, temperatura, estado de bateria, pressões interna e externa, entre outros). Os atuadores são comandados através de formas de energia com entrada controlada pelo módulo de leis de controle (injeção de combustível, níveis de tensão, comandos de pulso, entre outros). O estado dos atuadores é então atualizado por modelos matemáticos que representam seu comportamento físico. Informações sobre o estado dos atuadores são fornecidos por sensores a eles relacionados a todo o sistema de controle, bem como a sistemas de visualização como uma percepção do estado atual do sistema. Outros sensores também acoplados aos sistemas de controle e visualização são responsáveis pela medição de variáveis relativas ao ambiente (situação interna e externa à aeronave, situação térmica de componentes, velocidade e direção de vento, entre outras).

A representação do sistema de visualização através do diagrama de blocos representa a leitura dos sensores com seus valores físicos e a sua conversão em uma representação compreensível pelo piloto. Seja o piloto manual (remoto ou embarcado), essa conversão se dará em uma representação textual e gráfica em alto nível. Seja o piloto autônomo, essa conversão se dará em uma representação compatível com os parâmetros das funções que utilizem a leitura dos sensores. Em ambos os casos, o piloto gera comandos em alto nível para o sistema de Leis Operacionais para tratamento.

O bloco de Leis Operacionais recebe os comandos dos pilotos e decide sobre executá-los quando possível com ou sem restrições, adiá-los se necessário ou ainda ignorá-los quando esses representam violações de regras. Para tal decisão são considerados valores recebidos a partir do sensoriamento da aeronave. Quando os comandos são executados, o sistema de Leis Operacionais decide sobre quais elementos do sistema de Leis de Controle ativar e quais serão os seus valores de referência. As regras constantes nas Leis Operacionais representam restrições sobre a violação de normas de segurança, melhoria na experiência de usuário, gestão de recursos de tempo e energia entre outros aspectos inerentes àquela aeronave como operações específicas e descrições do manual

de voo.

O bloco de Leis de Controle recebe os valores de referência do sistema de Leis Operacionais juntamente com os valores instantâneos recebidos do sensoriamento da aeronave e os processa de modo a gerar comandos para os atuadores da aeronave. Cada atuador é comandado por um conjunto de controladores, cada qual ativo em um determinado estado do avião (controlado pelas Leis Operacionais). Esses controladores representam operações de ganho, deslocamentos, relações proporcionais, relações integrais e relações derivativas, dentro da teoria de Controle de Sistemas e fornecem aos atuadores a quantidade de energia calculada, fechando assim o ciclo da informação do comando à atuação.

2.3 Detecção de falhas em modelos

A detecção de falhas em modelos em projetos de sistemas complexos é fundamental para a sua correção antes da etapa de prototipação. Ela permite a aplicação de ciclos mais curtos e com menor custo de levantamento de requisitos, testes e correções, anteriores ao desenvolvimento de processos de custo elevado. A detecção de falhas ainda é importante para o mapeamento de casos de exceção, não previstos nas etapas anteriores de projeto. O quantitativo de casos de falhas mapeados por um método de detecção é chamado de cobertura da detecção. Em geral métodos de cobertura incompleta, que lidam com os casos de falhas mais comuns, possuem custo computacional baixo, apesar de necessitarem de maior especialização no domínio em análise. Métodos de cobertura completa são em geral generalistas, possuindo portanto um custo computacional mais elevado.

O teste de sistemas é uma técnica utilizada para o aumento de confiabilidade através da cobertura de casos frequentes. É importante notar que a cobertura de técnicas de teste não é completa, mas suficiente para sistemas não críticos e aplicável a alguns sistemas críticos [Friedman, 2014]. Para o teste de sistemas, engenheiros tipicamente empregam metodologias baseadas em requisitos e/ou cobertura para a geração de vetores de testes que podem objetivar a cobertura do projeto ou dos requisitos do projeto (sendo assim mais completa e possuindo maior complexidade de tempo). Em testes orientados a requisitos, um conjunto de vetores de teste e de saídas esperadas é desenvolvido baseado nos requisitos do sistema. A técnica de testes de modelos, além de exigir a identificação apropriada de casos de teste, exige a definição de instrumentos adequados para a asserção da correteude dos resultados obtidos. A identificação de tais instrumentos é conhecida como o problema de oráculo (*oracle problem*) [Nardi, 2017]. Um oráculo especifica o que um sistema deve realizar. Dadas as entradas ele deve afirmar a saída esperada do modelo. Para evitar que o oráculo seja complexo e susceptível a erros como o próprio sistema,

deve-se defini-lo através de abstrações em nível mais alto.

O uso de métodos formais parte de princípios matemáticos para demonstrar a correteza de um sistema. São necessários três componentes para especificar-se e verificar-se um sistema: Um modelo matemático para o comportamento do sistema; uma maneira de especificar-se as propriedades que se deseja verificar; e uma metodologia para provar que a propriedade é matematicamente verdadeira neste modelo. A técnica de verificação de modelos é um método formal para a análise automática e exaustiva de um modelo em respeito a uma especificação [Ferreira, 2016]. A verificação de modelos é aplicada em sistemas de diversos domínios como projetos de hardware, padrões de comunicação, projetos de circuitos, protocolos de rede, controle de sistemas autônomos, sistemas biológicos, sistemas em teorias dos jogos e sistemas críticos. A especificação é uma definição lógica que modela condições a serem atendidas pelo sistema modelado. Uma especificação pode ser representada através de lógicas proposicionais, temporais e probabilísticas, dentre outras. A verificação formal é comumente aplicada na modelagem de problemas críticos, em que a aplicação de testes ou simulações, que possuem custo computacional consideravelmente menor, não é suficiente dado que elas não possuem cobertura completa do modelo. Exemplos de sistemas críticos são sistemas de controle de veículos autônomos, cuja verificação motivou este trabalho.

Diversas ferramentas estão disponíveis para a detecção de falhas em modelos. Nesta seção destacamos as principais aplicadas ao contexto da verificação de projetos de veículos autônomos. Essas ferramentas associam diferentes técnicas para o processo de verificação, muitas vezes com pequenas alterações em seus algoritmos ou ainda associando técnicas para gerar técnicas híbridas. Apresentamos também as principais técnicas utilizadas, bem como seus motores, que são os métodos formais utilizados para processamento das etapas de verificação. Ainda, apresentamos os principais formalismos lógicos que descrevem os modelos em uma linguagem de entrada para os motores.

2.3.1 Formalismos lógicos

Os tipos de lógica são diversos e incluem a álgebra booleana, a lógica modal, a lógica epistêmica, a lógica do conhecimento [Ferreira, 2016], as lógicas de primeira (lógica de predicados) e segunda ordem, a lógica temporal e a lógica probabilística. Cada tipo de sistema possui uma lógica mais adequada correspondente, para a qual sua modelagem se torna melhor abstraída. Sistemas paralelos e reativos, como sistemas de controle de veículos autônomos, dependem de uma análise de uma sequência de eventos, sendo mais adequada a eles a modelagem através da lógica temporal [Ferreira, 2016]. Dentre as diversas lógicas relevantes e representativas em seu domínio para contextualização, aplicamos a

lógica temporal neste trabalho devido ao comportamento dependente do tempo dos sistemas dinâmicos que constituem os modelos de aviação modelados.

Lógica temporal (TL do inglês *Temporal Logic*) é adequada para a modelagem de sistemas reativos e paralelos, que demandem a análise de uma sequência de eventos [Ferreira, 2016]. Ela é descrita em proposições lógicas em termos do tempo e de sequências de eventos modelados em um grafo de transições de estado. Existem diversas lógicas temporais, dentre as quais a Lógica Árvore de Computação (CTL, do inglês *Computation Tree Logic*) e a Lógica Temporal Linear (LTL do inglês *Linear Temporal Logic*) são as mais utilizadas. A principal diferença entre as lógicas LTL e CTL está na capacidade da LTL ser capaz de lidar com um caminho computacional infinito como o que é gerado pelo uso de variáveis de domínio infinito, enquanto a CTL lida com uma árvore de caminhos infinitos a partir do nó raiz. O CTL* é um superconjunto de CTL e LTL, sendo capaz de lidar com ambos os escopos e ainda novos que representam a união de suas operações lógicas.

A lógica temporal herda os operadores lógicos da álgebra booleana ($\vee, \wedge, \rightarrow, \neg$). Além disso, a TL utiliza quantificadores da lógica de primeira ordem \forall e \exists , os tratando como quantificadores de caminho para processamento na computação da árvore de decisão binária direcionada que modela a função booleana modelada. Além dos quantificadores da lógica de primeira ordem, a TL utiliza operadores temporais, os quais permitem o processamento na sequência de estados.

Quantificadores de caminho na TL são $A\Phi$, verificando se Φ é verdade para todos os caminhos (e.g. se um avião está voando, invariavelmente ele vai pousar) e $E\Phi$, verificando se existe pelo menos um caminho para o qual Φ é verdade (e.g. se um avião está pousado existe um caminho através do qual ele futuramente estará no ar). Eles são utilizados para representar que todo caminho computacional a partir do estado atual respeitam a propriedade Φ .

Operadores temporais na TL são $F\Phi$, verificando se Φ é verdade em um estado futuro (e.g. um avião em decolagem futuramente entrará no estado de voo em cruzeiro), $G\Phi$, verificando se Φ é verdade em todos os estados futuros (e.g. em qualquer situação a velocidade de um avião será menor do que a Velocidade Não Exceder de seu manual), $X\Phi$, verificando se Φ é futuro no estado seguinte (e.g. se houver um comando de pouso, o próximo estado do avião será híbrido VTOL-Cruzeiro), $\Phi_1 U \Phi_2$, verificando se Φ_1 é verdade até que Φ_2 se torne verdade (e.g. o estado do avião será híbrido VTOL-Cruzeiro até que o avião atinja a posição de pouso ou a altitude de cruzeiro).

Em LTL, quantificadores E não são utilizados, uma vez que existe um único caminho a ser computado. Em CTL, operadores temporais precisam ser precedidos por um quantificador de caminho. A propriedade $EG\Phi$ verifica se Φ existe um caminho onde Φ seja sempre verdade. A propriedade $AF\Phi$ verifica se Φ se torna verdade fatalmente em

todos os caminhos a partir do estado atual. A propriedade $EF\Phi$ verifica se existe um caminho onde Φ se torna verdade fatalmente. A propriedade $AG\Phi$ verifica se Φ é sempre verdade em todos os caminhos que partem do estado atual.

Um operador temporal CTL consiste em um quantificador de caminho e um operador temporal linear. Os quantificadores de caminho são: A - "para todo caminho" e E - "existe um caminho". Os operadores temporais lineares sobre propriedades p e q são: Xp - p é verdade no próximo estado; Fp - p será verdade em algum momento no futuro; pUq - p será verdade até que q seja verdade no futuro.

2.3.2 Motores para verificação de modelos

Nesta seção apresentamos os principais motores para verificação de modelos, SAT e BDD. Eles comumente são agregados em ferramentas de verificação formal tanto como opções em suas formas puras quanto através de soluções híbridas.

2.3.2.1 Satisfabilidade booleana (SAT)

O problema de satisfabilidade booleana (SAT, do inglês *Satisfiability*) consiste em determinar se existe algum assinalamento para uma dada fórmula proposicional que a torne verdadeira ou provar que tal assinalamento inexistente. Esse problema é tratado através de resolvedores de satisfabilidade que implementam algoritmos e heurísticas que calculam sua solução. Resolvedores SAT comumente recebem as fórmulas proposicionais na forma normal conjuntiva (CNE, do inglês *Conjunctive Normal Form*) [Andrade, 2008]. Uma primeira abordagem para se resolver um problema de SAT seria a de testar todos os assinalamentos possíveis. Essa abordagem ingênua possui um custo de tempo computacional muito grande. Davis e Putnam [Putnam, 1960] apresentaram o primeiro resolvidor SAT conhecido na literatura baseado na operação de Resolução. Uma abordagem com desempenho superior à de Davis e Putnam e amplamente utilizada em resolvedores de SAT da atualidade é a do resolvidor de SAT DPLL (Davis, Putnam, Logemann e Loveland) [Loveland, 1962].

Problemas de domínios diversos expressos em lógica de primeira ordem podem ser resolvidos através de motores SMT (do inglês *Satisfiability Modulo Theories*). O SMT consiste no problema das teorias de módulo de satisfabilidade, um problema de decisão para fórmulas lógicas com relação a combinações de teorias de fundo expressas em lógica clássica de primeira ordem com igualdade [Biere & Bloem, 2014].

2.3.2.2 Diagrama binário de decisão (BDD)

A técnica de representação em diagramas binários de decisão (BDD, do inglês *Binary Decision Diagram*) cria uma representação implícita que codifica cada estado como a atribuição de valores booleanos às variáveis do sistema de modo que transições possam ser expressas como dois conjuntos de variáveis: o primeiro representando o estado anterior e o segundo representando o estado atual [Ferreira, 2016]. Essa representação reduz o tamanho da descrição do modelo através da remoção de informações redundantes, tornando possível a modelagem de sistemas maiores utilizando os mesmos recursos computacionais. Além disso, a redução do tamanho do modelo permite a execução eficiente de algoritmos de busca no grafo.

BDDs representam árvores binárias de decisão direcionadas com dois tipos de vértices v : terminais e não-terminais. Vértices não-terminais são rotulados com o símbolo da variável booleana que representam, dado por $var(v)$ e com dois sucessores: $zero(v)$, quando $var(v) = 0$ e $one(v)$, quando $var(v) = 1$. Um vértice terminal é rotulado apenas por 0 ou 1, dado pela função $value(v)$ que possui o valor da função booleana representada dados os valores das variáveis assumidos naquele caminho [Ferreira, 2016].

Árvores binárias de decisão comumente representam informações redundantes, gerando um custo computacional desnecessário. Diagramas binários de decisão ordenados (OBDD, do inglês *ordered binary decision diagram*) representam as funções booleanas através de grafos acíclicos direcionados (DAG, do inglês *directed acyclic graph*). Um OBDD é obtido a partir de uma árvore binária de decisão através da fusão de subárvores idênticas e da eliminação de nós com informação redundante. A representação do OBDD como um grafo direcionado acíclico permite o compartilhamento de subestruturas. Além disso, a canonicidade de um OBDD pode ser obtida uma vez que as suas variáveis possuam o mesmo ordenamento em um caminho de uma raiz a um terminal e que árvores isomórficas não existam dentro da representação [Bryant, 1986].

Um problema clássico sobre BDDs é a escolha da ordem da representação das variáveis da função booleana modelada. O tamanho de um BDD está diretamente associado a esse ordenamento e a escolha do ordenamento que o minimiza é co-NP-completo [Bryant, 1986]. O assinalamento do ordenamento de variáveis em um BDD é comumente realizado de forma empírica, dada a sua relação com a semântica do modelo. Ainda, há aproximações heurísticas que aproximam o problema do ordenamento de variáveis em um BDD [Ferreira, 2016].

2.3.3 Técnicas para a verificação de modelos

Modelos representados através de formalismos lógicos podem ser processados de forma a verificar a satisfabilidade de propriedades a eles relacionadas. Por exemplo, pode-se expressar a propriedade "sempre que um pedido for feito, uma resposta a ele será gerada no futuro", ou "um estado de erro nunca será alcançável". Técnicas distintas são aplicáveis a modelos e propriedades de diferentes naturezas. Representações e algoritmos avançados são capazes de explorar grafos de grandes dimensões de forma exaustiva, determinando se a propriedade é ou não satisfeita pelo mesmo. Nesta seção apresentamos as principais técnicas utilizadas pelas ferramentas analisadas na seção seguinte.

2.3.3.1 Verificação de modelos simbólica

A verificação de modelos simbólica (*Symbolic Model Checking*, como foi definida) foi proposta por Emerson, Clarke e por Sifakis [Clarke, 1980, Emerson, 1982, Sifakis, 1982]. Ela verifica modelos construídos sobre máquinas de estados finitos descritas em uma linguagem específica e propriedades especificadas em lógica temporal, dado um conjunto de estados iniciais. Possuindo recursos como ser completamente automática, cobertura completa do espaço de estados ao percorrer exaustivamente os caminhos no grafo e construção de contraexemplos, a verificação de modelos simbólica é amplamente aplicada. A construção de contraexemplos é especialmente importante para a compreensão do erro e correção do modelo. Como desvantagens, a verificação de modelos simbólica possui a explosão de recursos computacionais devido à exaustão do espaço de estados, exponencial em relação ao número de variáveis. Um método relevante para mitigar esse problema é a aplicação de diagramas binários de decisão, BDD [Mcmillan, 1992, Bryant, 1986], que contudo não remove a exponencialidade do pior caso.

2.3.3.2 Verificação de modelos limitados (BMC)

A verificação de modelos limitados (BMC do inglês *Bounded Model Checking* se mostra relevante em aplicações industriais como em técnicas de automação de projeto eletrônico, através de SAT [Biere, 2009]. A BMC é usada para invalidação de respostas, que encontra violações de propriedades temporais. Outra aplicação relevante da BMC é a geração automática de casos de teste para ampliar a cobertura de casos e refutar redundância nos projetos. A BMC representa um traço de contra-amostra de comprimento limitado simbolicamente e verifica a fórmula proposicional resultante com um solucionador SAT. Se for possível satisfazer a fórmula, sendo o caminho viável, uma atribuição satisfatória retornada pelo solucionador SAT pode ser traduzida em um traço contraexemplo con-

creto que mostra que a propriedade é violada. Caso contrário, o limite é aumentado e o processo repetido. Extensões completas para a BMC permitem interromper este processo em um ponto, com a conclusão de que a propriedade não pode ser violada, esperançosamente, antes que os recursos disponíveis sejam esgotados.

2.3.3.3 Verificação de modelos probabilística (PMC)

A verificação probabilística de modelos (PMC do inglês *Probabilistic Model Checking*) é um método formal, exaustivo e automático para a modelagem e análise de sistemas estocásticos, comumente cadeias de Markov (contínuas ou discretas no tempo) ou processos de decisão de Markov. Com isso, seu comportamento depende apenas de seu estado atual e cada transição entre estados ocorre em tempo real. Dada uma propriedade ϕ expressa como uma fórmula em uma lógica probabilística temporal, a PMC verifica se um modelo do sistema estocástico modelado satisfaz tal propriedade com uma probabilidade maior ou igual à probabilidade limiar θ . Um verificador de modelos probabilístico tipicamente representa o sistema modelado através de um BDD com probabilidades assinaladas às transições entre estados. Uma cadeia de Markov contínua no tempo (CTMC do inglês *Continuous Time Markov Chain*) é um conjunto de um conjunto finito de estados, sendo um deles o estado inicial, uma matriz de probabilidades de transição e uma função de rotulamento que rotula cada estado do sistema através do conjunto de proposições que modelam as propriedades de interesse.

2.3.4 SMV e ferramentas derivadas para verificação de modelos

O SMV (Symbolic Model Verifier) [Clarke, 1996] é uma ferramenta criada em um período em que a técnica de verificação mais amplamente utilizada era a simulação extensiva, com cobertura limitada. Ferramentas como provadores de teorema e verificadores de provas proviam cobertura completa, mas exigiam grandes tempos computacionais e significativa intervenção do usuário.

O SMV implementa verificação de modelos por lógica temporal, representados através de lógica temporal proposicional e com sistemas computacionais representados por grafos de estado-transição. A verificação através dessa técnica realiza uma busca em largura e determina se as especificações são satisfeitas pelo modelo. O uso de algoritmos simbólicos permite a verificação de espaços de estados grandes. Nessa abordagem as relações de transição são representadas implicitamente por fórmulas booleanas e implementados através de BDDs. Uma vantagem relevante dessa abordagem em relação às

predecessoras é a de que o procedimento é completamente automático. O verificador de modelos recebe como entrada uma descrição do modelo e um conjunto de especificações a serem verificadas com fim de determinar se as fórmulas são verdadeiras ou não para aquele modelo, fornecendo um contraexemplo em caso negativo. A modelagem de sistema parcialmente especificados pode ser feita através do assinalamento de valores não determinísticos a suas saídas.

Com o intuito de integrar outras técnicas de verificação a ferramenta NuSMV foi desenvolvida a partir do SMV. Ele estende o SMV através da integração de outras técnicas de verificação de modelos baseadas em SAT além de reimplementar o SMV na direção de um verificador de modelos simbólico estado da arte que lida com múltiplas tecnologias, de código aberto e flexível.

Como uma evolução do NuSMV, o NuXMV [Bozzano, 2020] foi desenvolvido. O NuXMV é uma ferramenta de verificação que herda todas as funcionalidades do NuSMV, expandindo o domínio de análise de estados finitos ao domínio de estados infinitos e outros recursos genéricos. Ele provê a análise síncrona de sistemas com finitos e infinitos estados. Para casos de finitos estados, o NuXMV possui um mecanismo de verificação baseado em algoritmos avançados em SAT. Para os casos com infinitos estados, o nuXmv utiliza técnicas de solução por SMT expandidas em domínio através da associação a técnicas de verificação de modelos limitada (SBMC, *Simple Bounded Model Checking*). O NuXmv possui a capacidade de lidar com tipos de dados infinitos como dados reais e inteiros não limitados, suportando modelos de controle de sistemas no domínio contínuo. As Figuras 2.3 e 2.4 apresentam, respectivamente, o modelo de três leis operacionais de uma operação de pouso e suas propriedades para execução na ferramenta NuXMV.

```

-- VAF = 4 -- [APS-LM-1124]
next(SLPEFF) := SLPEFF;
next(SLOPE) := SLOPE;
next(SLOPEC) := case
| VAF=4 & SLPEFF : SLOPE;
| VAF=4 & !SLPEFF : 0;           -- VERIFY
| TRUE : SLOPEC;
esac;
next(SINFI) := case
| VAF=4 & SLPEFF : {1..10};
| VAF=4 & !SLPEFF : 0;         -- VERIFY
| TRUE : SINFI;
esac;

-- VAF = 5 -- [APS-LM-1125]           -- VERIFY
next(THREVC) := THREVC;
next(RVOFF) := RVOFF;
next(NUMREV) := case
| VAF=5 & (THREVC = RVOFF) : TRUE;      -- All reversers operating
| VAF=5 & (THREVC = !RVOFF) : FALSE;    -- All reversers inoperative
| TRUE : NUMREV;
esac;

-- VAF = 6 -- [APS-LM-1126]           -- auto-brakes is operating
next(ABACC1) := ABACC1;
next(ABACC2) := ABACC2;
next(ABACC3) := ABACC3;
next(ABKSET) := ABKSET;
next(ACCAB) := case
| VAF=6 & ABKOPT & (ABKSET = 0) : -ABACC1;  -- VERIFY
| VAF=6 & ABKOPT & (ABKSET = 1) : -ABACC2;  -- Minimum acceleration value
| VAF=6 & ABKOPT & (ABKSET = -1) : -ABACC3;  -- Intermediate acceleration value
| VAF=6 & !ABKOPT : -9;                    -- Maximum acceleration value
| TRUE : ACCAB;                            -- Default value SEVEN9
esac;

-- VAF = 7 -- [APS-LM-2459]           -- CALL CVWIND
next(VWGL) := case
| VAF=7 : {1..10};
| TRUE : VWGL;
esac;

```

Figura 2.3: Três leis operacionais de uma operação de pouso modeladas para a ferramenta NuXMV [Alino, 2016].

```

-- Linear Temporal Logic SPECS
--   LTLSPEC G ( VAF > 0 & VAF < 8 )    -- verificar se vaf pode ser zero ou 47
|
|
-- Computation Tree Logic SPECS LLR

SPEC AG ( VAF = 4 & !SLPEFF -> AF SLOPEC = 0      )      -- VAF - 4
SPEC AG ( VAF = 4 &  SLPEFF -> AF SLOPEC = SLOPE )      -- VAF - 4

SPEC AG ( VAF = 4 & SLPEFF   -> AF SINFI >= 1 & SINFI <= 10 )  -- VAF - 4
SPEC AG ( VAF = 4 & !SLPEFF -> AF SINFI = 0      )      -- VAF - 4

SPEC AG ( VAF = 5 & THREVC =  RVOFF   -> AF NUMREV )      -- VAF - 5
SPEC AG ( VAF = 5 & THREVC = !RVOFF   -> AF !NUMREV )    -- VAF - 5

SPEC AG ( VAF = 6 & ABKOPT & ABKSET = 0   -> AF (ACCAB==ABACC1) )  -- VAF - 6
SPEC AG ( VAF = 6 & ABKOPT & ABKSET = 1   -> AF (ACCAB==ABACC2) )  -- VAF - 6
SPEC AG ( VAF = 6 & ABKOPT & ABKSET = -1  -> AF (ACCAB==ABACC3) )  -- VAF - 6
SPEC AG ( VAF = 6 & !ABKOPT                -> AF (ACCAB=-9)      )  -- VAF - 6

```

Figura 2.4: Propriedades a serem verificadas pela ferramenta NuXMV sobre as leis operacionais referidas na Figura 2.3 [Alino, 2016].

Capítulo 3

Geração de modelos de sistemas de aeronaves

Um problema comum no desenvolvimento de ferramentas para aplicação em problemas industriais, sistemas de orientação de voo inclusos, é a escassez de modelos para testes. Este capítulo apresenta um conjunto de modelos para teste de ferramentas sobre projetos de sistemas autônomos, bem como a metodologia adotada para seu desenvolvimento. O conjunto apresentado possui papel nuclear neste trabalho uma vez que permite a execução com parâmetros controlados de toda a metodologia apresentada no Capítulo 4. A geração dos modelos consiste na primeira etapa do processo apresentado na Figura 3.1, que parte da geração dos modelos até a sua verificação.

Modelos de sistemas de orientação de voo (FGS, do inglês *Flight Guiding System*) devem comparar o estado sensoriado da aeronave ao estado desejado, gerando comandos que minimizem sua diferença. Modelos de FGS para teste de ferramentas sobre projetos de sistemas autônomos se dividem em duas categorias: modelos sintéticos e modelos reais. Modelos sintéticos possuem geralmente o propósito de, a partir de uma parametrização específica, modelar diferentes estratégias de projetos dos FGSs. Por não serem destinados a ferramentas de verificação de modelos, possuem para essa finalidade um nível de complexidade na parametrização que gera um número de estados a serem verificados muito maior do que o de sistemas reais, além de um código volumoso contendo diversas características que não precisariam ser representadas em estados específicos do



Figura 3.1: Diagrama das etapas de geração à verificação de um sistema de voo

sistema. Modelos reais, de aplicação industrial, possuem uma grande complexidade de implementação, sendo pouco adequados para o teste das ferramentas durante seu estágio de desenvolvimento. Além disso são protegidos por direitos de propriedade intelectual, não sendo facilmente acessíveis.

Este capítulo apresenta modelos que objetivam o teste de ferramentas baseadas em Matlab/Simulink, com codificação aberta e disponível a outros pesquisadores. Utilizamos um conjunto de modelos desenvolvidos dentro do próprio projeto PASARP¹ e em parceria com a Embraer que inclui um modelo asa-fixa [Konig, 2018b], um modelo híbrido VTOL/asa-fixa [Konig, 2018a], um modelo VANT com três graus de liberdade [Turetta, 2018] e dois modelos completos com abstrações distintas em relação aos parâmetros físicos da aeronave. Os modelos simplificados implementados possuem detalhes e explicações que facilitam a inserção de funcionalidades e detalhes de comportamento por parte de projetistas especialistas. Um modelo mais completo foi apresentado por Miller [Miller, 2003], Figura 3.2, entretanto para o teste das ferramentas desenvolvidas neste trabalho necessitamos de modelos com estratégias de implementação explícitas para a verificação de propriedades, não disponíveis nesse modelo. Outro modelo de interesse construído pela NASA [Riddick, 2018] também de mostrou inadequado por possuir um esquema de parametrização complexo que vincula as dinâmicas da aeronave a arquivos de inicialização MATLAB externos ao Simulink.

Nesse contexto desenvolvemos um conjunto de modelos genéricos de FGS com ênfase em seu bloco de Leis Operacionais que contempla características de modelos VTOL, asa fixa e híbridos. O conjunto desenvolvido possui versões dos modelos mencionados anteriormente adaptadas segundo a estrutura de abstração apresentada neste capítulo, estando seus elementos apresentados na Tabela 3.1. Cada modelo foi implementado no ambiente Simulink e simulado em um caso de voo com a missão de decolagem, deslocamento e pouso sem obstáculos. Este capítulo também apresenta um esquema geral para a construção de modelos sintéticos específicos com aplicabilidade em ferramentas de verificação de modelos de aeronaves autônomas. Os modelos desenvolvidos são uma contribuição para trabalhos que envolvam verificação de sistemas de controle de aeronaves, em especial sistemas autônomos.

3.1 Fluxo de informação do comando à atuação

Entender quais são as etapas do fluxo de informação do comando à atuação é essencial para se entender quais são os elementos essenciais de um modelo de sistema de controle

¹O projeto PASARP, hospedado no laboratório LUAR - DCC/UFMG, se dedica a desenvolver soluções para a verificação de sistemas de aviação.

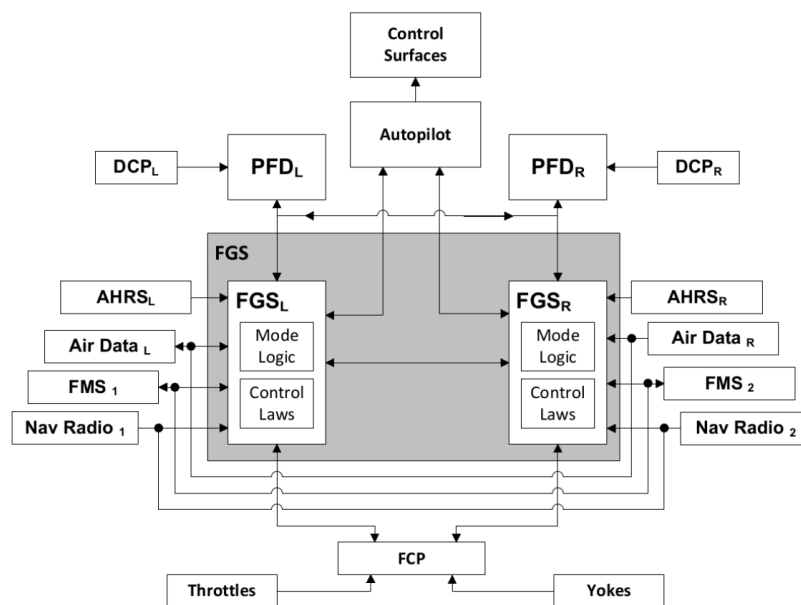


Figura 3.2: Visão geral do FGS [Miller, 2003]. Nele o sistema recebe entradas dos subsistemas *Attitude Heading Reference System* (AHRS), *Air Data System* (ADS), *Flight Management System* (FMS), e dos rádios de navegação. Utilizando esta informação calcula comandos que movimentam as superfícies de controle necessárias para diminuir a diferença entre as posições calculada e desejada.

de aeronave autônoma. O fluxo de informação do comando à atuação apresentado na Figura 3.3 de um VANT parte de um conjunto de comandos de alto nível dados por um piloto ou por um piloto autônomo de forma remota ou embarcada. Tais comandos definem as mudanças ou manutenção no deslocamento da aeronave. Os comandos de alto nível são então transformados em referências para os atuadores que podem atuar diretamente ou através de um sistema de controle. Consideraremos apenas os casos em que os comandos são controlados, abstraindo atuações diretas como controles que repetem o sinal de entrada na saída. As referências servem de entrada para os sistemas de controle que geram os sinais para os atuadores, que modificam ou mantém as dinâmicas da aeronave. O estado da aeronave é então medido por sensores que geram um retorno tanto para o sistema de controle quanto para o programa do painel de controle, que gera a visualização dos dados para os operadores. O fluxo de informação do comando à atuação se organiza como:

1. Comandos de mudança ou manutenção de estado são gerados através do painel de controle;
2. Um controlador em nível lógico operacional trata os comandos gerando referências e selecionando os blocos de controle dinâmico para os atuadores associados;

| Nome | Componente | Subsistemas | Código | Representação |
|------------------------------------|-------------------|--------------------|---------------|----------------------|
| Alvo estático | Alvo | 0 | A1 | Figura 3.15 |
| Alvo com decolagem e pouso | Alvo | 0 | A2 | Figura 3.17 |
| Alvo Turretta | Alvo | 0 | A3 | Figura 3.16 |
| Alvo em deslocamento | Alvo | 0 | A4 | Figura 3.18 |
| Rastreador em 4 direções | Rastreador | 2 | R1 | Figura 3.11 |
| Rastreador Turretta | Rastreador | 0 | R2 | Figura 3.21 |
| Rastreador angular | Rastreador | 3 | R3 | Figura 3.10 |
| Orientação em velocidade e atitude | Orientação | 3 | O1 | Figura 3.13 |
| Orientação Turretta | Orientação | 2 | O2 | Figuras 3.25 e 3.26 |
| Orientação angular | Orientação | 2 | O3 | Figuras 3.12 |
| Aeronave em velocidade e atitude | <i>Airframe</i> | 0 | F1 | Figura 3.9 |
| Aeronave Turretta | <i>Airframe</i> | 0 | F2 | Figura 3.27 |
| Míssil completo | <i>Airframe</i> | 5 | F3 | Figura 3.6 |
| Modelo VTOL controlado | Modelo completo | 4 | M1 | Figura 3.8 |
| Modelo Turretta | Modelo completo | 4 | M2 | Figura 3.14 |
| Modelo de um míssil controlado | Modelo completo | 4 | M3 | Figura 3.5 |

Tabela 3.1: Sistemas componentes dos modelos completos.

3. Blocos de controle dinâmico recebem e processam as referências gerando potencial para os atuadores;
4. Atuadores são alimentados de acordo com o sistema de controle e modificam ou mantém o estado da aeronave;
5. Sensores medem e encaminham as características físicas do estado da aeronave para os sistemas de leis operacionais, de controle e para o programa de painel de controle;
6. O estado da aeronave é apresentado no painel de controle.

Partindo da análise desse fluxo, geramos um esquema em que sistemas de controle de aeronaves possam ser divididos em três partes principais: Leis Operacionais, Leis de Controle e Piloto (relacionado ao Software que pode tanto ser um piloto autônomo, um piloto remoto ou uma apresentação de dados para um piloto a bordo). O sistema de controle então é aplicado sobre a modelagem da aeronave em si, que contempla o conjunto de sensoriamentos e atuações, calculados de acordo com as equações físicas que repre-

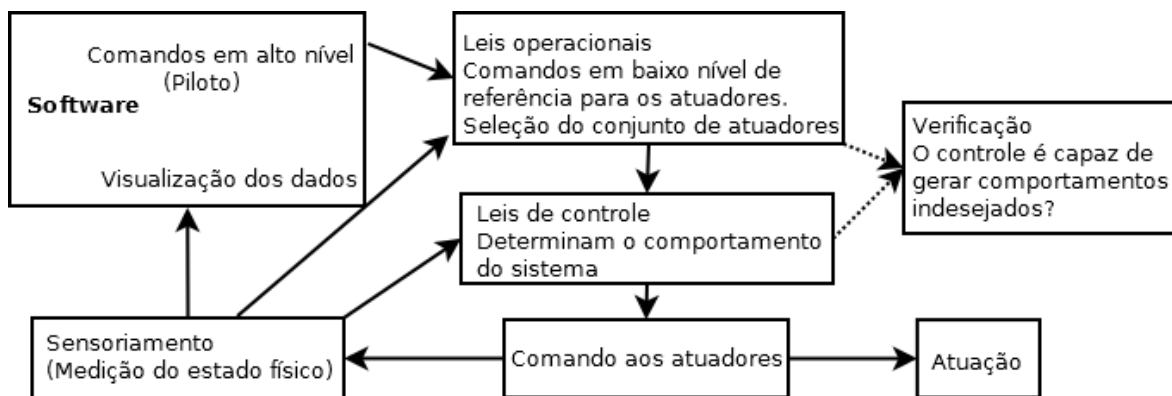


Figura 3.3: Diagrama geral de comando, controle e verificação de um sistema de voo

sentam sua dinâmica. Para cada bloco pertencente à modelagem realizada apresentamos no Capítulo 5 modelos em linguagens de verificação de modelos que os representam.

3.2 Análise da estrutura de projeto de um modelo de controle de projétil em Simulink

Esta seção consiste na análise de um modelo de controle automático de um projétil em perseguição a um alvo [The Mathworks, 2014]. Esse modelo foi escolhido por pertencer ao pacote de exemplos do Simulink para sistemas de controle aéreo, possuindo um artigo correspondente notável e de fácil acesso. Aqui abstraímos as implementações de controle analógico como caixas pretas e destacamos os parâmetros importantes para a sua adaptação para o problema de controle automático de uma aeronave para transporte de passageiros. Utilizar um modelo já verificado como referência ouro para a construção de novos modelos permite a geração de novos modelos para a realização de testes de ferramentas de verificação de sistemas de aviação. Ainda, diferentes modelos já abordados pelo projeto PASARP [Turetta, 2018, König, 2018a] podem ter seus sistemas de leis operacionais e de controle acoplados a este, de modo a aumentar o conjunto de modelos.

O modelo Simulink utilizado foi gerado pela The Mathworks Inc. [The Mathworks, 2014] e foi construído a partir do modelo físico apresentado na Figura 3.4. O modelo originalmente representa um projétil controlado pela cauda viajando entre Mach 2 e Mach 4 com altitudes entre 10.000 pés e 60.000 pés e com ângulos de ataque variando em 20 graus. Nela η representa a deflexão do leme da cauda, G representa o centro de gravidade, a_z representa a aceleração normal, α representa o ângulo de incidência e q representa a taxa de variação do ângulo de curvatura (*body rate*). A mesma abordagem utilizada para implementar um controle de míssil pode ser

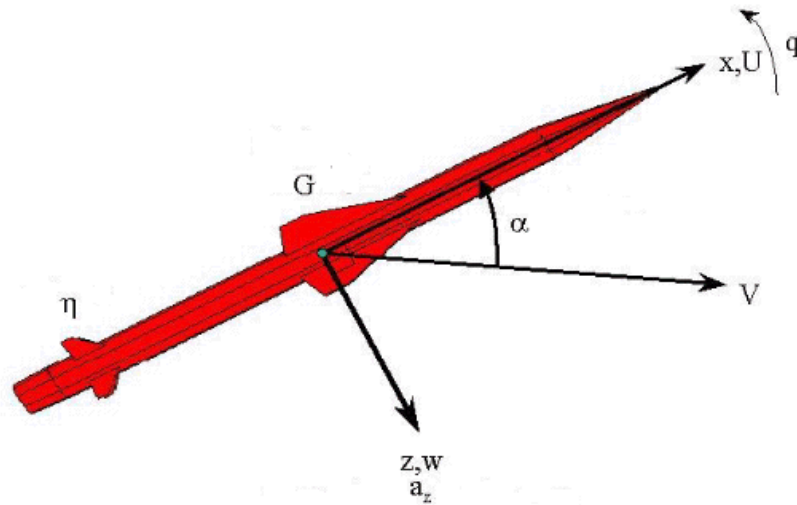


Figura 3.4: Modelo das dinâmicas da fuselagem de um projétil [The Mathworks, 2014] (adaptada)

utilizada para controlar descida de trem de pouso e outros modos de operação de um avião de transporte de passageiros.

O elemento principal do modelo é a representação não linear das dinâmicas de corpo rígido da fuselagem. Mesmo não sendo esse o ponto de foco do tradutor implementado, é importante notar que a abstração das dinâmicas da fuselagem pode ser feita enquanto o seu controle é verificado. As forças e momentos aerodinâmicos atuando sobre o projétil são gerados a partir de coeficientes que são funções não lineares do ângulo de incidência e do *Mach number*. A atmosfera é modelada independentemente à configuração da fuselagem, podendo ser utilizada em diversas configurações de modelos. A representação do sistema completo em blocos Simulink (Figura 3.5) é dividida em três subsistemas para rastreamento, orientação e comportamento da fuselagem considerando o piloto automático.

O modelo do comportamento da fuselagem, Figura 3.6 possui quatro subsistemas principais controlados através de um piloto automático que objetiva controlar a aceleração normal ao corpo do projétil. O modelo da atmosfera calcula a mudança nas condições atmosféricas a partir da mudança de altitude nas regiões da troposfera (até 11Km) e baixa estratosfera (11 a 20 Km). O modelo do atuador do leme e dos sensores acoplam o piloto automático à fuselagem. O modelo da aerodinâmica e das equações de movimento calcula a magnitude das forças e momentos atuando no corpo do projétil e integra as equações de movimento gerando as forças e momentos aplicados ao projétil nos eixos de seu corpo definindo as equações lineares e angulares. Ainda neste bloco, os coeficientes aerodinâmicos são armazenados em conjuntos de dados e seus valores são interpolados

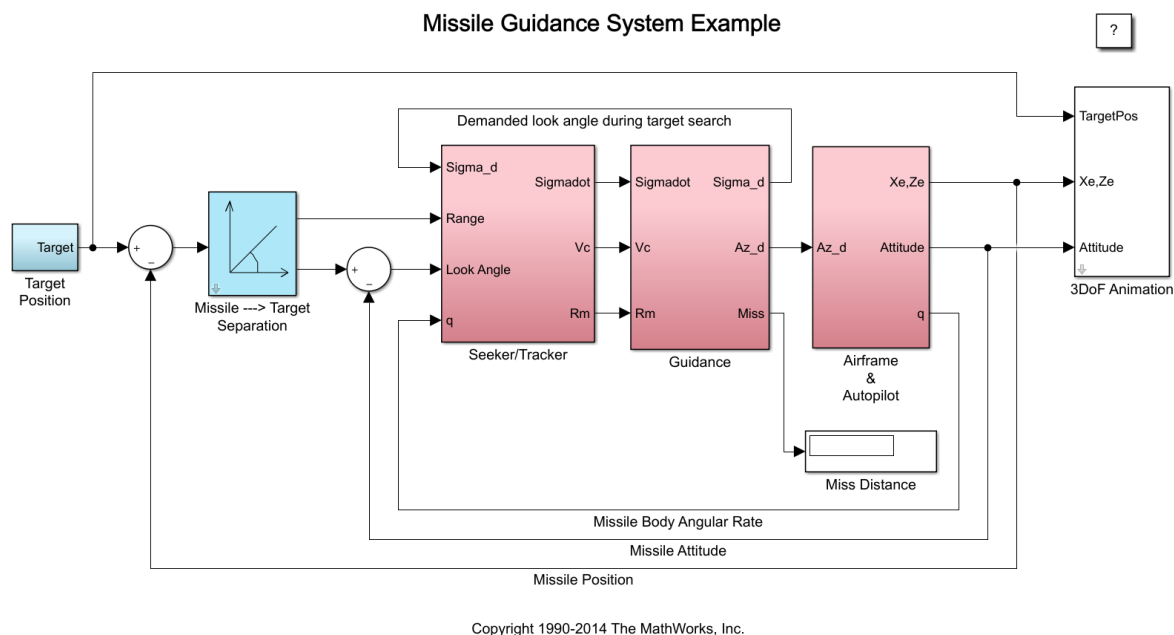


Figura 3.5: Modelo Simulink contendo a referência (posição do alvo), o rastreador, o sistema de controle e a fuselagem autopilotada de um projétil (M3). [The Mathworks, 2014]. A implementação de seus sistemas se apresenta das Figuras 3.18 (Alvo), 3.10 (Rastreador), 3.12 (Direcionamento), 3.6 (Airframe).

de modo a determinar as condições de operação.

O sistema de piloto automático controla a aceleração normal ao corpo do projétil. A implementação analisada utiliza uma estrutura de três laços em malha fechada utilizando medições de um acelerômetro situado à frente do centro de gravidade e uma taxa de rotação para prover um amortecimento adicional. Os ganhos do controlador são calculados a partir do ângulo de incidência e do *Mach number* estando ajustados para um desempenho robusto a uma altitude de 10.000 pés. Importante salientar que a malha de controle utilizada não é a PID típica.

O sistema de orientação se baseia em um laço em malha fechada que consiste em um subsistema Localizador/Rastreador (*Seeker/Tracker*) que mede a movimentação relativa entre o projétil (*airframe*) e o alvo e o subsistema de orientação que gera os comandos de aceleração normal a serem transmitidos ao piloto automático, cujo sensoriamento retorna ao Localizador/Rastreador fechando um laço conforme apresentado na Figura 3.5. O diagrama de transição entre os estados do sistema de orientação é representado pelo bloco *Guidance* apresentado na Figura 3.7 e apresenta um bloco de leis operacionais relativas ao estado de perseguição ao alvo e detonação do projétil (note, a mesma abordagem poderia ser utilizada para controlar descida de trem de pouso e outros modos de operação de um avião de transporte de passageiros). A saída do subsistema Localizador/Rastreador gera referências para os atuadores das alavancas dos lemes de modo a alinhar o projétil

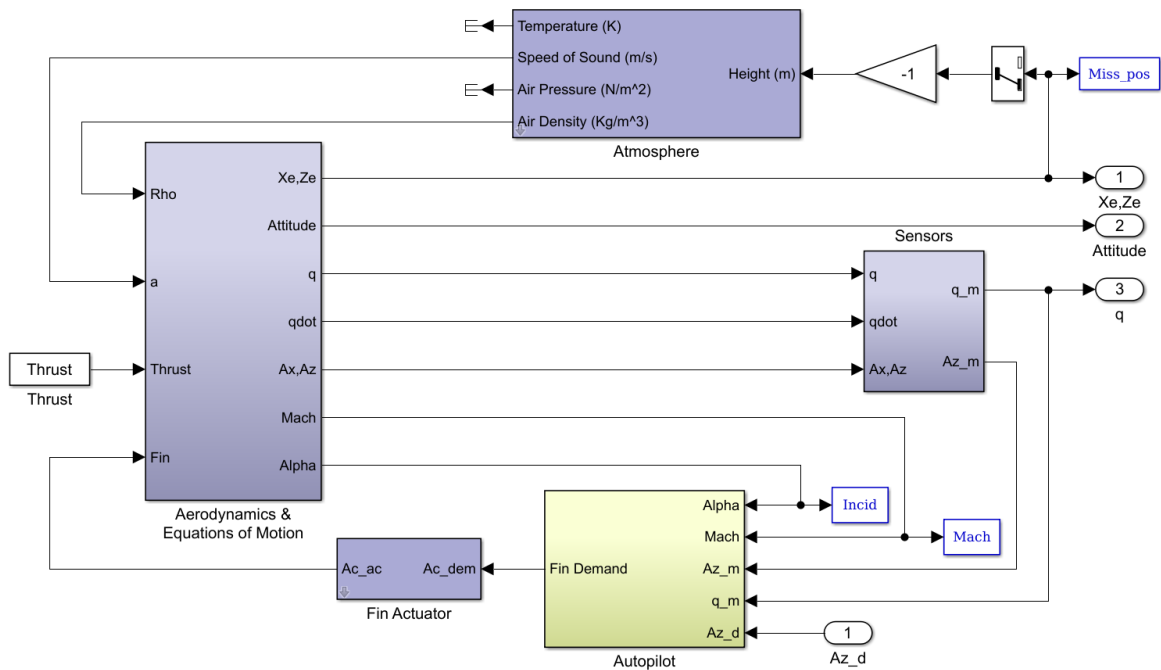


Figura 3.6: Modelo do comportamento da *airframe* de um projétil (F3) [The Mathworks, 2014].

ao alvo.

3.3 Estrutura de abstração para modelos de aeronaves controladas em Simulink

Para gerar um esquema geral de tradução de modelos de aeronaves descritos em códigos Simulink para linguagens de verificação formal é importante a definição de uma estrutura de abstração genérica dos modelos de aeronave. Desse modo, simplifica-se o entendimento de quais partes do modelo podem ser traduzidas utilizando diferentes formalismos e motores de verificação. A estrutura de abstração utilizada pela The Mathworks Inc. [The Mathworks, 2014] foi escolhida por sua evidência na literatura, por pertencer ao pacote padrão de aeromodelos do Simulink e por sua simplicidade, dividindo todo um sistema complexo em apenas quatro subsistemas maiores. A representação do sistema completo em blocos Simulink (Figura 3.5) é dividida em quatro subsistemas para determinação do alvo, rastreamento, orientação e comportamento da *airframe* considerando o piloto automático. O sistema de alvo determina quais as referências GPS a serem atingidas pela *airframe*, sendo comandadas pelo piloto. O sistema de orientação se baseia em dois sistemas menores onde o primeiro, Rastreamento, é responsável pelo posicionamento do alvo em relação à *airframe* e o segundo, Direcionamento, determina a orien-

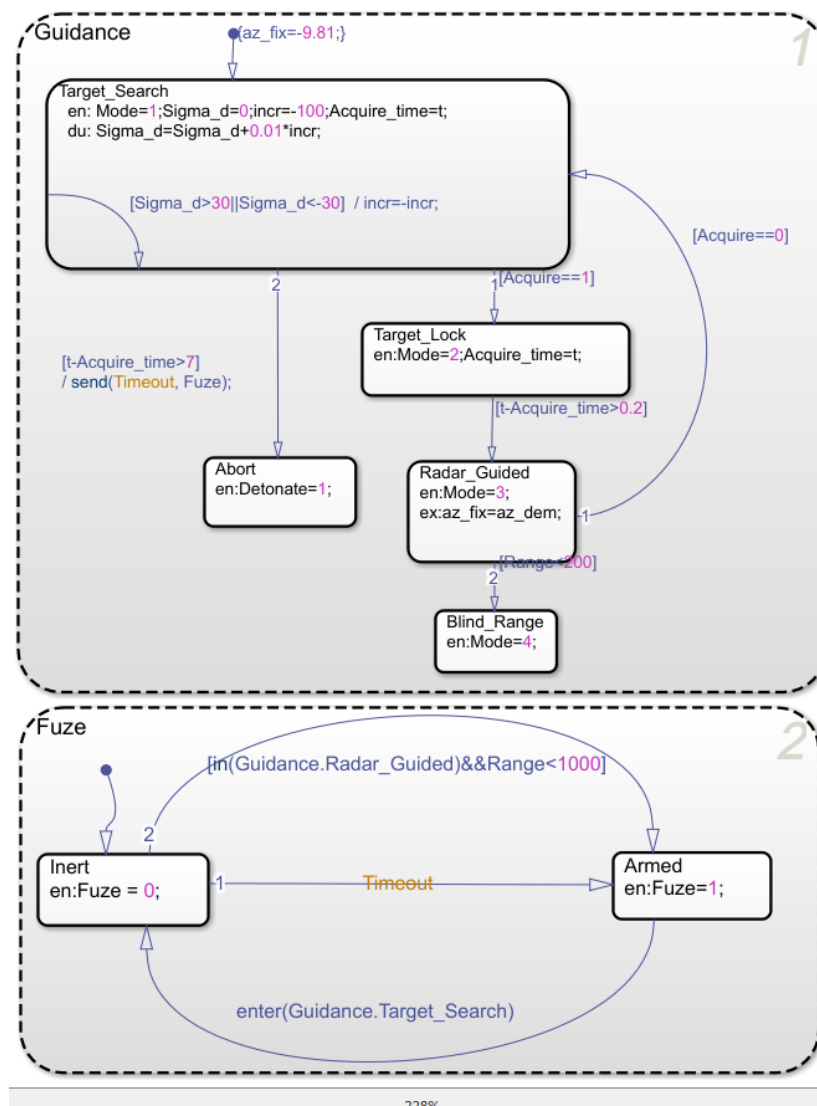


Figura 3.7: Diagrama de transição de estados de um sistema de orientação modelado através do Matlab Simulink [The Mathworks, 2014]

tação a ser seguida pela mesma. O sistema da *airframe* determina todo o seu comportamento físico, considerando tantas variáveis físicas quantas se modele. Uma implementação utilizando os quatro subsistemas baseada nessa estrutura é apresentada na Figura 3.8 e é detalhada ao longo da discussão desta seção.

O modelo do **comportamento da *airframe***, Figura 3.6, possui quatro subsistemas principais controlados através de um piloto automático que objetiva controlar a aceleração normal ao corpo do projétil. A partir de sua análise, percebemos que o bloco responsável pela *airframe* modela todo o comportamento físico da aeronave, dadas as referências dos atuadores geradas pelo sistema de orientação através do bloco de direcionamento. Ela possui como saída o conjunto de sensoriamentos do estado físico da

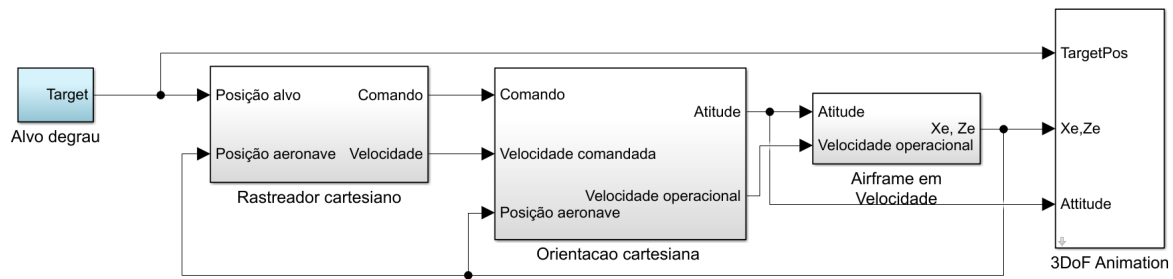


Figura 3.8: Uma implementação (M1) sobre uma estrutura de abstração dividida em Alvo, Rastreador, Orientação e Airframe. A implementação de seus sistemas se apresenta das Figuras 3.17 ou 3.15 (Alvos A1 e A2), 3.11 (Rastreador R1), 3.13 (Orientação O1), 3.9 (Airframe F1).

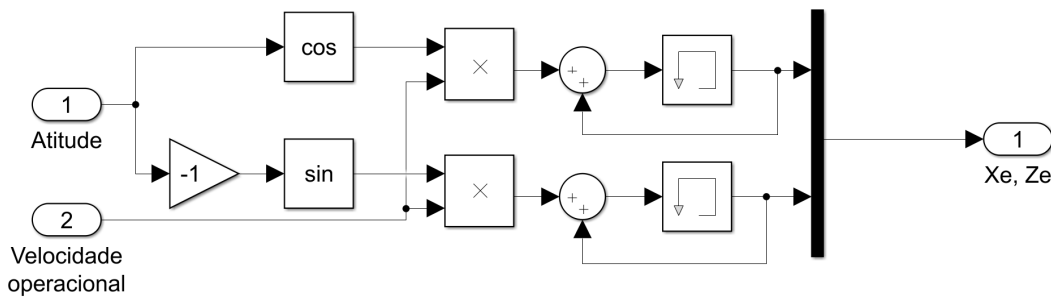


Figura 3.9: Modelo da *airframe* nos eixos cartesianos XZ considerando velocidade e atitude, com coordenadas GPS como saída (F1).

aeronave. Uma abordagem simples para o comportamento da aeronave é aquela que determina, dada uma velocidade (medida em unidades por ciclo) de entrada e uma atitude, as coordenadas GPS da aeronave ao longo do tempo e se mostra implementada na Figura 3.9.

O **sistema rastreador**, (Localizador/Rastreador ou *Seeker/Tracker*), mede a movimentação relativa entre o projétil e o alvo dadas as suas referências de posição. O modelo apresentado na Figura 3.10 considera distâncias angulares entre a *airframe* e o alvo, possuindo um sinalizador de sua aquisição. A partir de sua análise, percebemos que ele fornece as referências físicas de comportamento desejadas da *airframe* como saída. A parcela do controle operacional correspondente às referências de comportamento da *airframe* cabem também a este sistema. Um sistema rastreador em XZ, modelando apenas o deslocamento em uma dimensão com variação de altura, com cálculo de distâncias cartesianas e com geração de comandos em seus quatro eixos é apresentado na Figura 3.11.

O **sistema de orientação** de uma *airframe* deve, a partir de um conjunto de entradas que representem referências à sua movimentação, derivar a partir de leis operacionais de mais alto nível quais as referências para os seus atuadores. A Figura 3.12 apresenta um conjunto para processamento dos parâmetros passados a um sistema de orientação de

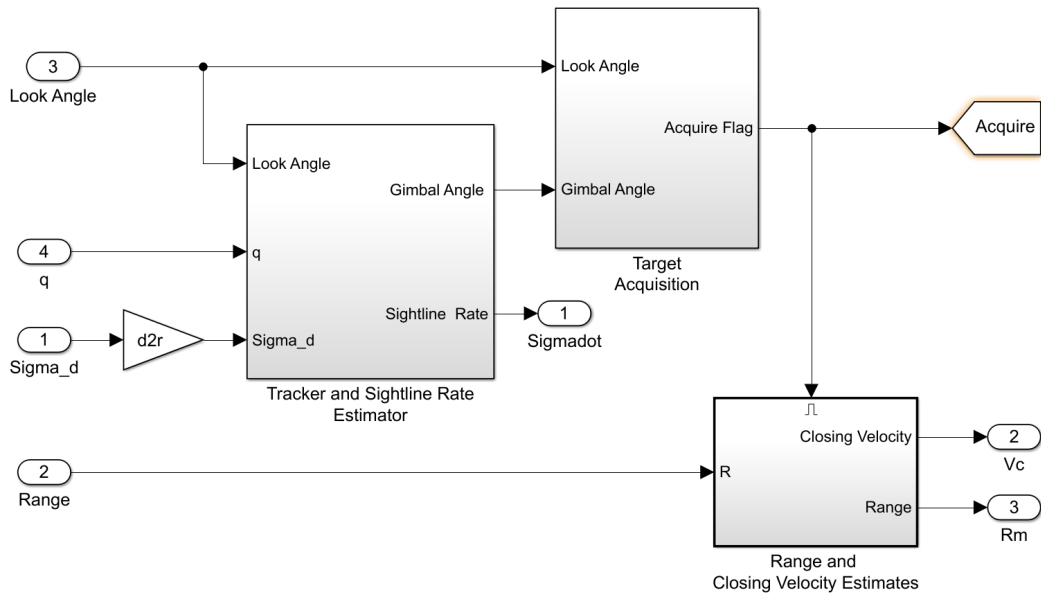


Figura 3.10: Modelo de um sistema rastreador de um míssil (R3) [The Mathworks, 2014].

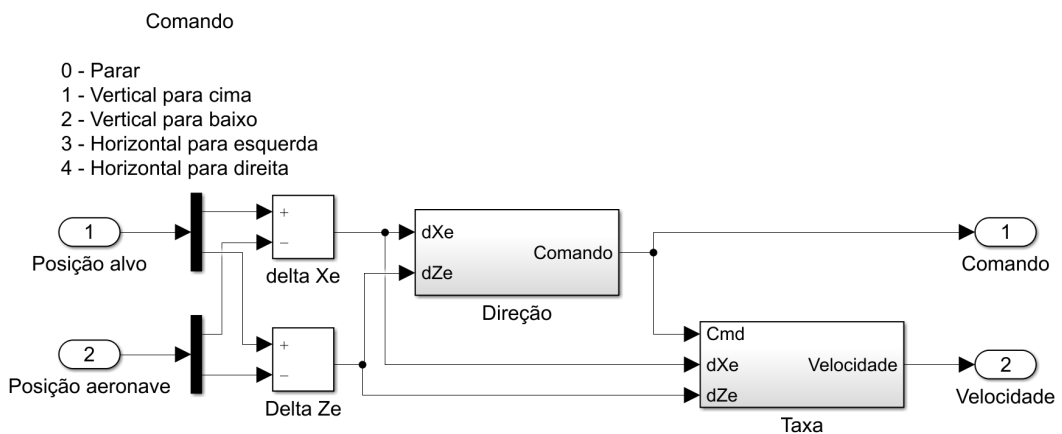


Figura 3.11: Modelo de um sistema rastreador em XZ sobre os eixos cartesianos (R1). A implementação de seus sistemas se apresenta das Figuras 3.19 (Direção), 3.20 (Taxa).

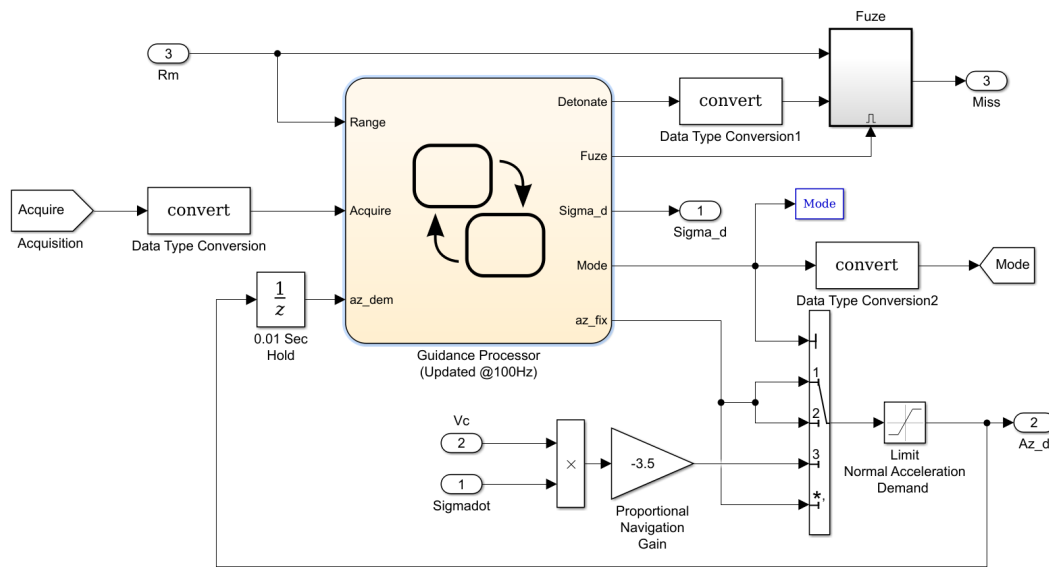


Figura 3.12: Modelo de um sistema de orientação de um míssil (O3) [The Mathworks, 2014]. A máquina de estados referente ao *Guidance Processor* apresenta-se na Figura 3.7.

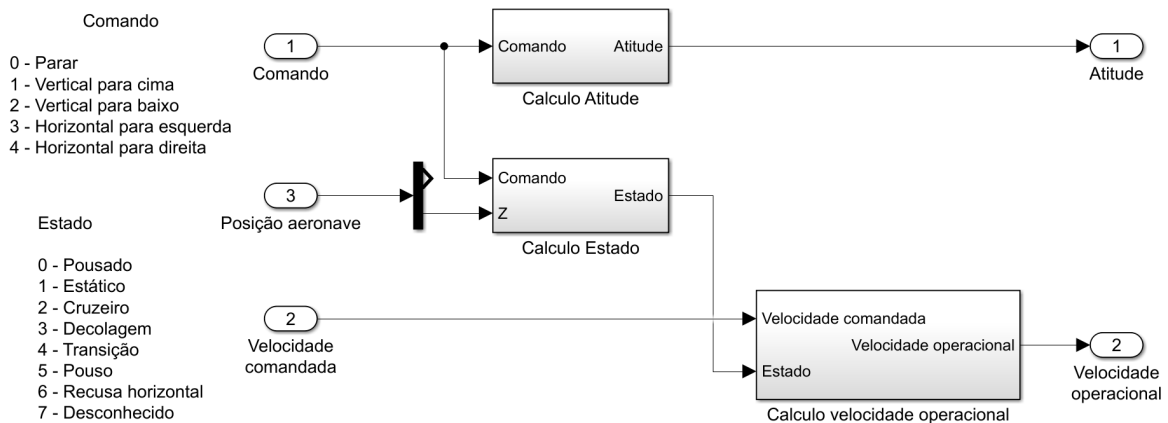


Figura 3.13: Modelo de um sistema de orientação baseado em atitude e velocidade O1. A implementação de seus sistemas se apresenta nas Figuras 3.22 (Calculo Atitude), 3.23 (Calculo Estado), 3.24 (Calculo velocidade operacional).

modo a determinar os ângulos de atuação de um míssil. Note a existência de uma máquina de estados para o cálculo, a partir de uma lógica de mais alto nível, do comportamento da *airframe*. A partir de sua análise geramos um sistema análogo, Figura 3.13, que recebe o comando gerado pelo rastreador para cálculo da atitude da *airframe* bem como o processamento de seu estado para a determinação das velocidades operacionais da aeronave dadas restrições operacionais de voo.

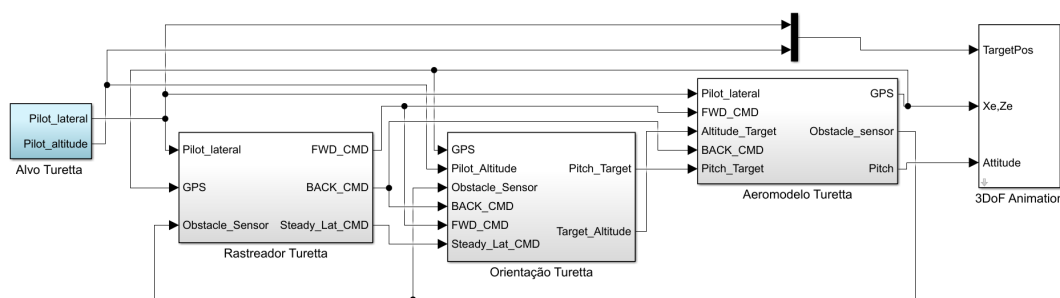


Figura 3.14: Um modelo VANT com três graus de liberdade (M2) [Turretta, 2018] (Adaptado). A implementação de seus sistemas se apresenta das Figuras 3.16 (Alvo), 3.21 (Rastreador), 3.25 e 3.26 (Direcionamento), 3.27 (Airframe).

3.4 Decisões de implementação de modelos de aeronaves controladas em Simulink

Esta seção avalia decisões de implementação dos quatro blocos principais da estrutura de abstração apresentada na Seção 3.3 (alvo, rastreador, orientação e *airframe*) a partir de dois modelos [The Mathworks, 2014, Turretta, 2018], sendo o segundo adaptado à estrutura sem modificações funcionais apresentado na Figura 3.14. A partir dessa análise foram implementados modelos como o apresentado na Figura 3.8, cuja implementação também é avaliada nesta seção.

3.4.1 Implementação do bloco alvo

A implementação de um alvo dentro da estrutura de abstração escolhida possui a pré-definição das posições das etapas de voo e como saída um vetor de posições, que determinam o alvo da aeronave em questão. A implementação mais simples de um alvo é apresentada na Figura 3.15, que determina apenas uma posição estática para o alvo. Ela representa um destino sem determinar seu caminho, o que pode ser utilizado para avaliar cumprimentos de leis operacionais como por exemplo a recusa de uma ordem de deslocamento horizontal de um VTOL sem haver uma decolagem prévia, onde espera-se que a aeronave decole à altura do manual de voo, desloque-se horizontalmente e pouse. Implementações que consideram etapas distintas de voo são apresentadas nas Figuras 3.16 e 3.17, que utilizam funções degrau com tempo controlado para determinar os momentos de ordem de decolagem, pouso e cruzeiro. Uma implementação de um alvo para um míssil é apresentada na Figura 3.18, onde os parâmetros do alvo são dados em coordenadas polares e onde o alvo também possui variações de posição contínua ao longo do tempo.

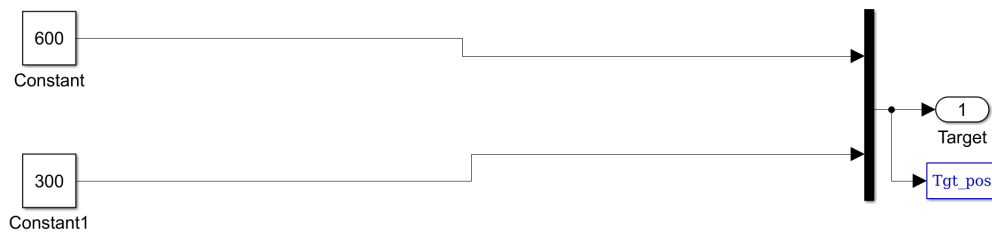


Figura 3.15: Modelo de um alvo estático com posição de destino (600, 300) nos eixos cartesianos XZ (A1).

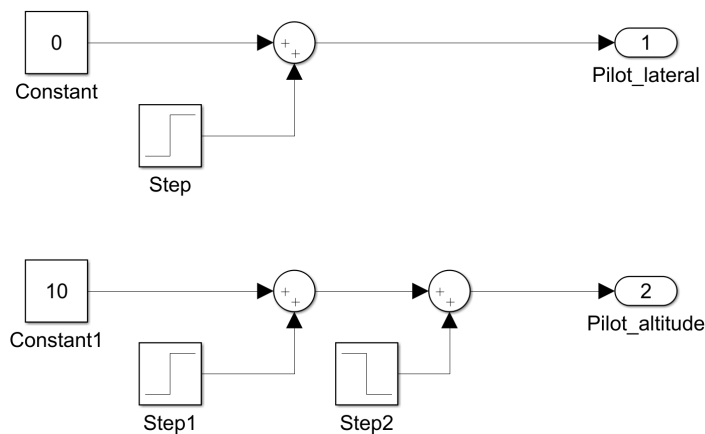


Figura 3.16: Modelo de um alvo nos eixos cartesianos XZ iniciando em decolagem (10, 0) e passando por um deslocamento até o pouso (A3) [Turetta, 2018] (Adaptado).

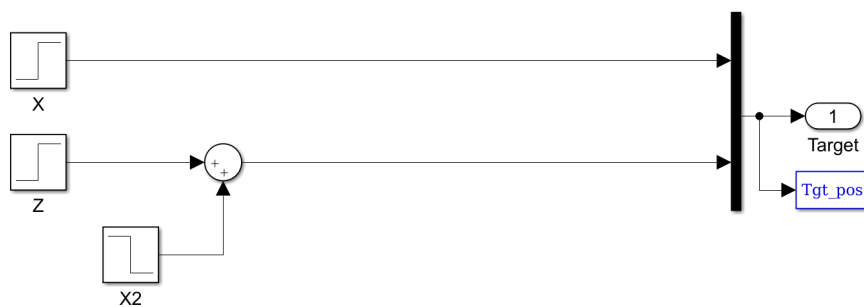


Figura 3.17: Modelo de um alvo nos eixos cartesianos XZ com decolagem, deslocamento e pouso (A2).

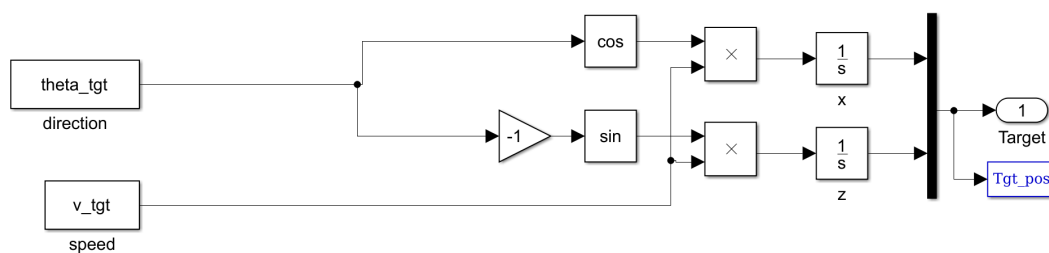


Figura 3.18: Modelo de um alvo nos eixos cartesianos XZ controlado por coordenada polar e velocidade (A4) [The Mathworks, 2014]

3.4.2 Implementação do bloco rastreador

Uma implementação de rastreador dentro da estrutura de abstração escolhida representa a avaliação da posição relativa entre a aeronave e o alvo, possuindo como entradas suas posições e como saídas parâmetros para a movimentação da aeronave.

A Figura 3.10 apresenta um modelo de um sistema rastreador de um míssil com referências polares que possui como entradas o ângulo de busca, dado como os ângulos relativos entre a *airframe* e o alvo, a distância entre eles e a taxa de rotação da *airframe*. Através de três blocos ele verifica se o alvo foi adquirido pelo míssil, a velocidade de aproximação e a distância a ainda ser percorrida. Esses blocos são um rastreador propriamente dito que calcula referências para a atuação dos elementos angulares da *airframe*, um verificador para a aquisição do alvo e um estimador de distância e velocidade de aproximação.

A Figura 3.11 apresenta um modelo de um sistema rastreador em XZ sobre os eixos cartesianos que possui como entrada as posições do alvo e da aeronave. Diferentemente do modelo apresentado na Figura 3.10, ele calcula as distâncias dentro do próprio bloco. A partir desses parâmetros ele calcula o comando a ser designado à orientação (Figura 3.19) bem como sua velocidade (Figura 3.20). A avaliação do comando a ser enviado consiste apenas de estruturas lógicas e análises de limiar. Uma margem de erro de 1 é assumida para considerar que a *airframe* atingiu a posição-alvo designada. Um bloco de memória é utilizado para garantir a sincronização do modelo prevenindo erros como os mencionados na Seção 3.1. O cálculo da velocidade apresentado pela Figura 3.20 ordena que a *airframe* atinja a posição-alvo no próximo ciclo, tipicamente incoerente ao manual de voo. Essas leis operacionais são asseguradas pelo bloco de Orientação.

A Figura 3.21 apresenta o modelo de um rastreador em X que considera o sensoramento de obstáculos. Ele possui como entrada um vetor de sinalizadores de obstáculos, a posição GPS da *airframe* e a referência do alvo em *Pilot_lateral*. Ele recusa os comandos caso eles induzam a *airframe* a uma colisão. Como saída ele fornece os comandos de deslocamento em X ou de estacionar até que o obstáculo seja removido.

3.4.3 Implementação do bloco orientação

Uma implementação de orientação dentro da estrutura de abstração escolhida representa a recepção do conjunto de comandos pelo rastreador e da análise de estados do sistema para a geração de referências para os atuadores da *airframe* (comumente por meio de leis de controle nela implementadas).

A Figura 3.12 apresenta o modelo de um sistema de orientação de um míssil. Ele considera um conjunto de estados, distâncias, velocidade e ângulos para alimentar uma

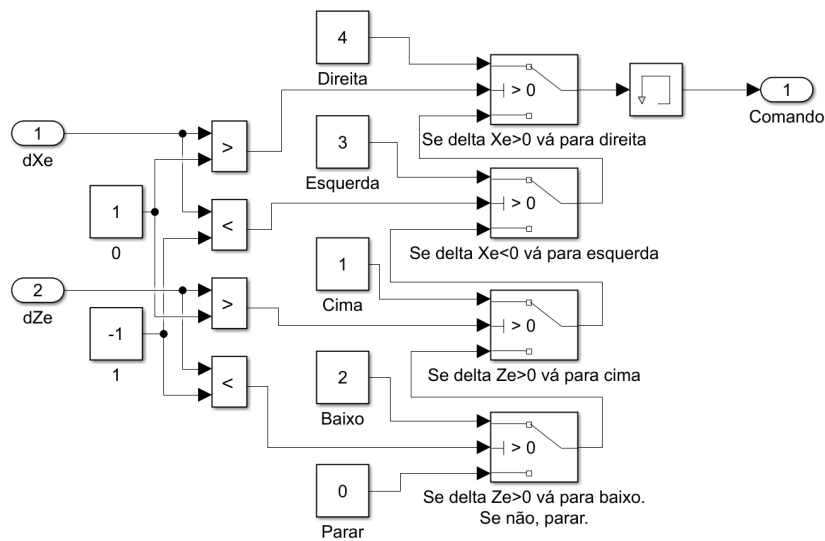


Figura 3.19: Modelo para definição do comando direcional de um rastreador em XZ em quatro direções cartesianas (avanço, recuo, subida e descida).

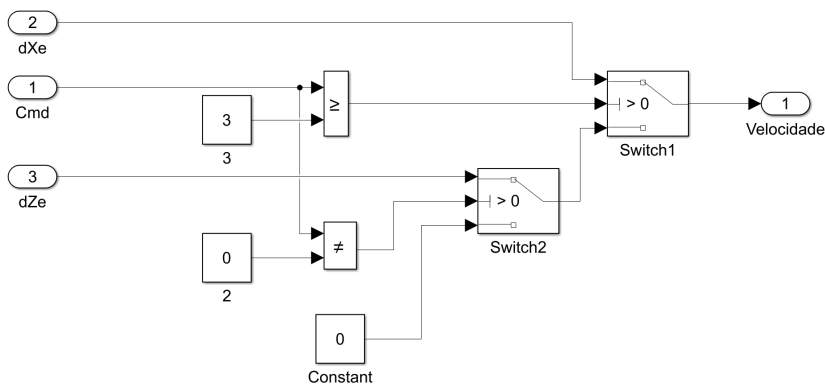


Figura 3.20: Modelo para definição da velocidade em um rastreador em XZ em quatro direções cartesianas.

máquina de estados e determinar os ângulos de referência para a *airframe*, além do modo de operação e um sinalizador sobre o erro do míssil sobre o alvo. O diagrama de transição entre os estados do sistema de orientação é representado pelo bloco *Guidance* apresentado na Figura 3.7 e apresenta um bloco de leis operacionais relativas ao estado de perseguição ao alvo e detonação do projétil.

A Figura 3.13 apresenta um modelo de um sistema de orientação baseado em atitude e velocidade. Ele é implementado através de três blocos. O primeiro (Figura 3.22) implementa o cálculo da atitude da *airframe* sobre uma das quatro direções cartesianas a partir de um comando sobre uma delas. O segundo (Figura 3.23) implementa a determinação do estado da aeronave a partir de sua altitude e do comando passado. O terceiro (Figura 3.24) implementa a determinação da velocidade operacional a partir do estado da

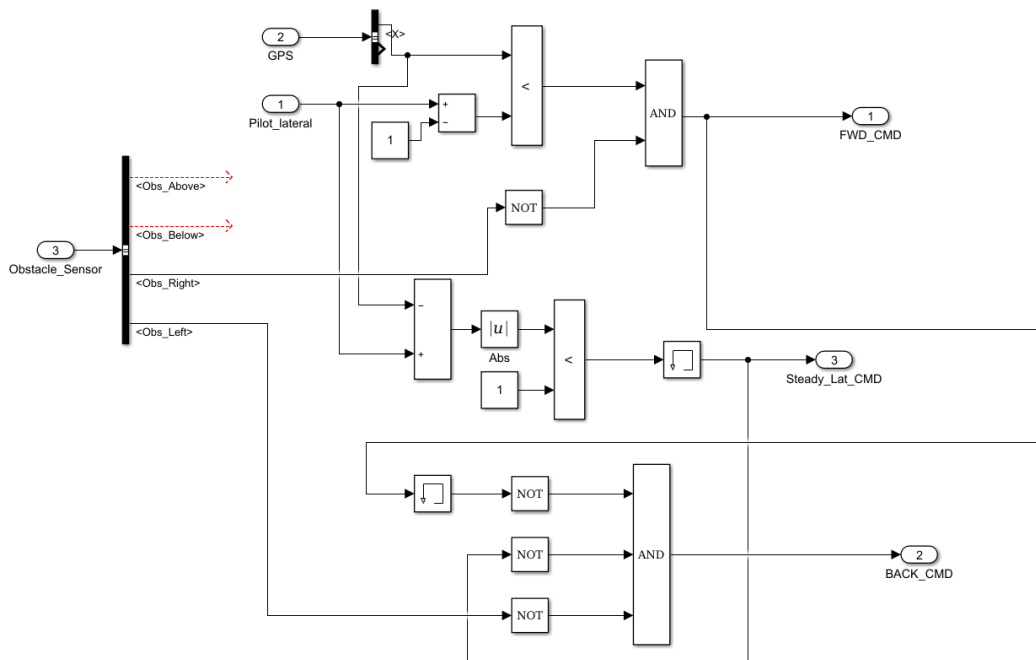


Figura 3.21: Modelo de um rastreador em X que considera o sensoriamento de obstáculos (R2) [Turetta, 2018] (Adaptado).

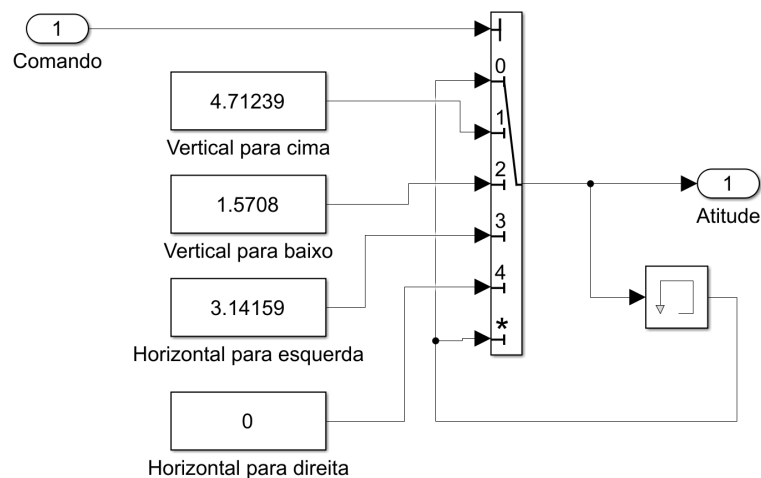


Figura 3.22: Um bloco para a orientação que, a partir de um comando que indique uma de quatro direções cartesianas, determina a atitude da aeronave.

aeronave e da velocidade comandada a partir de leis operacionais.

As Figuras 3.25 e 3.26 segmentam a implementação de uma orientação que determina as referências de ângulo de ataque a partir do vetor de comandos fornecidos pelo rastreamento e de altitude a partir de um estado implícito obtido através da altitude do alvo, posição da *airframe* e vetor de obstáculos.

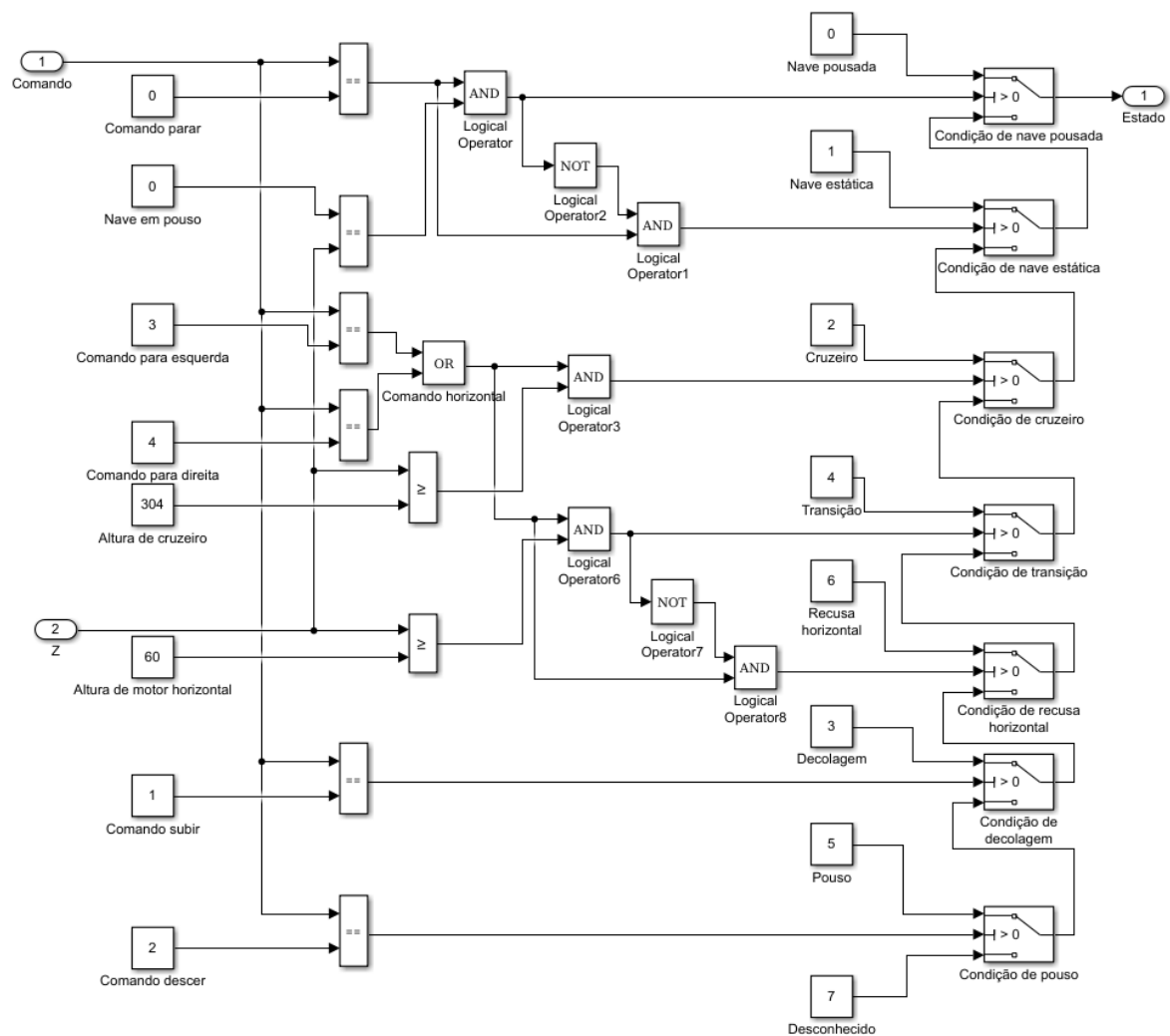


Figura 3.23: Um bloco para orientação que, a partir de um comando que indique uma de quatro direções cartesianas e da altitude atual da aeronave, determina o estado da aeronave segundo leis operacionais que definem alturas de cruzeiro e de ligação dos motores horizontais.

3.4.4 Implementação do bloco *airframe*

Uma implementação de *airframe* dentro da estrutura de abstração escolhida representa o conjunto de leis dinâmicas que determinam o comportamento da *airframe* dadas as referências pela Orientação. Ela pode modelar o corpo físico da *airframe*, condições de sensoriamento externas (como obstáculos, temperatura, velocidade do vento) e internas (como posição em GPS, velocidade em relação ao solo, velocidade em relação ao ambiente), outros aspectos físicos como resistência do ar e gravidade dentre outros de interesse.

Uma implementação da *airframe* de um míssil é apresentada na Figura 3.6. Ela possui como parâmetro apenas a aceleração normal (a_z) demandada, a partir da qual

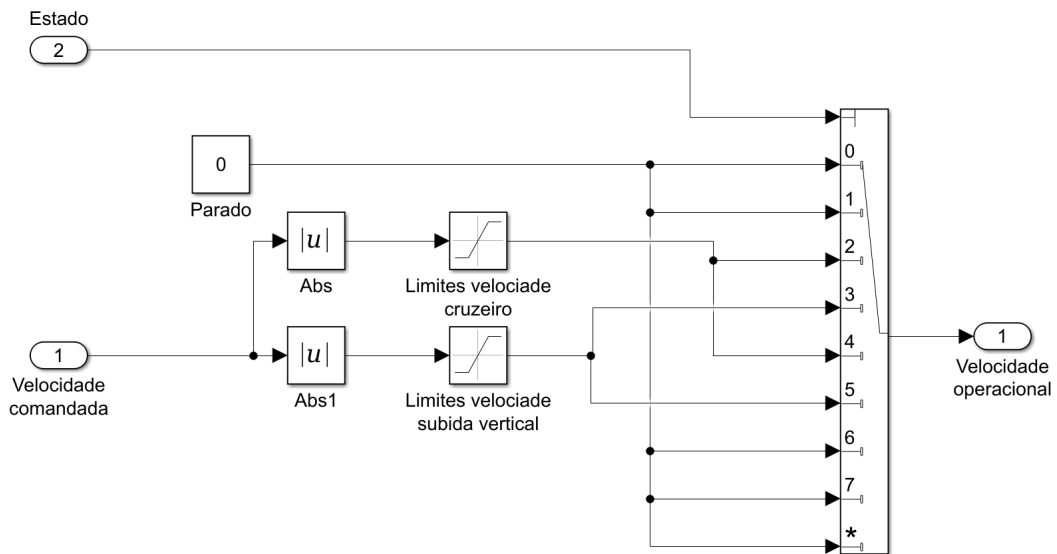


Figura 3.24: Um bloco para a orientação que, a partir de uma velocidade comandada e do estado atual da aeronave, determina a sua velocidade operacional segundo leis operacionais que restringem a velocidade de cruzeiro, velocidade de subida vertical e altitude de cruzeiro.

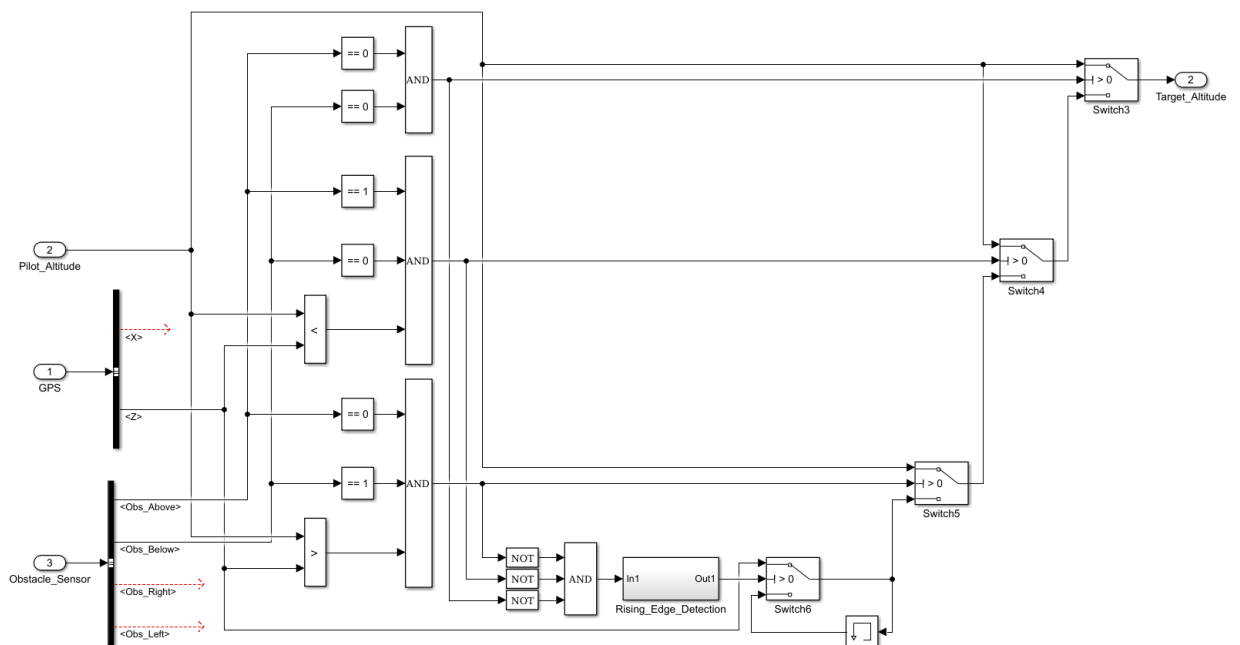


Figura 3.25: Fragmento de uma orientação (O2) que determina a altitude referência para a *airframe* a partir da altitude relativa da aeronave em relação ao alvo e vetor de sensores para obstáculos [Turetta, 2018] (Adaptado).

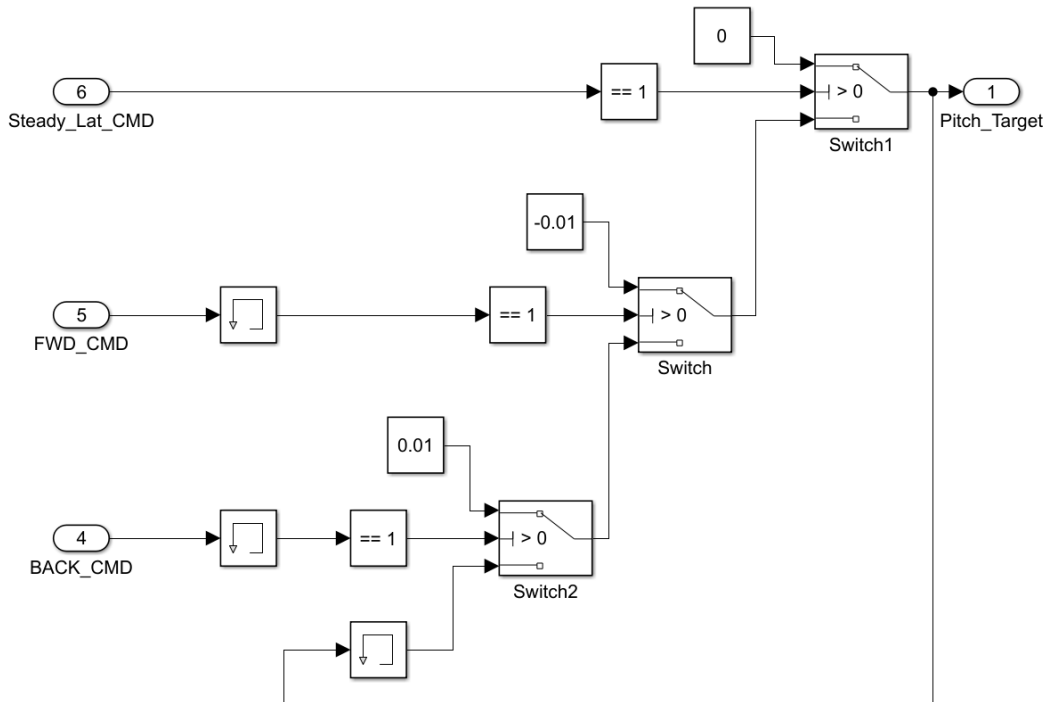


Figura 3.26: Fragmento de uma orientação que determina o ângulo de inclinação referencial para a *airframe* a partir do conjunto de comandos passados pelo sistema de rastreamento [Turetta, 2018] (Adaptado).

um sistema de controle (*autopilot*) calcula a atuação do leme. Com a atuação do leme e o conjunto de sensores, atmosfera, aerodinâmica e equações de movimento, o modelo calcula as posições em X e Z, a atitude da *airframe* e a sua taxa de rotação.

Uma implementação simples da *airframe* em 3 graus de liberdade, apresentada na Figura 3.9, possui como saída apenas os valores de sua posição, calculada como o somatório em cada componente das velocidades instantâneas a cada ciclo de simulação. Para isso foi realizada a decomposição vetorial nos componentes X e Z da atitude da *airframe*. O bloco de memória representa um atraso para que se realize a acumulação correta da posição, evitando problemas de concorrência como os apresentados na Seção 2.1, conforme a Equação 3.1.

$$p_i = p_{i-1} + v_i \quad (3.1)$$

Uma implementação mais detalhada de uma *airframe* em 3 graus de liberdade que considera leis de controle, o estado do ambiente e uma abstração do veículo está apresentada na Figura 3.27.

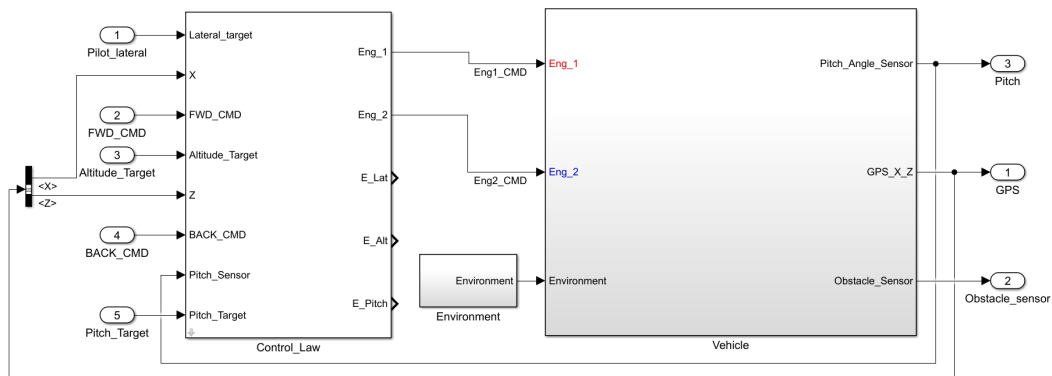
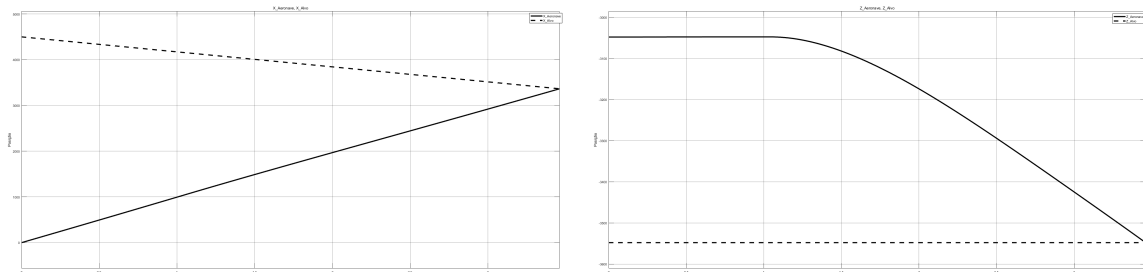


Figura 3.27: Modelo da *airframe* nos eixos cartesianos XZ considerando parâmetros físicos e ambiente (F2) [Turetta, 2018].



(a) Aproximação da *airframe* ao alvo em relação ao eixo X

(b) Aproximação da *airframe* ao alvo em relação ao eixo Z

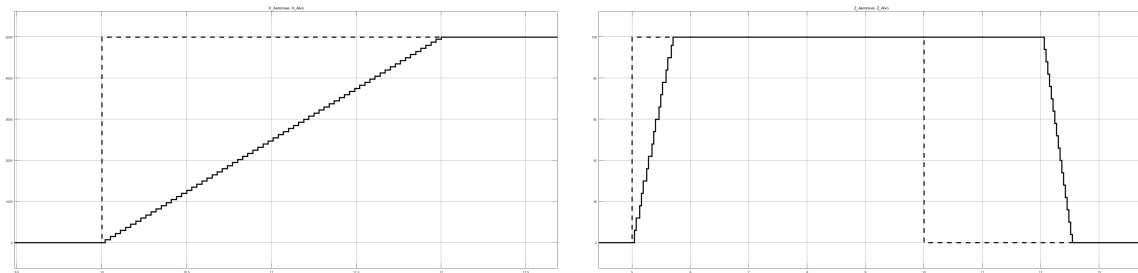
Figura 3.28: Simulações da aproximação da *airframe* em relação ao alvo utilizando o modelo apresentado na Figura 3.5 com ângulo de deslocamento do alvo em 3.1416 radianos e velocidade do alvo em 328 unidades por ciclo.

3.5 Simulação de modelos implementados

Para demonstrar as influências das decisões de implementação adotadas no comportamento dos modelos realizamos um conjunto de simulações, cujos resultados são apresentados nesta seção.

A implementação apresentada na Figura 3.5 utilizando os componentes A4, R3, O3, F3 da Tabela 3.1 gerou os resultados de simulação apresentados na Figura 3.28 utilizando como parâmetros do alvo um ângulo de deslocamento de 3.1416 radianos e velocidade de 328 unidades por ciclo. Nessa simulação o míssil se aproxima diretamente ao alvo no eixo X com uma característica linear gerada pela velocidade constante de ambos os projéteis. Em relação ao eixo Z, percebemos que o alvo se mantém em altitude constante (devido ao ângulo de deslocamento de 180 graus) e que o míssil se aproxima com um comportamento quase linear após a atuação de seu controlador com um atraso.

A implementação apresentada na Figura 3.8 utilizando os componentes A2, R1, O1,

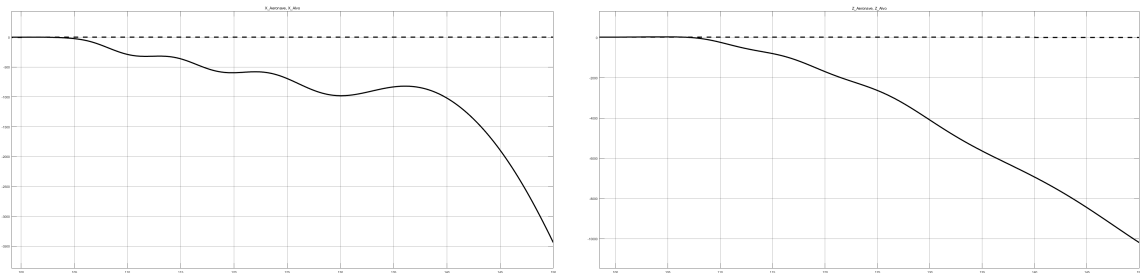


(a) Aproximação da *airframe* ao alvo em relação ao eixo X (b) Aproximação da *airframe* ao alvo em relação ao eixo Z

Figura 3.29: Simulações da aproximação da *airframe* em relação ao alvo utilizando o modelo apresentado na Figura 3.8 com posicionamento do alvo em 5000 no eixo X a partir do tempo 10 e 100 no eixo Z a partir do tempo 5, ambos iniciados em 0.

F1 da Tabela 3.1 gerou os resultados de simulação apresentados na Figura 3.29 utilizando como parâmetros do alvo em 5000 no eixo X a partir do tempo 10 e 100 no eixo Z a partir do tempo 5. Isso representa uma missão de viagem até a posição 5000 de GPS a uma altura de 100. Nessa simulação percebemos o tratamento da ordem do bloco de rastreamento para que se viaje a 5000 unidades de deslocamento no eixo X por unidade de tempo (interpretado como 5000m/s, intencionalmente absurdo) sendo tratada pelo bloco de direcionamento conforme leis operacionais que restringem esse comando à velocidade máxima de cruzeiro. O mesmo fenômeno ocorre no eixo Z. Note que este modelo não trata o estado misto de transição entre o estado de decolagem/pouso e o estado de cruzeiro. A característica degrau do avanço das posições deve-se ao modelo da *airframe* apenas representar a aceitação da velocidade de entrada, sem implementar um controle de suas dinâmicas.

A implementação apresentada na Figura 3.14 utilizando os componentes A3, R2, O2, F2 da Tabela 3.1 gerou os resultados de simulação apresentados na Figura 3.30 utilizando como parâmetros do alvo em 5 no eixo X a partir do tempo 100 e 20 no eixo Z a do tempo 60 ao tempo 140, tendo início com X em 0 e Z em 10. Isso representa uma missão de viagem até a posição 5 de GPS a uma altura de 20. Nessa simulação percebemos a influência do controle não regulado das dinâmicas da *airframe* em seu comportamento, o que gerou um distanciamento crescente da aeronave em relação ao alvo. Ainda, percebemos que o bloco de direcionamento não está tratando adequadamente a leitura do bloco de rastreamento, uma vez que um aumento no distanciamento entre a *airframe* e o alvo não gera uma mudança de sentido no deslocamento da *airframe*. Apesar de a verificação do controle dinâmico da *airframe* não ser coberto por este trabalho, destacamos a necessidade de ajuste em leis operacionais a serem aqui verificadas.



(a) Aproximação da *airframe* ao alvo em relação ao eixo X
(b) Aproximação da *airframe* ao alvo em relação ao eixo Z

Figura 3.30: Simulações da aproximação da *airframe* em relação ao alvo utilizando o modelo apresentado na Figura 3.14 com posicionamento do alvo em 5 no eixo X a partir do tempo 100 e 20 no eixo Z a do tempo 60 ao tempo 140, tendo início com X em 0 e Z em 10.

Capítulo 4

Tradução automática de modelos Simulink para NuXMV

Modelos de sistemas em Simulink são legíveis, práticos e possuem a vantagem de serem executáveis para simulações. Essas características tão positivas durante o desenvolvimento de um projeto possuem como contrapartida a geração de um código extenso com diversas informações sobre o funcionamento das próprias simulações e da exibição dos blocos graficamente. Ferramentas para a verificação de modelos utilizam entradas o mais concisas possível, uma vez que o número de estados representados segue uma ordem superior à polinomial em relação ao tamanho da entrada. Assim, faz-se necessária a tradução dos modelos Simulink em modelos que sejam simplificados a ponto de serem verificáveis em tempo hábil e verossímeis a ponto de possuírem cobertura das características do modelo original.

As tentativas realizadas no projeto PASARP em se traduzir os modelos de forma manual geraram códigos de baixa complexidade que ou expressavam apenas fração das funcionalidades do modelo original ou que traduziam modelos sintéticos muito distantes dos realistas. Foi percebido que traduzir manualmente blocos individuais é um trabalho possível, mas modelar de forma completa suas interações pode se tornar impraticável pela extensão do número de situações geradas pelas interações e conexões entre eles. Dessa forma, foi desenvolvida uma ferramenta que utiliza-se de traduções manuais de uma biblioteca de blocos como acessório para uma tradução automática que cubra toda a relação entre eles. Uma vantagem de se descrever os blocos individualmente e utilizá-los para a composição de modelos maiores é que a sua correteude pode ser verificada individualmente, de forma que se prove que seu comportamento corresponde ao esperado pelo bloco que ele modela. A respeito da complexidade da interação entre os blocos, Postma et al [Postma, 2015] apresenta dificuldade em identificar o fluxo de dados

no Simulink através da abordagem de conectar blocos usando linhas de sinal em modelos complexos. Isso devido ao uso de pares *goto/from* e blocos de memória *data store*. Tais situações foram modeladas através do controle de escopo de variáveis, instanciando *data stores* como variáveis pertencentes aos escopos hierárquicos correspondentes. O uso de pares *goto/from* não foi traduzido por consistir em uma prática de implementação desaconselhada e pelos problemas no fluxo de dados apresentado na Seção 2.1.

4.1 A ferramenta *Simulink to Model Checking Translator*

O sistema SMCT (Simulink to Model Checking Translator) é uma ferramenta que, dada uma entrada em formato .mdl (Simulink Model File), gera um arquivo em linguagem de verificação formal (como o XMV) que representa os blocos contidos no modelo de entrada e suas respectivas conexões. Essa abstração é realizada tratando os blocos como *Modules* (módulos) com seus atributos, entradas e saídas modelados como variáveis. O assinalamento do comportamento de cada variável inerente a um módulo é realizado considerando o comportamento esperado daquele atributo de bloco Simulink em um sistema genérico. Exemplos dessa abstração são: o bloco de soma, com duas entradas e uma saída sendo convertido em uma operação de soma com dois somandos e uma soma; o bloco de ganho com uma entrada e uma saída sendo convertido em uma operação de multiplicação por constante. Abstrações mais complexas como o bloco integrador e o controlador PID carecem que se tornem explícitas as decisões de implementação para que a abstração do sistema seja entendida pelo usuário. Essas decisões de implementação são apresentadas em comentários nos arquivos de saída da ferramenta SMCT.

A ferramenta SMCT possui como finalidade possibilitar a verificação de sistemas descritos em linguagem de blocos Simulink de forma automática em duas etapas (tradução e verificação), tornando possível tal verificação por projetistas que não possuam vasto conhecimento na área de Verificação Formal. Os blocos traduzidos possuem sua funcionalidade verificada através de propriedades que descrevem seu comportamento esperado com perguntas simples como: "Um bloco de soma deve possuir sua saída com valor igual à soma de suas entradas" e "Um bloco de memória deve sempre conter em sua saída o valor da entrada anterior, ou o valor de inicialização atribuído no estado inicial", de modo a assegurar ao usuário a corretude da ferramenta.

A implementação do SMCT foi realizada tendo em vista a organização do arquivo de modelo Simulink que pode ser dividido em Cabeçalho, Definição de Padrões e Instância dos Blocos. No Cabeçalho se define o nome do sistema e diversos outros parâmetros que

não são interessantes para o processo de verificação como especificações para o motor gráfico a ser utilizado. Na Definição de Padrões tem-se uma declaração de todos os tipos de blocos utilizados no sistema e dos valores-padrão de seus parâmetros. A definição de valores-padrão para os parâmetros é uma etapa importante no entendimento do formato .mdl uma vez que na Instância dos Blocos é comum se omitir a definição de parâmetros, gerando inconsistências na análise sintática da tradução. Na Instância de Blocos, tem-se a declaração de cada bloco com seu nome, um identificador único (*sid*) e um conjunto de parâmetros. Tanto na Definição de Padrões quanto na Instância dos Blocos, tem-se uma grande irregularidade no número e formato de parâmetros, mesmo entre blocos do mesmo tipo. Tal composição gramatical complexa gera desafios na tradução que foram resolvidos utilizando uma técnica que: 1. Gera uma instância-padrão na etapa de Definição de Padrões para cada bloco; 2. Clona essa instância para cada novo bloco instanciado; 3. Atualiza os parâmetros de cada nova instância através de um analisador sintático e semântico específico a cada bloco gerado.

A cobertura do SMCT pode ser avaliada enquanto largura e profundidade. Nesse sentido, entenda-se largura como o número de blocos já modelados na ferramenta. As bibliotecas Simulink possuem milhares de tipos de blocos, onde ainda é comum a criação de tipos de blocos personalizados, tornando inviável uma cobertura completa em termos de largura. Ainda assim, o número de blocos utilizados na prática em projetos de um domínio específico, qual sistemas de aeronave, é restrito tornando factível uma cobertura em largura pseudo-completa. A cobertura em profundidade da ferramenta é avaliada enquanto sua capacidade de abranger diferentes arquiteturas de projeto, com derivações em subsistemas, declaração de variáveis com hierarquia cruzada, instâncias múltiplas de subsistemas, entre outras. Nesse sentido, a cobertura do SMCT é completa, com a exceção do tratamento de estruturas GOTO-FROM, que são mecanismos de desvio de fluxo implícito. Percebemos que o uso de desvio de fluxo implícito dificulta o processo de tradução pois esconde relações de hierarquia cruzada. Desse modo recomendamos uma diretriz de projeto que apenas utilize fluxo explícito de dados nos modelos para simplificar o processo de tradução. Com o intuito de expandir constantemente a cobertura em largura do SMCT, os blocos desconhecidos são normalmente instanciados com suas conexões realizadas mas sem uma implementação funcional, de modo que o arquivo de saída indica quais blocos ainda não foram implementados para implementação manual. Nesse caso, sugere-se que se envie ao desenvolvedor a listagem de blocos não implementados para que sejam agregados a uma próxima versão da ferramenta.

Condições para a verificação são dadas por leis operacionais e leis de controle, constantes nos manuais de voo pertencentes a cada modelo de aeronave. Tais condições incluem margens de segurança para variáveis da aeronave (e.g. Velocidade Não Exceder

(VNE)) e cenários de operação (e.g. a decolagem não pode gerar uma oscilação maior do que a referência para não gerar desconforto no passageiro). Os cenários de operação, tipicamente embarque, decolagem e cruzeiro, devem ser mutuamente excludentes com cada atuador sendo controlado por um único controlador.

4.2 Algoritmo e Implementação da tradução automática de Simulink para NuXMV

O paradigma de Orientação a Objetos foi escolhido para implementar a ferramenta devido à correspondência entre os tipos de blocos com seus parâmetros e a estrutura de classes com atributos. A Figura 4.1 apresenta a correspondência entre as linguagens Simulink, Java (como implementação no paradigma de Orientação a Objetos) e NuXMV no processo de tradução dos blocos e seus sinais em módulos e suas variáveis. Para cada parâmetro relevante dos tipos de blocos é criado um atributo da classe correspondente. A atualização dos valores dos parâmetros é realizada através de uma função que recebe uma dupla (nome do parâmetro, valor do parâmetro), de modo que identifica-se o atributo correspondente na classe e realiza-se a carga de valor. Essa função é importante pois transfere a cada classe a responsabilidade da análise sintática e semântica do código fonte (.mdl) correspondente ao conteúdo de cada bloco. Desse modo simplifica-se a leitura do complexo polimorfismo inerente ao formato .mdl. Cada classe implementa também dois construtores: o primeiro utiliza o nome do bloco para sua instância. O segundo clona um objeto (usualmente o bloco padrão carregado na fase de Definição de Padrões) para posterior atualização de seus parâmetros. A ambos construtores associa-se uma função de inicialização que prepara a instância do bloco tanto na instância de seus atributos quanto na preparação de nomes de saída e definição de posição de declaração.

O direcionamento de código para a análise sintática e semântica por parte das classes que representam os blocos é feito por uma função principal. Essa função cerca o escopo de um módulo que representa o sistema como um todo. Ao encontrar um bloco de subsistema a função é invocada novamente de forma recursiva, representando o aspecto hierárquico de blocos Simulink de forma fiel.

A geração de código xmv então é realizada pela execução de uma função de cada bloco encontrado que retorna seu código de declaração ou de definição de módulo, todos posicionados corretamente no código xmv a partir de uma variável de controle definida na função de inicialização de cada classe de bloco.

A ferramenta SMCT segue um algoritmo base que se divide em sete passos para a transformação do código Simulink em código-objeto para NuXMV. Seu algoritmo seg-

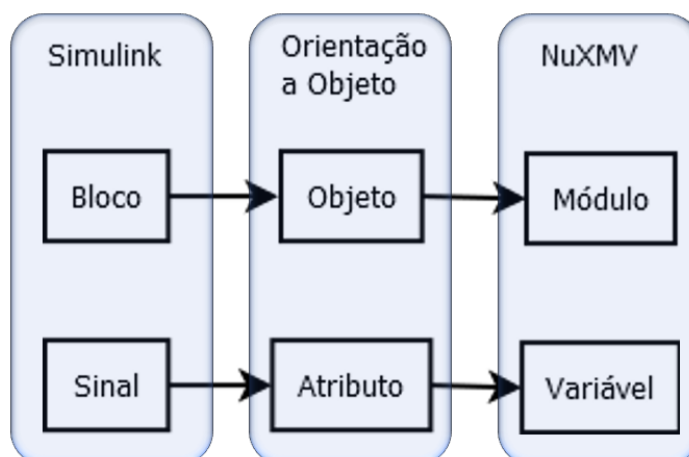


Figura 4.1: O algoritmo de tradução utiliza da relação entre blocos Simulink e módulos NuXMV, modelados intermediariamente como objetos de classes Java. De forma análoga, os sinais no Simulink são relacionados a variáveis NuXMV, modelados intermediariamente como atributos com seus valores assinalados nos objetos Java.

menta o código-fonte, utilizando-se do padrão textual dos códigos *.mdl*, em: i. cabeçalho; ii. definição de padrões; iii. instância de blocos; iv. conexões. O **Passo 1** busca a definição do nome do sistema e ignora todo o código anterior, referente a estilos gráficos, definições de simulação e outras definições não estruturais. Depois, busca o início do segmento de definição de padrões. O **Passo 2** busca o lexema que define o assinalamento de um tipo de bloco a ser definido. Ele então busca em uma biblioteca de tipos de bloco se aquele tipo já foi definido. Se sim, cria uma instância em linguagem intermediária de um modelo-padrão, atualizando seus parâmetros de acordo com os assinalamentos apresentados no código-fonte. Se não, cria a instância de um bloco padrão, assinalando que tal bloco ainda não foi implementado e carrega seus parâmetros sem processá-los. Uma vez processados todos os padrões, busca o lexema referente ao início da instanciação de blocos. O **Passo 3** busca cada instância dos blocos no código-fonte e para cada uma delas gera uma cópia do bloco-padrão correspondente, atualizando seus novos parâmetros como o identificador único (*sid*). Caso o bloco seja um subsistema, sua instância consiste em uma chamada especial dos Passos 3-4 (**Passo 5**). Uma vez processadas todas as instâncias, busca o lexema referente ao início das declarações de conexões entre os blocos. O **Passo 4** busca as linhas de conexão entre os blocos no código-fonte, utilizando as referências, *sids*, conectadas para estabelecer as entradas e saídas dos blocos já instanciados. Essas referências se tornam os parâmetros e retornos dos códigos NuXMV gerados como código-objeto. Uma vez processadas as linhas de conexão, procede-se para a geração de código NuXMV, **Passo 6**, com um modelo próprio para cada tipo de bloco, cuja conversão é implementada na classe correspondente pertencente à biblioteca de modelos da ferramenta SMCT.

Essa conversão baseia-se na funcionalidade mapeada de cada bloco, com especificações de implementação extraídas da referência oficial do Simulink [The Mathworks, 2020]. O

Passo 7 gera os arquivos de saída da ferramenta, contendo código-objeto e termina o programa. Em resumo, utiliza-se os seguintes passos:

1. A partir do segmento de cabeçalho derivar o nome do sistema e descartar códigos inúteis;
2. A partir do segmento de definição de padrões, para cada tipo de bloco encontrado gerar uma instância-padrão e assinalar seus valores;
3. A partir do segmento de instância de blocos, para cada bloco encontrado gerar uma cópia da instância-padrão correspondente e atualizar seus parâmetros;
4. A partir do segmento de conexões, para cada linha que conecta blocos encontrada assinalar as entradas e saídas de cada bloco;
5. Para cada subsistema repetir os Passos 1-5;
6. Para cada bloco, agora definido com seus parâmetros, entradas e saídas, gerar uma substituição utilizando um código para NuXMV correspondente;
7. Gerar o código-objeto e terminar.

4.3 Blocos Simulink traduzidos automaticamente pela SMCT

Esta seção trata dos blocos Simulink [The Mathworks, 2020] cobertos pela ferramenta enquanto automaticamente traduzidos. Para cada bloco ela traz seu Nome seguido do Nome do bloco encontrado no Simulink; Descrição das funcionalidades do bloco, com sua biblioteca; Implementação XMV do bloco; Propriedade para validação da implementação realizada. Uma descrição mais completa da estrutura dos blocos estende esta seção no Apêndice A, objetivando mais clara compreensão das decisões e estratégias, bem como metodologia de tradução.

4.3.1 Seletor de Barramentos (Bus Selector)

O Seletor de Barramentos, disponível na biblioteca **Simulink / Commonly Used Blocks**, seleciona sinais de um barramento de entrada. O Seletor de Barramentos no Simulink recebe como entrada um barramento que condensa diversos sinais e decompõe os sinais

em um conjunto de saídas. Ele é entendido como um vetor em sua entrada que possui cada uma de suas posições atribuídas a uma variável de saída. Mais propriamente, o Seletor de Barramentos tem como saída um subconjunto específico de elementos do barramento em sua entrada. O bloco pode ter como saída um conjunto de sinais separados (como foi modelado para esta ferramenta) ou um novo barramento. A ordenação dos sinais em um Seletor de Barramentos é sempre mantida da entrada para a saída.

Implementação XMV

Declaração

nome_bloco: *block_nome_bloco(barramento_entrada)*

Especificação do módulo

MODULE *block_nome_bloco(barramento_entrada)*

ASSIGN

[PARA CADA SINAL NA ENTRADA] *nome_sinal* :=
barramento_entrada[posicao_sinal]

Propriedade para validação

G(nome_sinal=barramento_entrada[posicao_sinal])

4.3.2 Comparação a constante (Compare to Constant)

O bloco de Comparação a Constante, disponível na biblioteca **Simulink / Logic and Bit Operations** determina como um sinal de entrada se compara a uma constante especificada. O operador de comparação é parametrizável entre os relacionais básicos (igualdade, maior, menor, igual ou maior, igual e menor, diferença). A saída do bloco é o correspondente binário 1 se a comparação foi verdadeira e 0 se a comparação for falsa.

Implementação XMV

Declaração

relop_nome_bloco := *sinal_entrada operador constante*

Propriedade para validação

G(relop_nome_bloco = (sinal_entrada operador constante))

4.3.3 Constante (Constant)

O bloco Constante, disponível na biblioteca **Simulink / Commonly Used Blocks**, gera um sinal de valor constante real ou complexo como escalar, vetor ou matriz. O valor constante do sinal é definido através de um parâmetro do bloco.

Implementação XMV

Declaração

con_nome_bloco : *valor_constante*

Propriedade para validação

$G(\text{con_nome_bloco} = \text{valor_constante})$

4.3.4 Demultiplexador (Demux)

O bloco Demultiplexador, disponível na biblioteca **Simulink / Commonly Used Blocks**, extrai os componentes de um vetor de sinais de entrada, separando os seus componentes como saídas.

Implementação XMV

Declaração

nome_bloco : *block_nome_bloco(barramento_entrada)*

Especificação do módulo

MODULE *block_nome_bloco(barramento_entrada)*

 ASSIGN

 [PARA CADA SINAL NA ENTRADA] *nome_sinal* :=
barramento_entrada[posicao_sinal]

Propriedade para validação

$G(\text{nome_sinal} = \text{barramento_entrada}[\text{posicao_sinal}])$

4.3.5 Ganho (Gain)

O bloco Ganho, disponível na biblioteca **Simulink / Commonly Used Blocks**, multiplica a entrada por um valor constante. A entrada pode ser um escalar, vetor ou matriz. O fator de multiplicação é informado como um parâmetro.

Implementação XMV

Declaração

*gain_nome_bloco: entrada*ganho*

Propriedade para validação

$G(\text{gain_nome_bloco}=(\text{entrada}*\text{ganho}))$

4.3.6 Entrada (Inport)

O bloco de Entrada, disponível na biblioteca **Commonly Used Blocks**, liga sinais de fora de um sistema ao sistema que o contém, hierarquicamente inferior. Apesar de possuir apenas a função de ligar dois pontos é um elemento estrutural na parametrização de funções entre os blocos.

Implementação XMV

Declaração (Normalmente interpretado como parâmetro de um módulo)

in_nome_bloco := entrada

Propriedade para validação

$G(\text{in_nome_bloco}=\text{entrada})$

4.3.7 Integrador pelo método de somas discretas (Integrator)

O bloco Integrador, disponível na biblioteca **Simulink / Commonly Used Blocks**, gera como saída o valor da integral de sua entrada em relação ao tempo. A saída é calculada através da soma discreta da variável de integração (integrando) a cada unidade de tempo.

Implementação XMV

Declaração

init(int_nome_bloco): condicao_inicial

next(int_nome_bloco): int_nome_bloco + integrando

Propriedade para validação

int_nome_bloco=condicao_inicial

$G(\text{next}(\text{int_nome_bloco})=\text{int_nome_bloco} + \text{integrando})$

4.3.8 Lógico (Logical Operator)

O bloco Lógico, disponível na biblioteca **Commonly Used Blocks** realiza a operação lógica especificada entre suas entradas. Uma entrada é considerada verdadeira se diferente de zero e falsa se igual a zero. A saída é gerada a partir da realização da operação lógica, definida em um parâmetro, entre as entradas, gerando a saída com valor lógico binário correspondente.

Implementação XMV

Declaração

log_nome_bloco := entrada1 [Para cada entrada] operador entradaN

Propriedade para validação

G(log_nome_bloco=(entrada1 [Para cada entrada] operador entradaN))

4.3.9 Memória (Memory)

O bloco de Memória, disponível na biblioteca **Simulink / Discrete**, sustenta sua entrada como atraso por um passo de tempo. Blocos de memória são importantes em aplicações que necessitem de valores anteriores (como acumuladores) ou que lidem com problemas de sincronização.

Implementação XMV

Declaração

nome_bloco : block_memory(condicao_inicial, entrada)

Especificação do módulo

MODULE *block_memory(condicao_inicial, entrada)*

ASSIGN

init(out) := condicao_inicial

next(out) := entrada

Propriedade para validação

out=condicao_inicial

G(next(out)=(entrada))

4.3.10 Multiplexador (Mux)

O bloco Multiplexador, disponível na biblioteca **Simulink / Commonly Used Blocks** combina suas entradas em um único vetor de saída. Ele possui uma função importante inclusive na organização do projeto, condensando barramentos em linhas únicas que são demultiplexadas nos sistemas que as utilizam.

Implementação XMV

Declaração

textitnome_bloco : *block_nome_bloco*([Lista de entradas do multiplexador])

Especificação do módulo

MODULE *block_nome_bloco*([Lista de entradas do multiplexador])

ASSIGN

[PARA CADA ENTRADA] *out*[*i*] := *entrada*_{*i*};

Propriedade para validação

[PARA CADA ENTRADA] $G(out[i] = entrada_i)$

4.3.11 Saída (Outport)

O bloco de saída, disponível na biblioteca **Simulink / Commonly Used Blocks**, liga um sinal de um sistema para fora desse sistema, fornecendo saídas para níveis hierarquicamente superiores. Apesar de não ser modelado como uma função em si, é extremamente importante como elemento de retorno de subsistemas.

Implementação XMV

Declaração

out_nome_bloco := *entrada*

Propriedade para validação

$G(out_nome_bloco=entrada)$

4.3.12 Controlador PID pelo método de Euler (PID Controller)

O bloco Controlador PID implementa um controlador PID, PI, PD, P ou I. Sua implementação foi baseada na realizada por Vilhena [Vilhena, 2018].

Implementação XMV

Declaração

nome_bloco : *block_nome_bloco*(*error*)

Especificação do módulo

MODULE *block_nome_bloco*(*erro*)

VAR

integrator : Real;

previous_error : Real;

ASSIGN

init(*integrator*) := *initial_integrator*;

next(*integrator*) := $I * error + integrator$;

init(*previous_error*) := *initial_error*;

next(*previous_error*) := *error*;

DEFINE

$out := (P * error + I * error + integrator + D * (error - previous_error))$

Propriedade para validação

Este módulo foi validado por [Vilhena, 2018] através da obtenção de triplas P, I, D em contraexemplos que, considerando tolerâncias variáveis no erro do controlado, correspondiam a valores encontrados por métodos clássicos na literatura.

4.3.13 Produto (Product)

O bloco Produto, disponível na biblioteca **Simulink / Commonly Used Blocks**, realiza a multiplicação ou divisão de duas entradas. A operação a ser realizada é definida em um dos parâmetros, bem como seus detalhamentos.

Implementação XMV

(*prod_nome_bloco*) := [Para cada entrada, seu nome e operação (multiplicação ou divisão)];

Propriedade para validação

$G(prod_nome_bloco = ((Para\ cada\ entrada,\ seu\ nome\ e\ operação\ (multiplicação\ ou\ divisão))))$

4.3.14 Operador relacional (Relational Operator)

O bloco Operador Relacional, disponível na biblioteca **Simulink / Commonly Used Blocks**, realiza a operação relacional especificada sobre a entrada. A operação a ser realizada é definida através de um parâmetro (igualdade, maior, menor, igual ou maior, igual e menor, diferença) e a saída é o resultado binário sobre a operação realizada.

Implementação XMV

Declaração

```
relop_nome_bloco := entrada1 operacao entrada2;
```

Propriedade para validação

```
G(relop_nome_bloco = (entrada1 operacao entrada2))
```

4.3.15 Saturação (Saturation)

O bloco Saturação, disponível na biblioteca **Simulink / Commonly Used Blocks**, produz um sinal de saída que é o sinal da entrada limitado a valores de saturação superior e inferior. Ele é importante para representar fenômenos físicos submetidos a limites, além de ser importante para verificações de violação de limiar.

Implementação XMV

Declaração

```
nome_bloco : block_nome_bloco(entrada)
```

Especificação do módulo

```
MODULE block_nome_bloco(entrada)
```

```
DEFINE
```

```

    out := case
      entrada > upper_limit : upper_limit;
      entrada < lower_limit : lower_limit;
      TRUE : entrada;
    esac;
```

Propriedade para validação

```
G((out <= upper_limit) & (out >= lower_limit))
```

4.3.16 Soma (Sum)

O bloco de Soma, disponível na biblioteca **Simulink / Math Operations**, realiza a soma ou subtração de suas entradas. O tipo da operação é definida através de um parâmetro.

Implementação XMV

Declaração

sum_nome_bloco := [Para cada entrada, seu nome e operação (soma ou subtração)];

Propriedade para validação

$G(\text{sum_nome_bloco} = ([\text{Para cada entrada, seu nome e operação (soma ou subtração)}]))$

4.3.17 Comutador (Switch)

O bloco Comutador, disponível na biblioteca **Simulink / Commonly Used Blocks**, transmite uma de suas entradas à saída dependendo do resultado de uma operação de controle. Ele é um dos principais componentes de desvio de fluxo, aparecendo predominantemente em unidades de controle.

Implementação XMV

Declaração

nome_bloco : *block_switch_simple(entrada_v,erdadeiro,controle,entrada_falso)*;

Especificação do módulo

MODULE *block_switch_simple(entrada_v,erdadeiro,controle,entrada_falso)*

DEFINE

```

    out := case
    controle : entradaverdadeiro;
    !controle : entradafalso;
    esac;
```

Propriedade para validação

$G((\text{controle}\&(\text{out}=\text{entrada}_v\text{erdadeiro}))|(!\text{controle}\&(\text{out}=\text{entrada}_f\text{also})))$

Capítulo 5

Tradução e verificação de projetos de sistemas autônomos

Este capítulo apresenta resultados da tradução automática pela ferramenta SMCT de três modelos, dois referentes ao controle da orientação de uma aeronave (O1 e O2 da Tabela 3.1) e um referente ao controle de seu rastreamento (R1 da Tabela 3.1) e que são apresentados no Apêndice A. Esses modelos foram escolhidos por possuírem a complexidade de implementação de sistemas completos condensada em um tamanho apresentável neste texto.

Os códigos apresentados como resultados possuem uma estrutura com os seguintes componentes em ordem:

MODULE - Declaração dos módulos que representa a conectividade dos blocos modelados através da sua parametrização, bem como seu identificador para instâncias. O módulo "MAIN" de cada modelo foi criado manualmente para gerar uma instância dos blocos que pudesse ser verificada.

VAR - Declaração das variáveis que apresenta outros blocos aos quais o bloco modelado se conecta, bem como os sinais que realizam as conexões.

DEFINE - Definição dos sinais que realiza a declaração de constantes, parâmetros de inicialização, além de operações relacionais e lógicas simples.

Conjunto de especificações **LTLSPEC** - Apresenta as propriedades de validação do comportamento do modelo gerado em relação ao esperado pelo bloco modelado. Essas propriedades referem-se ao correto funcionamento dos blocos e consequentemente da sua integração. Propriedades triviais, como a verificação de blocos constantes, são apresentadas para fins de completude.

```
-- no counterexample found with bound 990
-- no counterexample found with bound 991
-- no counterexample found with bound 992
-- no counterexample found with bound 993
-- no counterexample found with bound 994
-- no counterexample found with bound 995
-- no counterexample found with bound 996
-- no counterexample found with bound 997
-- no counterexample found with bound 998
-- no counterexample found with bound 999
-- no counterexample found with bound 1000
```

Figura 5.1: Trecho da saída da ferramenta NuXMV atestando o cumprimento das propriedades verificadas pelo modelo utilizando BMC até uma profundidade $k=1000$.

Os modelos foram verificados utilizando SMT (*Satisfiability Modulo Theories*) através da técnica de BMC (*Bounded Model Checking*) até uma profundidade $k = 1000$ (uma profundidade computável em tempo razoável e com cobertura suficiente, já que nas execuções realizadas contraexemplos são encontrados até uma profundidade $k = 10$). Para isso foi utilizada a ferramenta NuXMV no modo iterativo **NuXMV.exe -int**, com construção do modelo através do comando **go_msat** e sua verificação através do comando **msat_check_ltlspec_bmc -k 1000**. Nenhum dos modelos verificados apresentou contraexemplos, gerando uma saída da ferramenta como a apresentada na Figura 5.1, validando as propriedades descritas até o limiar especificado.

Os resultados apresentados demonstram a execução completa do fluxo da geração à verificação apresentado na Seção 3.1 ao traduzir modelos gerados a partir da estrutura de abstração definida utilizando a ferramenta SMCT e verificando os resultados utilizando a ferramenta NuXMV. Eles são um indicador positivo no sentido de se verificar sistemas complexos como os de controle de voo autônomo, especialmente por associar técnicas automáticas de diferentes domínios, o que diminui a exigência multidisciplinar dos especialistas encarregados da detecção de falhas nos modelos.

5.1 Tradução do subsistema de direção para o rastreador R1

O subsistema de direção para o rastreador R1 com código apresentado na Seção do Anexo A.1 foi traduzido recebendo como sinais de entrada as variações nos eixos cartesianos X e Z e trazendo um conjunto de módulos *switch* que realiza o desvio de fluxo sobre qual dos sinais constantes definidos será levado ao comando de saída. A decisão é tomada

considerando uma ordem de prioridade dos comandos e os valores das variações nos eixos. As propriedades de validação verificam se o comando gerado respeita os sinais de entrada e a ordem de precedência dos comandos de saída.

```

01. MODULE main
02.   VAR
03.     dXe : integer;
04.     dZe : integer;
05.     instancia_calculo_direcao: calculo_direcao(dXe,dZe);

06. MODULE calculo_direcao(in_dXe, in_dZe)
07.   VAR
08.     SedeltaZe0vaparacima : block_switch_simple(con_Cima,relop_RelationalOperator4,
SedeltaZe0vaparabaixoSenaoparar.out);
09.     SedeltaZe0vaparabaixoSenaoparar : block_switch_simple(con_Baixo,
relop_RelationalOperator3,con_Parar);
10.     Memory2 : block_memory(0,SedeltaXe0vaparadireita.out);
11.     SedeltaXe0vaparadesquerda :
block_switch_simple(con_Esquerda,relop_RelationalOperator1,SedeltaZe0vaparacima.out);
12.     SedeltaXe0vaparadireita :
block_switch_simple(con_Direita,relop_RelationalOperator2,SedeltaXe0vaparadesquerda.out);
13.   DEFINE
14.     con_0 := 1;
15.     con_1 := -1;
16.     con_Baixo := 2;
17.     con_Cima := 1;
18.     con_Direita := 4;
19.     con_Esquerda := 3;
20.     con_Parar := 0;
21.     out_Comando := Memory2.out;
22.     relop_RelationalOperator4 := in_dZe > con_0;
23.     relop_RelationalOperator3 := in_dZe < con_1;
24.     relop_RelationalOperator2 := in_dXe > con_0;
25.     relop_RelationalOperator1 := in_dXe < con_1;
26. LTLSPEC G(relop_RelationalOperator4 <-> (in_dZe > con_0))
27. LTLSPEC G(relop_RelationalOperator3 <-> (in_dZe < con_1))
28. LTLSPEC G(relop_RelationalOperator2 <-> (in_dXe > con_0))
29. LTLSPEC G(relop_RelationalOperator1 <-> (in_dXe < con_1))
30. LTLSPEC G(con_0 = 1)

```

```

31.  LTLSPEC G(con_1 = -1)
32.  LTLSPEC G(con_Baixo = 2)
33.  LTLSPEC G(con_Cima = 1)
34.  LTLSPEC G(con_Direita = 4)
35.  LTLSPEC G(con_Esquerda = 3)
36.  LTLSPEC G(con_Parar = 0)

37.  MODULE block_switch_simple(in_true, criteria, in_false)
38.  DEFINE
39.    out := case
40.      criteria : in_true;
41.      !criteria : in_false;
42.      TRUE: in_false;
43.    esac;
44.  LTLSPEC G(criteria&(out=in_true)|!criteria&(out=in_false))

45.  MODULE block_memory(in_init, in_data)
46.  VAR
47.    out : integer;
48.  ASSIGN
49.    init(out) := in_init;
50.    next(out) := in_data;
51.  LTLSPEC (out = in_init)
52.  LTLSPEC G(in_data = next(out))

```

5.2 Tradução do subsistema de estado para a orientação O1

O subsistema de estado para a orientação O1 com código apresentado na Seção do Anexo A.3 foi traduzido recebendo como sinais de entrada o comando gerado pelo cálculo da direção e a altura da aeronave (eixo Z). O comando é comparado a um conjunto de valores constantes que representam diferentes tomadas de decisão de ação. A altura da aeronave é comparada a valores de limiar para ligação dos motores e voo de cruzeiro. Um conjunto de operações lógicas e relacionais é utilizado para a decisão, através de um conjunto de blocos *switch*, de qual o estado da aeronave será utilizado como saída, também modelado por um conjunto de constantes. As propriedades de validação verificam se o estado con-

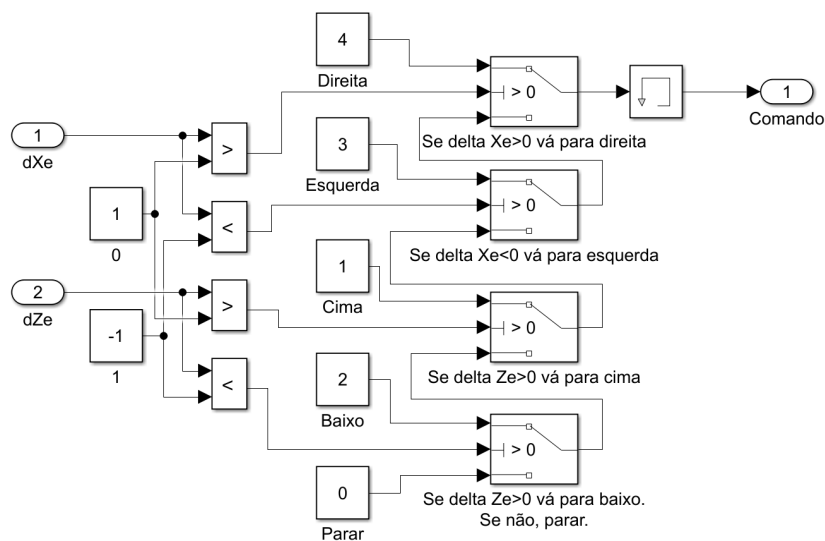


Figura 5.2: Modelo apresentado na Figura 3.19 traduzido na Seção 5.1. Nele, as constantes 0 e 1 definem o limiar percepção da variação das entradas dXe e dZe . As constantes *Direita*, *Esquerda*, *Cima*, *Baixo* e *Parar* são as constantes direcionais para a geração da saída do bloco. No código traduzido foi criado o módulo *main* manualmente para representar as entradas dXe e dZe . As constantes de limiar foram modeladas nas linhas 14-15 e verificadas nas linhas 30-31. As operações relacionais entre as entradas e as análises de limiar são modeladas nas linhas 22-25 e verificadas nas linhas 25-28. As constantes direcionais são modeladas nas linhas 16-20 e verificadas nas linhas 32-36. Os *switches* que determinam o comando a ser armazenado pela memória são modelados nas linhas 08-12 e instanciam o módulo definido nas linhas 37-44. O bloco de memória que armazena o comando decidido pelo bloco é modelado na linha 10 e instancia o módulo definido nas linhas 45-52. A saída do bloco, *Comando*, é modelada na linha 21.

diz com o comando de entrada, respeitando leis operacionais que restringem operações dos motores em alturas especificadas.

```

01. MODULE main
02.   VAR
03.     comando : {0,1,2,3,4};
04.     z : integer;
05.     instancia_calculo_estado_orientacao_cartesiana:
calculo_estado_orientacao_cartesiana(comando,z);

06. MODULE calculo_estado_orientacao_cartesiana(in_Comando, in_Z)
07.   VAR
08.     Condiçãodenavepousada : block_switch_simple(con_Navepousada,
log_LogicalOperator,Condiçãodenaveestatica.out);
09.     Condiçãoderecusahorizontal : block_switch_simple(con_Recusahorizontal,

```

```

log_LogicalOperator8,Condicadodecolagem.out);
10.    Condiçãodetransicao : block_switch_simple(con_Transicao,log_LogicalOperator6,
Condiçãoderecusahorizontal.out);
11.    Condiçãodenaveestatica : block_switch_simple(con_Naveestatica,
log_LogicalOperator1,Condiçãodecruzeiro.out);
12.    Condiçãodecruzeiro : block_switch_simple(con_Cruzeiro,log_LogicalOperator3,
Condiçãodetransicao.out);
13.    Condiçãodepouso : block_switch_simple(con_Pouso,relop_RelationalOperator8,
con_Desconhecido);
14.    Condiçãodedecolagem : block_switch_simple(con_Decolagem,
relop_RelationalOperator7,Condiçãodepouso.out);
15.    DEFINE
16.    con_Alturadecruzeiro := 304;
17.    con_Alturademotorhorizontal := 60;
18.    con_Comandodescer := 2;
19.    con_Comandoparadireita := 4;
20.    con_Comandoparaesquerda := 3;
21.    con_Comandoparar := 0;
22.    con_Comandosubir := 1;
23.    con_Cruzeiro := 2;
24.    con_Decolagem := 3;
25.    con_Desconhecido := 7;
26.    con_Naveempouso := 0;
27.    con_Naveestatica := 1;
28.    con_Navepousada := 0;
29.    con_Pouso := 5;
30.    con_Recusahorizontal := 6;
31.    con_Transicao := 4;
32.    relop_RelationalOperator4 := in_Z >= con_Alturademotorhorizontal;
33.    relop_RelationalOperator3 := in_Z >= con_Alturadecruzeiro;
34.    relop_RelationalOperator6 := in_Comando = con_Comandoparadireita;
35.    relop_RelationalOperator5 := in_Comando = con_Comandoparaesquerda;
36.    relop_RelationalOperator8 := in_Comando = con_Comandodescer;
37.    relop_RelationalOperator7 := in_Comando = con_Comandosubir;
38.    log_Comandohorizontal := relop_RelationalOperator5 | relop_RelationalOperator6;
39.    relop_RelationalOperator2 := con_Naveempouso = in_Z;
40.    out_Estado := Condiçãodenavepousada.out;
41.    relop_RelationalOperator1 := in_Comando = con_Comandoparar;
42.    log_LogicalOperator7 := !log_LogicalOperator6;

```

```

43.   log_LogicalOperator8 := log_LogicalOperator7 & log_Comandohorizontal;
44.   log_LogicalOperator := relop_RelationalOperator1 & relop_RelationalOperator2;
45.   log_LogicalOperator6 := log_Comandohorizontal & relop_RelationalOperator4;
46.   log_LogicalOperator3 := log_Comandohorizontal & relop_RelationalOperator3;
47.   log_LogicalOperator1 := log_LogicalOperator2 & relop_RelationalOperator1;
48.   log_LogicalOperator2 := !log_LogicalOperator;
49.   LTLSPEC G(relop_RelationalOperator4 = (in_Z >= con_Alturademotorhorizontal))
50.   LTLSPEC G(relop_RelationalOperator3 = (in_Z >= con_Alturadecruzeiro))
51.   LTLSPEC G(relop_RelationalOperator6 = (in_Comando = con_Comandoparadireita))
52.   LTLSPEC G(relop_RelationalOperator5 = (in_Comando = con_Comandoparaesquerda))
53.   LTLSPEC G(relop_RelationalOperator8 = (in_Comando = con_Comandodescer))
54.   LTLSPEC G(relop_RelationalOperator7 = (in_Comando = con_Comandosubir))
55.   LTLSPEC G(relop_RelationalOperator2 = (con_Naveempouso = in_Z))
56.   LTLSPEC G(relop_RelationalOperator1 = (in_Comando = con_Comandoparar))
57.   LTLSPEC G(con_Alturadecruzeiro = 304)
58.   LTLSPEC G(con_Alturademotorhorizontal = 60)
59.   LTLSPEC G(con_Comandodescer = 2)
60.   LTLSPEC G(con_Comandoparadireita = 4)
61.   LTLSPEC G(con_Comandoparaesquerda = 3)
62.   LTLSPEC G(con_Comandoparar = 0)
63.   LTLSPEC G(con_Comandosubir = 1)
64.   LTLSPEC G(con_Cruzeiro = 2)
65.   LTLSPEC G(con_Decolagem = 3)
66.   LTLSPEC G(con_Desconhecido = 7)
67.   LTLSPEC G(con_Naveempouso = 0)
68.   LTLSPEC G(con_Naveestatica = 1)
69.   LTLSPEC G(con_Navepousada = 0)
70.   LTLSPEC G(con_Pouso = 5)
71.   LTLSPEC G(con_Recusahorizontal = 6)
72.   LTLSPEC G(con_Transicao = 4)
73.   LTLSPEC G(log_Comandohorizontal = (relop_RelationalOperator5 |
relop_RelationalOperator6))
74.   LTLSPEC G(log_LogicalOperator7 = (!log_LogicalOperator6))
75.   LTLSPEC G(log_LogicalOperator8 = (log_LogicalOperator7 & log_Comandohorizontal))
76.   LTLSPEC G(log_LogicalOperator = (relop_RelationalOperator1 &
relop_RelationalOperator2))
77.   LTLSPEC G(log_LogicalOperator6 = (log_Comandohorizontal &
relop_RelationalOperator4))
78.   LTLSPEC G(log_LogicalOperator3 = (log_Comandohorizontal &

```

```

relop_RelationalOperator3))
79.  LTLSPEC G(log_LogicalOperator1 = (log_LogicalOperator2 &
relop_RelationalOperator1))
80.  LTLSPEC G(log_LogicalOperator2 = (!log_LogicalOperator))

81. MODULE block_switch_simple(in_true, criteria, in_false)
82.  DEFINE
83.    out := case
84.      criteria : in_true;
85.      !criteria : in_false;
86.      TRUE: in_false;
87.    esac;
88.  LTLSPEC G(criteria&(out=in_true)|!criteria&(out=in_false))

```

5.3 Tradução do bloco de orientação O2

O sistema de orientação O2 com código apresentado na Seção do Anexo A.2 foi traduzido recebendo como sinais de entrada para o cálculo da altitude alvo o valor de altitude comandada, as referências de GPS (das quais é utilizada apenas a coordenada Z, decomposta em um seletor de barramentos) e um conjunto de sensores de obstáculo. O cálculo da altitude alvo considera então a altitude desejada, a altitude atual e uma análise de sensores de obstáculo para determinar a seleção da altitude resultante. Para o cálculo do ângulo alvo, é utilizada uma tripla de comandos de deslocamento com uma ordem de prioridade, dada como entrada. O ângulo do alvo então é determinado como um incremento ou decremento em um valor constante, a manutenção do ângulo ou a manutenção da alteração anterior, armazenada em um bloco de memória. As propriedades de validação verificam se a altitude e o ângulo calculados como alvo respeitam o conjunto de leis operacionais definidas na sua tomada de decisão.

```

001. MODULE main
002.  VAR
003.    GPS : block_BusSelector;
004.    Pilot_Altitude : integer;
005.    Obstacle_Sensor : block_BusSelector1;
006.    BACK_CMD : boolean;
007.    FWD_CMD : boolean;
008.    Steady_Lat_CMD : boolean;
009.    instancia_orientacao_turretta: orientacao_turretta(GPS, Pilot_Altitude,

```

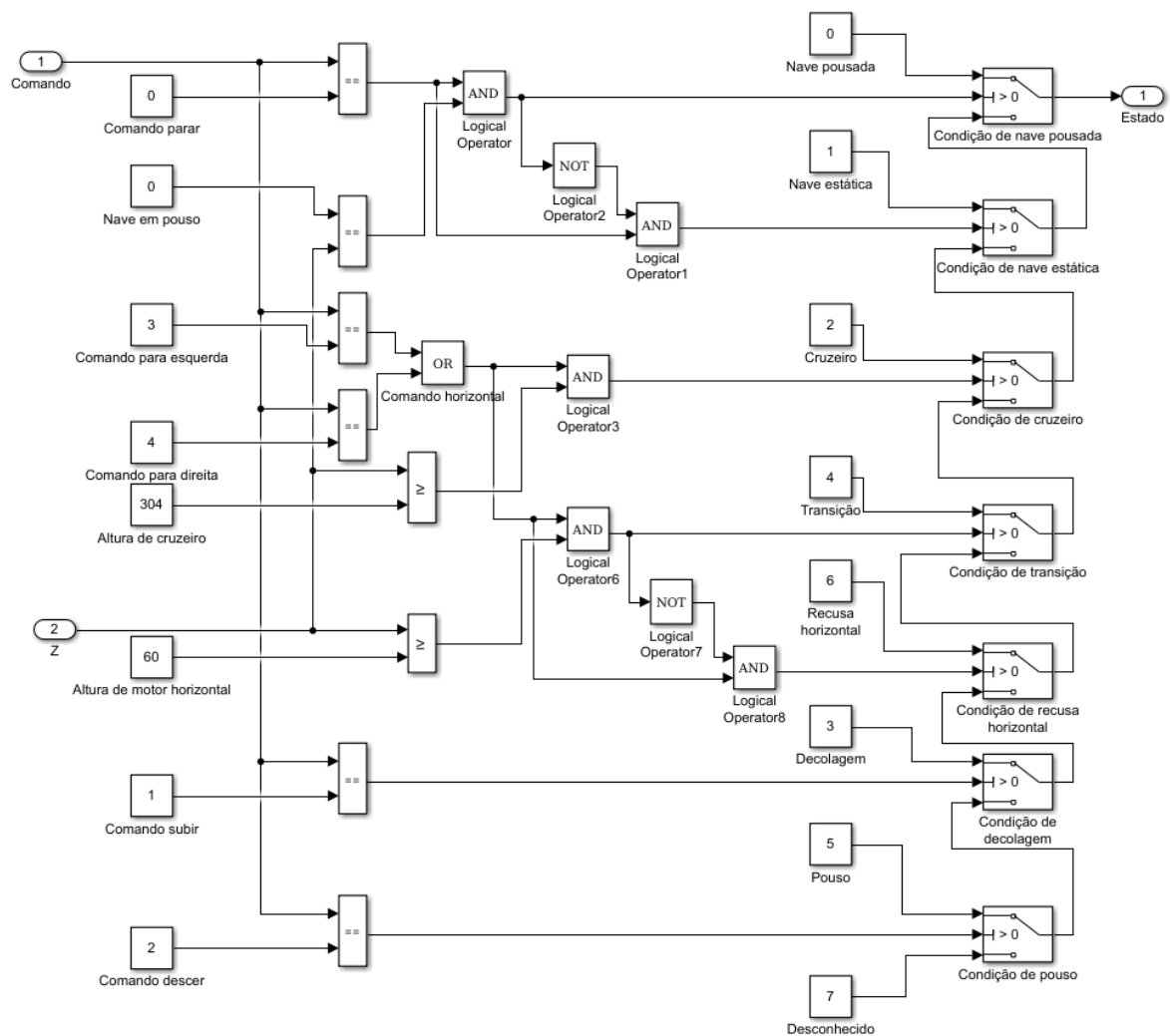


Figura 5.3: Modelo apresentado na Figura 3.23 traduzido na Seção 5.2. Nele, as constantes *Comandoparar*, *Comandoparaesquerda*, *Comandoparadireita*, *Comandosubir* e *Comandodescer* compõem o conjunto das constantes de comando, que são comparadas à entrada *Comando* para determinar qual foi a operação comandada. As constantes *Naveempouso*, *Alturadecruzeiro* e *Alturademotorhorizontal* compõem o conjunto das constantes a serem comparadas com a entrada *Z*, com a altura atual da aeronave. As constantes *Navepousada*, *Naveestática*, *Cruzeiro*, *Transição*, *Recusahorizontal*, *Decolagem*, *Pouso* e *Desconhecido* representam os estados para a saída do sistema. No código traduzido foi criado o módulo *main* manualmente para representar as entradas *comando* e *z*. As constantes de comando foram modeladas nas linhas 18-22 e verificadas nas linhas 59-63. As constantes que representam as alturas de nave pousada, cruzeiro e de ligação dos motores horizontais foram modeladas nas linhas 16, 17, 26 e verificadas nas linhas 57, 58, 67. As operações relacionais que comparam o comando de entrada às constantes associadas foram modeladas nas linhas 34-37, 41 e verificadas nas linhas 51-54, 56. As operações relacionais que comparam a entrada de altura às constantes associadas foram modeladas nas linhas 32, 33, 39. Os operadores lógicos que determinam a escolha da saída da associação dos *switches* foram modelados nas linhas 38, 42-48 e verificados nas linhas 74-81. Os *switches* que determinam o estado da aeronave para a saída são modelados nas linhas 08-14 e instanciam o módulo definido nas linhas 82-89. A saída de estado foi modelada na linha 40.

```

Obstacle_Sensor, BACK_CMD, FWD_CMD, Steady_Lat_CMD);

010. MODULE orientacao_turretta(in_GPS, in_Pilot_Altitude, in_Obstacle_Sensor,
in_BACK_CMD, in_FWD_CMD, in_Steady_Lat_CMD)
011.   VAR
012.     Switch6 : block_switch_simple(in_GPS.z_coordinate,
RisingEdgeDetection.out,Memory2.out);
013.     Switch5 : block_switch_simple(in_Pilot_Altitude,log_LogicalOperator8,
Switch6.out);
014.     Switch4 : block_switch_simple(in_Pilot_Altitude,log_LogicalOperator7,
Switch5.out);
015.     Switch3 : block_switch_simple(in_Pilot_Altitude,log_LogicalOperator6,
Switch4.out);
016.     Switch2 : block_switch_simple(con_Constant4,relop_CompareToConstant2,
Memory1.out);
017.     Switch1 : block_switch_simple(con_Constant2,relop_CompareToConstant1,
Switch.out);
018.     Switch: block_switch_simple(con_Constant3,relop_CompareToConstant,
Switch2.out);
019.     Memory3 : block_boolean_memory(FALSE,in_FWD_CMD);
020.     Memory: block_boolean_memory(FALSE,in_BACK_CMD);
021.     Memory2 : block_memory(0,Switch6.out);
022.     Memory1 : block_memory(0,Switch1.out);
023.     RisingEdgeDetection : block_rising_detection(log_LogicalOperator9);
024.   DEFINE
025.     relop_CompareToConstant5 := in_Obstacle_Sensor.Obs_Above = 1;
026.     relop_CompareToConstant4 := in_Obstacle_Sensor.Obs_Below = 0;
027.     relop_RelationalOperator3 := in_Pilot_Altitude > in_GPS.z_coordinate;
028.     relop_CompareToConstant7 := in_Obstacle_Sensor.Obs_Above = 0;
029.     relop_CompareToConstant6 := in_Obstacle_Sensor.Obs_Below = 0;
030.     relop_CompareToConstant8 := in_Obstacle_Sensor.Obs_Below = 1;
031.     relop_RelationalOperator2 := in_Pilot_Altitude < in_GPS.z_coordinate;
032.     relop_CompareToConstant := Memory3.out = 1;
033.     relop_CompareToConstant1 := in_Steady_Lat_CMD = 1;
034.     relop_CompareToConstant3 := in_Obstacle_Sensor.Obs_Above = 0;
035.     relop_CompareToConstant2 := Memory.out = 1;
036.     log_LogicalOperator9 := log_LogicalOperator10 & log_LogicalOperator11 &
log_LogicalOperator12;
037.     log_LogicalOperator7 := relop_CompareToConstant5 & relop_CompareToConstant6 &

```

```

relop_RelationalOperator2;
038.   log_LogicalOperator8 := relop_CompareToConstant7 & relop_CompareToConstant8 &
relop_RelationalOperator3;
039.   log_LogicalOperator6 := relop_CompareToConstant3 & relop_CompareToConstant4;
040.   con_Alturadecruzeiro := 304;
041.   con_Constant2 := 0;
042.   con_Constant3 := -0.01;
043.   out_Pitch_Target := Switch1.out;
044.   out_Target_Altitude := Switch3.out;
045.   con_Constant4 := 0.01;
046.   log_LogicalOperator12 := !log_LogicalOperator6;
047.   LTLSPEC G(relop_RelationalOperator3 = (in_Pilot_Altitude > in_GPS.z_coordinate))
048.   LTLSPEC G(relop_RelationalOperator2 = (in_Pilot_Altitude < in_GPS.z_coordinate))
049.   LTLSPEC G(con_Alturadecruzeiro = 304)
050.   LTLSPEC G(con_Constant2 = 0)
051.   LTLSPEC G(con_Constant3 = -0.01)
052.   LTLSPEC G(con_Constant4 = 0.01)
053.   LTLSPEC G(log_LogicalOperator9 = (log_LogicalOperator10 & log_LogicalOperator11))
054.   LTLSPEC G(log_LogicalOperator7 = (relop_CompareToConstant5 &
relop_CompareToConstant6 & relop_RelationalOperator2))
055.   LTLSPEC G(log_LogicalOperator8 = (relop_CompareToConstant7 &
relop_CompareToConstant8 & relop_RelationalOperator3))
056.   LTLSPEC G(log_LogicalOperator6 = (relop_CompareToConstant3 &
relop_CompareToConstant4))
057.   LTLSPEC G(log_LogicalOperator12 = (!log_LogicalOperator6))
058.   LTLSPEC G(relop_CompareToConstant5 = (in_Obstacle_Sensor.Obs_Above = 1))
059.   LTLSPEC G(relop_CompareToConstant4 = (in_Obstacle_Sensor.Obs_Below = 0))
060.   LTLSPEC G(relop_CompareToConstant7 = (in_Obstacle_Sensor.Obs_Above = 0))
061.   LTLSPEC G(relop_CompareToConstant6 = (in_Obstacle_Sensor.Obs_Below = 0))
062.   LTLSPEC G(relop_CompareToConstant8 = (in_Obstacle_Sensor.Obs_Below = 1))
063.   LTLSPEC G(relop_CompareToConstant3 = (in_Obstacle_Sensor.Obs_Above = 0))
064.   LTLSPEC G(relop_CompareToConstant1 = (in_Steady_Lat_CMD = 1))
065.   LTLSPEC G(relop_CompareToConstant2 = (Memory.out = 1))
066.   LTLSPEC G(relop_CompareToConstant = (Memory3.out = 1))

067. MODULE block_BusSelector
068.   VAR
069.     x_coordinate : integer;
070.     y_coordinate : integer;

```

```
071.     z_coordinate : integer;

072. MODULE block_BusSelector1
073.   VAR
074.     Obs_Above : {0,1};
075.     Obs_Below : {0,1};
076.     Obs_Left  : {0,1};
077.     Obs_Right : {0,1};

078. MODULE block_switch_simple(in_true, criteria, in_false)
079.   DEFINE
080.     out := case
081.       criteria : in_true;
082.       !criteria : in_false;
083.       TRUE: in_false;
084.     esac;
085.   LTLSPEC G(criteria&(out=in_true)|!criteria&(out=in_false))

086. MODULE block_boolean_memory(in_init, in_data)
087.   VAR
088.     out : boolean;
089.   ASSIGN
090.     init(out) := in_init;
091.     next(out) := in_data;
092.   LTLSPEC (out = in_init)
093.   LTLSPEC G(in_data = next(out))

094. MODULE block_memory(in_init, in_data)
095.   VAR
096.     out : integer;
097.   ASSIGN
098.     init(out) := in_init;
099.     next(out) := in_data;
100.   LTLSPEC (out = in_init)
101.   LTLSPEC G(in_data = next(out))

102. MODULE block_rising_detection(in_data)
103.   VAR
104.     out : boolean;
```

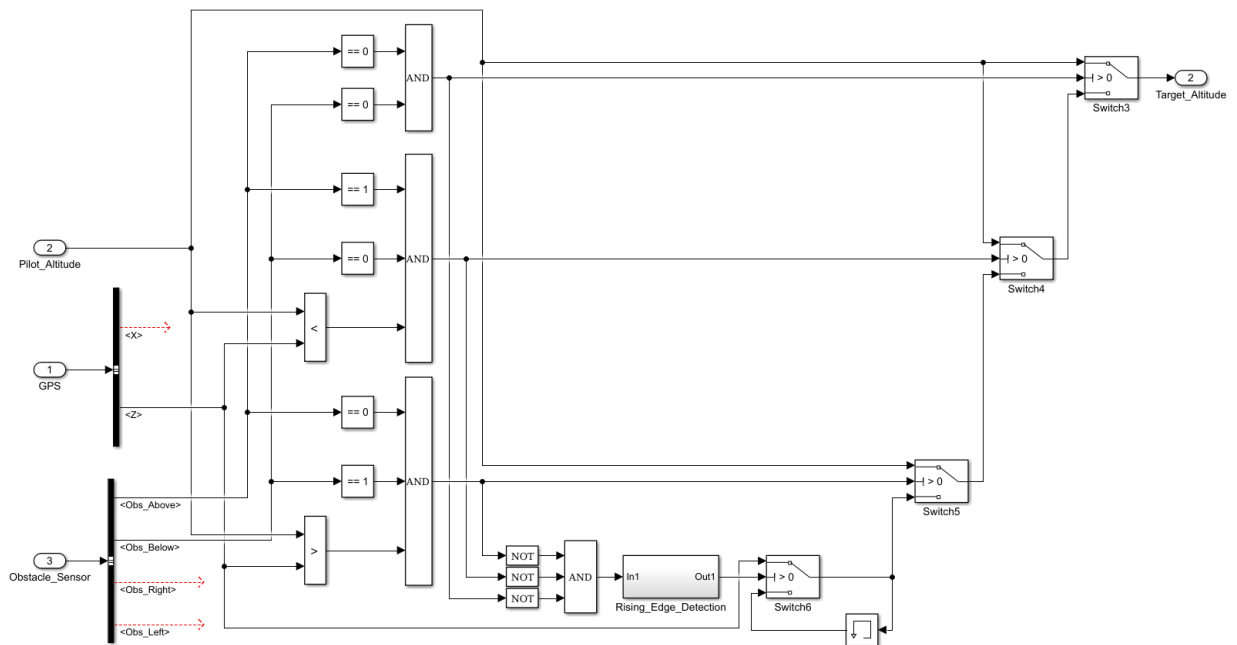



Figura 5.4: Modelo apresentado na Figura 3.25 traduzido na Seção 5.3. No código traduzido foi criado o módulo *main* manualmente para representar as entradas *Pilot_Altitude*, *GPS* e *Obstacle_Sensor*. Os seletores de barramento que decompõem as entradas *GPS* e *Obstacle_Sensor* são modelados nas linhas 003, 005 e instanciam os módulos correspondentes modelados nas linhas 067-077. Os operadores relacionais que comparam a altitude comandada ao valor de GPS são modelados nas linhas 027, 031 e são verificados nas linhas 047, 048. As comparações a constante que recebem os sensores de obstáculo são modeladas nas linhas 025, 026, 028-030, 034 e são verificadas nas linhas 058-063. Os operadores lógicos que determinam a escolha da saída da associação dos *switches* foram modelados nas linhas 036-039, 046 e verificados nas linhas 053-057. O detector de borda de subida foi modelado na linha 023 e instancia o módulo correspondente modelado nas linhas 102-108. Os *switches* que determinam a altitude alvo para a saída são modelados nas linhas 012-018 e instanciam o módulo definido nas linhas 078-085. O bloco de memória é modelado na linha 021 e instancia o módulo correspondente modelado nas linhas 094-101.

```

105.  ASSIGN
106.    init(out) := 0;
107.    next(out) := next(in_data) & !in_data;
108.    LTLSPEC G(next(out) = (next(in_data) & !in_data))

```

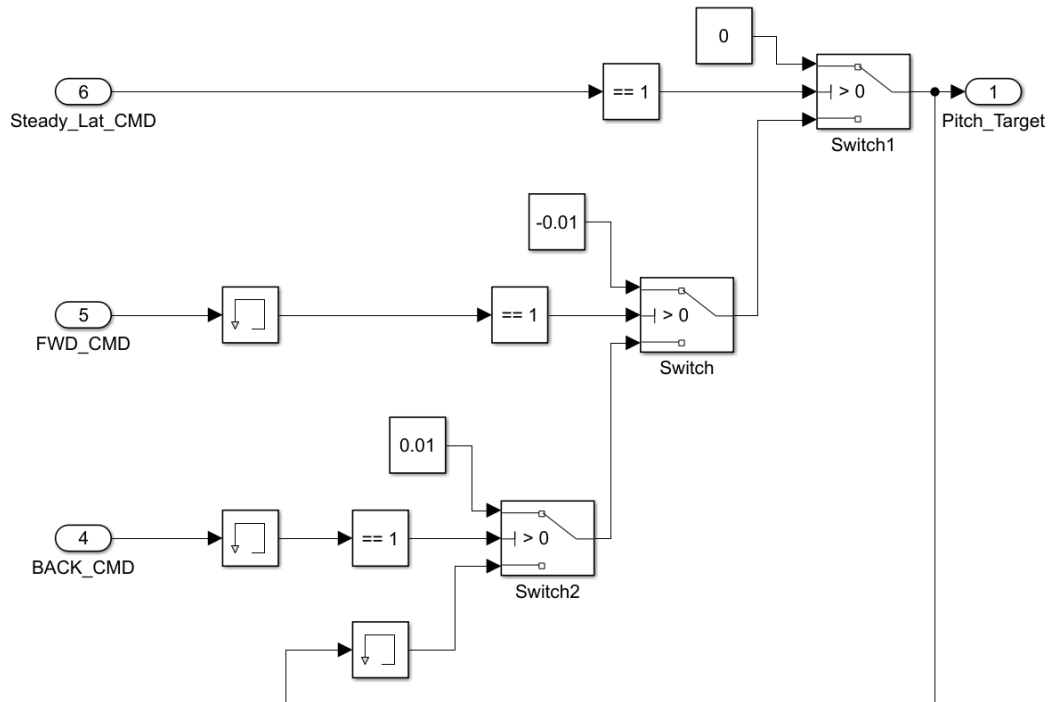


Figura 5.5: Modelo apresentado na Figura 3.26 traduzido na Seção 5.3. No código traduzido foi criado o módulo *main* manualmente para representar as entradas *Steady_Lat_CMD*, *FWD_CMD* e *BACK_CMD*. Os módulos de memória para atraso dos sinais de entrada são modelados nas linhas 019-020 e instanciam os módulos correspondentes modelados nas linhas 086-093. O módulo de memória para a manutenção do ângulo do ciclo anterior é modelado na linha 022 e instancia o módulo correspondente modelado nas linhas 094-101. As comparações a constante para transformação dos sinais de entrada inteiros com valores 0 ou 1 em sinais verdadeiro-falso são modeladas nas linhas 032-033, 035 e verificadas nas linhas 064-066. As constantes que determinam as variações de ângulo de saída do conjunto de *switches* são modeladas nas linhas 041-042, 045 e verificadas nas linhas 050-052. Os *switches* que determinam a saída *Pitch_Target* são modelados nas linhas 016-018 e instanciam o módulo modelado nas linhas 078-085.

5.4 Exemplo de validação de leis operacionais

Para exemplificar a verificação de leis operacionais tomaremos o modelo traduzido na Seção 5.2 para verificar três leis operacionais (1-3) que foram consideradas durante a fase de projeto e uma quarta lei operacional (4) que foi incluída após a sua conclusão:

1. Não deve ser possível iniciar um voo de cruzeiro abaixo da altura operacional (304 unidades, definida na especificação do projeto), gerando acidentes relacionados a colisões, por exemplo;
2. Não devem haver situações em que se recuse o deslocamento horizontal (desloca-

mento em relação à superfície) estando acima da altura de cruzeiro operacional, indicando um bloqueio inútil (este modelo não considera obstáculos);

3. O estado de erro (7 = Desconhecido) não deve ser alcançável pelo controlador, indicando comportamentos não previstos pelos projetistas;
4. Não deve ser possível transitar de um estado de decolagem para um estado de cruzeiro sem passar pelo estado de transição.

As propriedades LTLSPEC correspondentes às leis operacionais definidas são:

1. LTLSPEC $G((instancia_calculo_estado_orientacao_cartesiana.out_Estado = 2) \rightarrow (z \geq 304))$

— A variável de estado estar em Cruzeiro implica na altura em relação ao solo ser maior ou igual 304 unidades.

2. LTLSPEC $G((instancia_calculo_estado_orientacao_cartesiana.out_Estado = 6) \rightarrow (z \leq 304))$

— A variável de estado estar em Recusa implica na altura em relação ao solo ser menor ou igual 304 unidades.

3. LTLSPEC $G(instancia_calculo_estado_orientacao_cartesiana.out_Estado \neq 7)$

— A variável de estado não assume o valor Desconhecido.

4. LTLSPEC $G((instancia_calculo_estado_orientacao_cartesiana.out_Estado = 3) \rightarrow (next(instancia_calculo_estado_orientacao_cartesiana.out_Estado) \neq 2))$

— A variável de estado estar em condição de Decolagem implica em um estado seguinte diferente de Cruzeiro.

Para as leis operacionais 1-3 não foram encontrados contraexemplos, provando que o projeto cumpre tais leis. Para a lei operacional 4, entretanto, foi encontrado um contraexemplo onde o estado comando = 1 (subir), $z = 1$, estado = 3 (decolagem) transita para o estado comando = 4 (avanço à direita), $z = 304$, estado = 2 (cruzeiro). Esse contraexemplo indica que, caso o piloto não gere nenhum comando após a ordem de decolagem até que a aeronave alcance a altura de cruzeiro e imediatamente ordene o deslocamento à direita, a aeronave admitirá o estado de cruzeiro. Dessa forma, concluímos que o projeto cumpre as leis operacionais originais 1-3, mas não a lei adicionada posteriormente (4). A geração do contraexemplo evidencia uma condição de limiar em que a violação ocorre, sendo útil para nortear a sua correção, que após realizada, pode ser verificada utilizando a mesma propriedade.

```
Trace Description: MSAT BMC counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  comando = 1
  z = 1
  instancia_calculo_estado_orientacao_cartesiana.out_Estado = 3
```

Figura 5.6: Fragmento do estado inicial do contraexemplo de saída que contém as variáveis que geram a violação da Lei Operacional 4 pelo modelo traduzido na Seção 5.2.

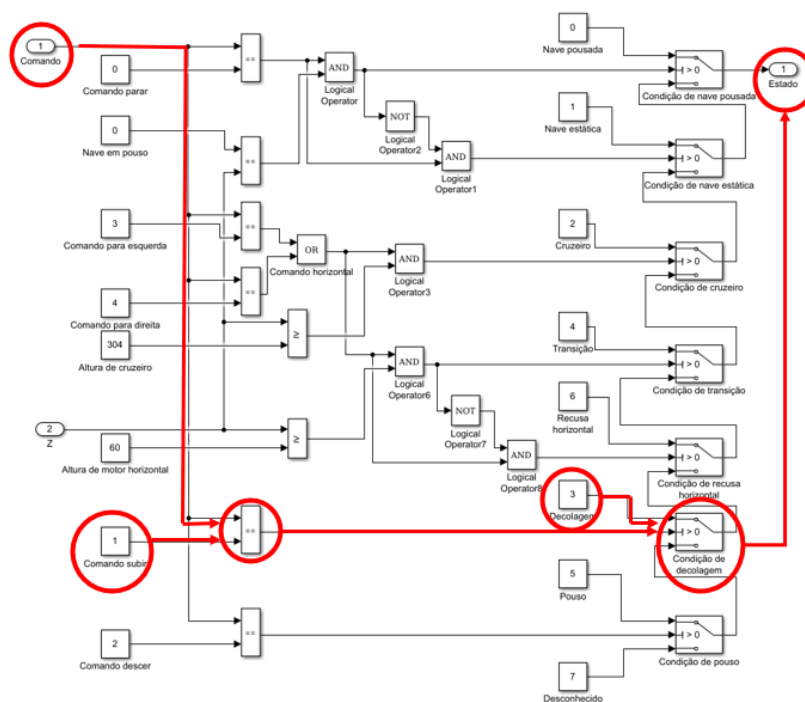


Figura 5.7: Fluxo de dados correspondente ao estado do contraexemplo apresentado na Figura 5.6, onde a entrada *Comando* com valor igual ao da constante *Comandosubir* habilita a constante *Decolagem* na saída *Estado* do sistema.

```
-> State: 1.2 <-
  comando = 4
  z = 304
  instancia_calculo_estado_orientacao_cartesiana.out_Estado = 2
```

Figura 5.8: Fragmento do estado de erro do contraexemplo de saída que contém as variáveis que geram a violação da Lei Operacional 4 pelo modelo traduzido na Seção 5.2.

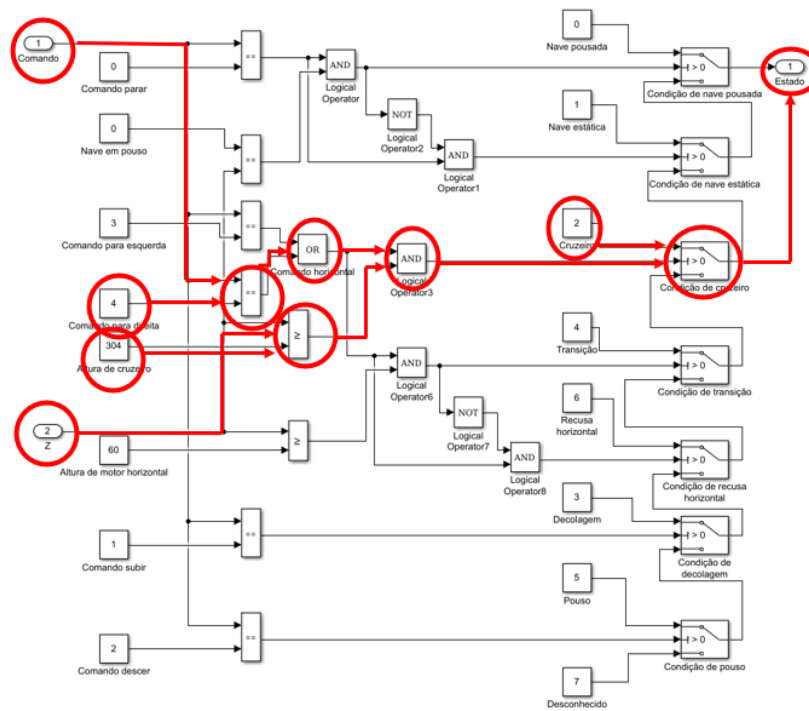


Figura 5.9: Fluxo de dados correspondente ao estado do contraexemplo apresentado na Figura 5.8, onde a entrada *Comando* com valor igual ao da constante *Comandoparadireita* e a entrada *Z* com valor igual a 340 habilitam a constante *Cruzeiro* na saída *Estado* do sistema.

Capítulo 6

Conclusão

A aplicação de veículos aéreos não tripulados para transporte de passageiros é uma área atual e com uma característica comercial relevante. Barreiras comerciais sobre a implantação de veículos aéreos não tripulados podem ser reduzidas ou resolvidas através da verificação formal dos modelos desses veículos, pois a redução da probabilidade de acidentes a adéqua a legislações específicas e aumenta a confiança dos clientes. As técnicas atualmente aplicadas como testes e simulações não possuem garantia da corretude dos modelos, aspecto em que a verificação formal possui excelência. Entretanto, ferramentas e técnicas para a verificação formal desses sistemas são escassas, sendo este trabalho contribuição relevante para o problema da verificação de sistemas de aviação.

Realizamos uma análise do problema da verificação de sistemas de aviação, apresentando um estudo das necessidades mercadológicas na implementação de novas tecnologias; técnicas e implementação de modelos que representam sistemas de aviação parciais e completos; a implementação de um conjunto de modelos em Matlab/Simulink para as ferramentas desenvolvidas; um estudo sobre o ambiente de projetos, fluxo de dados e aspectos detalhados da linguagem utilizada pela ferramenta MATLAB/Simulink; um estudo sobre um particionamento funcional de códigos *.mdl* no sentido de se extrair estruturas e funções formalmente verificáveis; um estudo sobre os principais blocos e seus principais parâmetros, entradas e saídas para a verificação de modelos de sistemas de aviação; um esquema de tradução automática de códigos *.mdl* para *.xmv*, de modo que sistemas de aviação possam ser automática e formalmente verificados; O desenvolvimento de uma ferramenta (SMCT) para a tradução automática de *.mdl* para *.xmv*; Uma análise e interpretação de resultados da verificação do um conjunto *benchmark* implementado. Os principais problemas abordados neste trabalho são:

1. O entendimento e padronização de estruturas de projetos de aeronaves autônomas

em linguagens de blocos;

2. A escassez de modelos para benchmark de ferramentas sobre projetos de aeronaves autônomas criados em Matlab/Simulink;
3. A dificuldade em se verificar projetos descritos em Matlab/Simulink utilizando ferramentas de verificação formal devido à interdisciplinaridade e complexidade das áreas envolvidas;
4. A necessidade da garantia do cumprimento de leis operacionais por sistemas de aeronaves autônomas.

Os resultados deste trabalho geram artefatos para se aumentar a confiabilidade de sistemas de aviação através da prova, mesmo que parcial, de sua corretude, de forma a confrontar as barreiras comerciais existentes nos dias de hoje. A prova do funcionamento correto dos sistemas de controle permite a geração de sistemas cada vez mais automáticos, resultando em diminuição de riscos por falha humana e possibilidade de desenvolvimento de um VANT totalmente autônomo com funções provadas corretas. Os modelos gerados apresentam simplificações de sistemas de aeronaves completos de modo que o processo de tradução e verificação pode ser facilmente acompanhado. Ainda sobre tais modelos, observa-se que mesmo pequenos modelos geram códigos *.mdl* muito grandes, de modo que a verificação de sistemas reais e completos é incompatível com a proposta deste texto. O estudo realizado sobre a linguagem *mdl* do Simulink, com sua estrutura, demonstrou ser possível o desenvolvimento de um esquema de tradução mesmo com a dificuldade apresentada por outros trabalhos [Postma, 2015, Nardi, 2017, Wassying, 2018]. O desenvolvimento da ferramenta SMCT abre possibilidades de tradução automática de sistemas cada vez mais complexos mediante a implementação de mais tipos de blocos Simulink, possuindo uma estrutura facilitada para tal (uma classe Java para cada tipo de bloco com uma interface explicitando seu esquema de desenvolvimento). A verificação dos códigos traduzidos demonstra a potencialidade da ferramenta XMV em gerar resultados de prova de corretude e de contraexemplos na avaliação de modelos de sistemas de aeronaves. Os modelos originais convertidos provados corretos através de especificações formais demonstram a capacidade da técnica em verificar módulos que componham modelos maiores e mais complexos. As especificações podem ser extraídas dos manuais de voo de aeronaves reais como margens de segurança para variáveis da aeronave e cenários de operação serem mutuamente excludentes com cada atuador sob controle de um único controlador. De forma sumarizada, as principais contribuições deste trabalho são:

1. O desenvolvimento de uma estrutura de abstração para projetos de aeronaves autônomas em linguagens de blocos dividida em quatro componentes modulares;
2. A implementação de um conjunto de modelos em Matlab/Simulink contendo quatro modelos completos compostos de forma modular onde seus componentes são intercambiáveis gerando até 16 modelos sintéticos;
3. Uma ferramenta automática para a tradução de códigos Simulink *.mdl* em códigos NuXMV *.xmv* com funcionalidade de blocos provada correta;
4. Um método para a verificação do cumprimento de leis operacionais por projetos traduzidos pela ferramenta SMCT.

Conclui-se que o trabalho atingiu seu objetivo em gerar um ferramental inicial para a redução das barreiras comerciais através da verificação formal de sistemas de aeronaves descritos em uma linguagem amplamente utilizada. Trabalhos futuros devem incluir pesquisas sobre a expansão da cobertura em largura da ferramenta desenvolvida; Pesquisas que apliquem a ferramenta SMCT para a verificação de sistemas de outros domínios; Estudos sobre outras ferramentas de projetos de sistemas de aviação utilizados na indústria como entrada para a ferramenta SMCT; Estudos de outras ferramentas de verificação automática que cubram aspectos como a dinâmica física do sistema de aeronaves; Outros que expandam o escopo deste trabalho.

Referências Bibliográficas

- [Air France, 2011] Air France (2011). Air flight 447.
- [Alino, 2016] Alino, E.; Campos, S. M. G. F. M. L. R. (2016). Modelagem e verificação formal do módulo de cálculo da distância, energia e tempo da frenagem de aeronaves. *Simpósio Embraer de Tecnologia e Inovação*.
- [Andrade, 2008] Andrade, F. V. (2008). *Contribuições para o problema de verificação de equivalência combinacional*. Doutorado em ciência da computação, Curso de Pós Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, Belo Horizonte.
- [Biere, 2009] Biere, A. (2009). Bounded model checking. *Handbook of Satisfiability*, pp. 457–481.
- [Biere & Bloem, 2014] Biere, A. & Bloem, R., editores (2014). *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*. Springer.
- [Bozzano, 2020] Bozzano, M. (2020). nuxmv 2.0.0 user manual. *FMK, Itália*.
- [Britannica Academic, 2015] Britannica Academic (2015). unmanned aerial vehicle (uav).
- [Bryant, 1986] Bryant, E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pp. 677–691.
- [Campos, 1996] Campos, S. (1996). *A Quantitative Approach to the Formal Verification of Real-Time Systems*. Tese de doutorado, Carnegie Mellon University, School of Computer Science.
- [Cimatti, 2002] Cimatti, A.; Clarke, E. G. E. G. F. P. M. R. M. S. R. T. A. (2002). Nusmv 2: An opensource tool for symbolic model checking. *International Conference on Computer Aided Verification*.

- [Clarke, 1980] Clarke, E. E. E. (1980). Characterizing correctness properties of parallel programs using fixpoints. *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pp. 169–181.
- [Clarke, 1996] Clarke, K. M. S. C. V. H.-G. E. (1996). Symbolic model checking. *DAR-PA/CMO*.
- [Cousot, 2009] Cousot, P.; Cousot, R. F. J. M. L. M. A. R. X. (2009). Why does astrée scale up? *Formal Methods in System Design*, 35.
- [Dragomir, 2018] Dragomir, V. P. S. T. I. (2018). The refinement calculus of reactive systems toolset. *TACAS 2018. Lecture Notes in Computer Science.*, pp. 201–208.
- [Emerson, 1982] Emerson, E. C. E. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131:52–71.
- [Ferreira, 2016] Ferreira, B. (2016). Verification of vehicular networks using probabilistic model checking.
- [Friedman, 2014] Friedman, J. L. J. (2014). Requirements modeling and automated requirements-based test generation. *SAE International*.
- [Gleick, 1996] Gleick, J. (1996). A bug and a crash.
- [Guimarães, 2017] Guimarães, B.; Mendonça, G. A. P. P. M. A. G. P. F. (2017). Dawncc: a source-to-source automatic parallelizer of cand c++ programs. *ACM Transactions on Architecture and Code Optimization*.
- [Guimarães, 2020] Guimarães, B.; Mendonça, G. P. F. (2020). Dawncc.
- [Güçlü, 2016] Güçlü, A.; Arıkan, K. K. D. (2016). Attitude and altitude stabilization of a fixedwing vtol unmanned air vehicle. *AIAA Modeling and Simulation Technologies Conference*.
- [Jeff Holden, 2016] Jeff Holden, N. G. (2016). Fast-forwarding to a future of on-demand urban air transportation.
- [Konig, 2018a] König, L. (2018a). Reunião de trabalho - um modelo asa-fixa.
- [Konig, 2018b] König, L. (2018b). Reunião de trabalho - um modelo híbrido vtol/asa-fixa.
- [Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107--115.

- [Lions, 1996] Lions, J. L. (1996). Ariane 5 flight 501 failure.
- [Loveland, 1962] Loveland, M. D. G. L. D. (1962). A machine program for theorem proving. *Communications of the Association for Computer Machinery*, pp. 394–397.
- [Mcmillan, 2000] Mcmillan, K. L. (2000). The smv system.
- [Mcmillan, 1992] Mcmillan, L. (1992). The smv system.
- [Melo, 2018] Melo, L.; Ribeiro, R. A. M. P. F. (2018). Inference of static semantics for incomplete c programs. *Proceedings of the ACM on Programming Language*.
- [Melo, 2020] Melo, L.; Ribeiro, R. A. M. P. F. (2020). Psyche-c.
- [Miller, 2003] Miller, S.; Tribble, A. C. T. D. E. (2003). Flight guidance system requirement specification. *NASA Contractor Report CR-2003-212426*.
- [Miller,] Miller, D. C. S. Formal methods case studies for do-333. *NASA/CR-2014-218244*.
- [Miller, 2014] Miller, D. C. S. (2014). Do-333 certification case studies. *FAA National Systems, Software and Airborne Electronic Hardware Conference*.
- [Morzenti, 1991] Morzenti, M. F. A. (1991). Real-time system validation by model checking in trio. *IEEE*, pp. 20–28.
- [Morzenti, 2003] Morzenti, A.; Pradella, M. S. P. S. P. (2003). Model-checking trio specifications in spin. *FME 2003: Formal Methods*.
- [Nardi, 2017] Nardi, L. B. M. D. O. (2017). Test oracles for simulink-like models. *Autom Soft Eng*, pp. 369–391.
- [Newrailwaymodellers.co.uk, 2020] Newrailwaymodellers.co.uk (2020). Railway crossing.
- [Ozsoy, 2017] Ozsoy, H. E. A. K. C. (2017). Model predictive control of an unmanned aerial vehicle. *Aircraft Engineering and Aerospace Technology*.
- [Postma, 2015] Postma, M. B. K. L. M. L. V. P. A. K. J. O. B. M. M. B. S. (2015). Signature required: Making simulink data flow and interfaces explicit. *Science of Computer Programming*, pp. 29–50.
- [Putnam, 1960] Putnam, M. D. H. (1960). A computation procedure for quantification theory. *Journal of the Association for Computer Machinery*, pp. 201–205.

- [Riddick, 2018] Riddick, K. C. D. E. C. D. G. M. S. E. (2018). A piloted evaluation of damage accommodating flight control using a remotely piloted vehicle. *American Institute of Aeronautics and Astronautics*.
- [Rodrigues, 2016] Rodrigues, B; Pereira, F. A. D. (2016). Sparse representation of implicit flows with applications to side-channel detection. *Universidade Federal de Minas Gerais*.
- [Rodrigues, 2020] Rodrigues, B; Pereira, F. A. D. (2020). Flowtracker.
- [SC-205, RTCA, Inc, 2011] SC-205, RTCA, Inc (2011). Rtc do-178c, software considerations in airborne systems and equipment certification.
- [SC-205, RTCA, Inc, 2015] SC-205, RTCA, Inc (2015). Rtc do-333, formal methods supplement to do-178c and do-278-a.
- [Sifakis, 1982] Sifakis, J. Q. J. (1982). A temporal logic to deal with fairness in transition systems. *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 217–225.
- [Spinroot.com, 2020] Spinroot.com (2020). What is spin.
- [Subcommittee on Investigations and Oversight, 1992] Subcommittee on Investigations and Oversight, C. (1992). Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia : Report to the chairman.
- [The Mathworks, 2014] The Mathworks, I. (2014). Missile guidance system example.
- [The Mathworks, 2020] The Mathworks, I. (2020). Model references.
- [Turetta, 2018] Turetta, F. (2018). Reunião de trabalho - um modelo vant com três graus de liberdade.
- [Vilhena, 2018] Vilhena, A. (2018). Modelagem e verificação de controle utilizando métodos formais. Dissertação de mestrado, Departamento de Engenharia Elétrica da Universidade Federal de Minas Gerais.
- [Wassyng, 2018] Wassyng, V. P. S. P. M. L. M. J. B. M. A. K. M. B. J. O. G. M. A. (2018). Software engineering practices and simulink: bridging the gap. *Int J Soft Tools Technol Transfer*, pp. 95–117.

Apêndice A

Detalhamento estrutural dos blocos Simulink traduzidos automaticamente pela SMCT


Este apêndice trata dos blocos Simulink abordados na Seção 4.3 de forma estendida, trazendo a totalidade de seus parâmetros em um detalhamento estrutural. Para cada bloco ela traz seu Nome seguido do Nome do bloco encontrado no Simulink; Símbolo esquemático; Biblioteca (com fim de evitar ambiguidades no tratamento dos blocos); Descrição da função do bloco no Simulink; descrição de suas Portas de entrada e saída (com ordenamento respeitado); Parâmetros utilizados no processo de tradução, com uma descrição, seu lexema em *Parâmetro do bloco* e outros detalhes utilizados na implementação; Parâmetros não tratados por pelo menos um dos seguintes motivos: Não possuir valor funcional, mas estético ou não pertencer ao escopo do trabalho ou possuir função relativa às operações de simulação (não influenciando funcionalmente no modelo); Implementação XMV do bloco; Propriedade para validação da implementação realizada. A apresentação dos blocos nesse formato objetiva mais clara compreensão das decisões e estratégias, bem como metodologia de tradução.

A.1 Seletor de Barramentos (Bus Selector)

O Seletor de Barramentos, disponível na biblioteca **Simulink / Commonly Used Blocks**, seleciona sinais de um barramento de entrada. O Seletor de Barramentos no Simulink recebe como entrada um barramento que condensa diversos sinais e decompõe os sinais em um conjunto de saídas. Ele é entendido como um vetor em sua entrada que pos-

sui cada uma de suas posições atribuídas a uma variável de saída. Mais propriamente, o Seletor de Barramentos tem como saída um subconjunto específico de elementos do barramento em sua entrada. O bloco pode ter como saída um conjunto de sinais separados (como foi modelado para esta ferramenta) ou um novo barramento. A ordenação dos sinais em um Seletor de Barramentos é sempre mantida da entrada para a saída.

Detalhamento estrutural

| Seletor de Barramentos (Bus Selector) | |
|--|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Barramento de entrada Valores reais ou complexos de qualquer tipo de dados suportado pelo Simulink, com a exceção de arranjos de barramentos. |
| Saída 1 | Elementos selecionados do barramento de entrada Para cada sinal de saída este bloco utiliza uma porta de saída distinta. |
| Parâmetros | |
| Sinais no barramento | Lista de elementos de sinal do barramento de entrada a serem selecionados para a saída. Parâmetro do bloco: <i>InputSignals</i> Tipo: matriz Valores: nomes dos sinais Padrão: '[]' |
| Sinais selecionados | Elementos selecionados do barramento de entrada. Parâmetro do bloco: <i>OutputSignals</i> Tipo: vetor de caracteres Valores: vetor de caracteres no formato: ' <i>signal1,signal2</i> ' Padrão: nenhum |

| | |
|---|--|
| Saída como um barramento virtual | <p>Determina se a saída deve ser vista como um barramento de elementos selecionados.</p> <p>Por padrão a saída do bloco possui um sinal único de saída para cada elemento do barramento de entrada. Se este parâmetro possuir o valor <i>'on'</i>, as saídas serão organizadas em um barramento virtual.</p> <p>Parâmetro do bloco: <i>OutputAsBus</i></p> <p>Tipo: vetor de caracteres</p> <p>Valores: <i>'on'</i> <i>'off'</i></p> <p>Padrão: <i>'off'</i></p> |
| Parâmetros não tratados | |
| <i>Filter by name</i> | Filtra o conjunto de entradas exibidas por um termo de busca |
| <i>Enable regular expression</i> | Filtra o conjunto de entradas exibidas por uma expressão regular |
| <i>Show filtered results as a flat list</i> | Modifica a aparência da lista filtrada |

Implementação XMV

Declaração

nome_bloco: *block_nome_bloco(barramento_entrada)*

Especificação do módulo

MODULE *block_nome_bloco(barramento_entrada)*

ASSIGN

[PARA CADA SINAL NA ENTRADA] *nome_sinal* :=
barramento_entrada[posicao_sinal]

Propriedade para validação

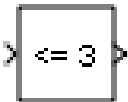
G(*nome_sinal=barramento_entrada[posicao_sinal]*)

A.2 Comparação a constante (Compare to Constant)

O bloco de Comparação a Constante, disponível na biblioteca **Simulink / Logic and Bit Operations** determina como um sinal de entrada se compara a uma constante especi-

ficada. O operador de comparação é parametrizável entre os relacionais básicos (igualdade, maior, menor, igual ou maior, igual e menor, diferença). A saída do bloco é o correspondente binário 1 se a comparação foi verdadeira e 0 se a comparação for falsa.

Detalhamento estrutural

| Comparação a constante (Compare to Constant) | |
|---|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Logic and Bit Operations |
| Portas | |
| Entrada 1 | Sinal de entrada, especificado como escalar a ser comparado com o parâmetro Valor Constante através da operação especificada. |
| Saída 1 | Sinal de saída, valor 0 se a comparação for falsa e 1 se a comparação for verdadeira. |
| Parâmetros | |
| Operador | Operador lógico para a comparação. Parâmetro do bloco: <i>relop</i> Tipo: vetor de caracteres Valores: '=' '<' '<=' '>=' '>' Padrão: '<=' |
| Constante a ser comparada | Especifica o valor constante a ser comparado com a entrada. Parâmetro do bloco: <i>const</i> Tipo: Vetor de caracteres Valores: escalar vetor matriz arranjo N-D Padrão: '3.0' |
| Parâmetros não tratados | |
| <i>Output data type</i> | Especifica o tipo de dado da saída, booleano ou <i>uint8</i> |
| <i>Enable zero-crossing detection</i> | Habilita a detecção de <i>zero-crossing</i> . |

Implementação XMV

Declaração

relop_nome_bloco := *senal_entrada operador constante*

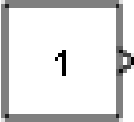
Propriedade para validação

$G(\text{relop_nome_bloco} = (\text{sinal_entrada operador constante}))$

A.3 Constante (Constant)

O bloco Constante, disponível na biblioteca **Simulink / Commonly Used Blocks**, gera um sinal de valor constante real ou complexo como escalar, vetor ou matriz. O valor constante do sinal é definido através de um parâmetro do bloco.

Detalhamento estrutural

| Constante (Constant) | |
|---|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Saída 1 | Valor constante especificado no parâmetro correspondente. |
| Parâmetros | |
| Valor constante | Especifica o valor constante de saída do bloco. Parâmetro do bloco: <i>Value</i> Tipo: vetor de caracteres Valores: escalar vetor matriz arranjo N-D Padrão: '1' |
| Parâmetros não tratados | |
| <i>Interpret vector parameters as 1-D</i> | Especifica a saída do bloco como um vetor de tamanho N se o parâmetro de Valor constante for avaliado como um vetor linha ou coluna de N elementos |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco devido a mudanças no parâmetro de Valor constante. |
| <i>Output minimum</i> | Especifica o valor mínimo atribuível à saída pelo Simulink. |
| <i>Output maximum</i> | Especifica o valor máximo atribuível à saída pelo Simulink. |
| <i>Output data type</i> | Especifica o tipo de dados da saída do bloco. |

*Lock output data type
setting against changes by
the fixed-point tools*

Previne que ferramentas que lidem com aritmetica de ponto fixo sobreponham o tipo especificado do bloco.

Implementação XMV

Declaração

con_nome_bloco : valor_constante

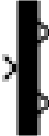
Propriedade para validação

G(con_nome_bloco = valor_constante)

A.4 Demultiplexador (Demux)

O bloco Demultiplexador, disponível na biblioteca **Simulink / Commonly Used Blocks**, extrai os componentes de um vetor de sinais de entrada, separando os seus componentes como saídas.

Detalhamento estrutural

| Demultiplexador (Demux) | |
|--------------------------------|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Vetor de entrada a ser decomposto em saídas pelo bloco. |
| Saídas | Sinais de saída extraídos do vetor de entrada. Os sinais de saída são ordenados de cima para baixo. |
| Parâmetros | |
| Número de saídas | Especifica o número de saídas e as suas dimensões. Parâmetro do bloco: <i>Outputs</i> Tipo: escalar ou vetor Valores: vetor de caracteres Padrão: '1' ou vetor |

| Parâmetros não tratados | |
|-------------------------|---|
| <i>Display option</i> | Especifica o modo de exibição gráfico do bloco demultiplexador. |

Implementação XMV

Declaração

nome_bloco: *block_nome_bloco(barramento_entrada)*

Especificação do módulo

MODULE *block_nome_bloco(barramento_entrada)*

ASSIGN

[PARA CADA SINAL NA ENTRADA] *nome_sinal* :=
barramento_entrada[posicao_sinal]


Propriedade para validação

G(nome_sinal=barramento_entrada[posicao_sinal])

A.5 Ganho (Gain)

O bloco Ganho, disponível na biblioteca **Simulink / Commonly Used Blocks**, multiplica a entrada por um valor constante. A entrada pode ser um escalar, vetor ou matriz. O fator de multiplicação é informado como um parâmetro.

Detalhamento estrutural

| Ganho (Gain) | |
|---------------------|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Valor real ou complexo como escalar, vetor ou matriz a ser multiplicado pelo ganho. |
| Saída 1 | O valor do sinal de entrada multiplicado pelo valor constante de ganho. |

| Parâmetros | |
|-------------------------|--|
| Ganho | <p>Especifica o valor pelo qual multiplicar a entrada. Pode ser um valor real ou complexo como escalar, vetor ou matriz.</p> <p>Parâmetro do bloco: <i>Gain</i></p> <p>Tipo: Vetor de caracteres</p> <p>Valores: '1' valor real ou complexo como escalar, vetor ou matriz</p> <p>Padrão: '1'</p> |
| Parâmetros não tratados | |
| <i>Multiplication</i> | Especifica o modo de multiplicação a ser realizado para entrada ou ganho como matriz. |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |

Implementação XMV

Declaração

$gain_nome_bloco : entrada * ganho$


Propriedade para validação

$G(gain_nome_bloco = (entrada * ganho))$

A.6 Entrada (Inport)

O bloco de Entrada, disponível na biblioteca **Commonly Used Blocks**, liga sinais de fora de um sistema ao sistema que o contém, hierarquicamente inferior. Apesar de possuir apenas a função de ligar dois pontos é um elemento estrutural na parametrização de funções entre os blocos.

Detalhamento estrutural

| Entrada (Inport) | |
|---------------------|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |

| Saída 1 | Valor do sinal de entrada no sistema. |
|---|---|
| Parâmetros | |
| Número da porta | Especifica a posição única da porta no bloco que a contém. |
| | Parâmetro do bloco: <i>Port</i> |
| | Tipo: Vetor de caracteres |
| | Valores: inteiro |
| Parâmetros não tratados | |
| <i>Icon display</i> | Especifica a informação exibida no ícone do bloco. |
| <i>Latch input by delaying outside signal</i> | Especifica a saída do bloco como o atraso em um passo de tempo da entrada. |
| <i>Latch input for feedback signals of function-call subsystem outputs</i> | Especifica a saída do bloco como um atraso dependente de um gatilho dado por uma chamada de função. |
| <i>Interpolate data</i> | Especifica se o bloco deve interpolar linearmente os valores da saída em passos de tempo em que a entrada seja ausente. |
| <i>Connect Input</i> | Ativa a ferramenta <i>Root Inport Mapper</i> para a importação, visualização e mapeamento de dados de sinal e barramento para entradas em nível raiz. |
| <i>Output function call</i> | Especifica que o sinal de entrada é a saída de um sinal de gatilho de chamada de função. |
| <i>Minimum</i> | Especifica o valor mínimo atribuível à saída pelo Simulink. |
| <i>Maximum</i> | Especifica o valor máximo atribuível à saída pelo Simulink. |
| <i>Data type</i> | Especifica o tipo do sinal dado como entrada. |
| <i>Lock output data type setting against changes by the fixed-point tools</i> | Impede a mudança de tipo da entrada devido a operações de ponto fixo. |
| <i>Output as nonvirtual bus</i> | Especifica se a entrada fornecida é virtual ou não. |
| <i>Unit</i> | Especifica a unidade física do sinal de entrada do bloco. |
| <i>Port dimensions (-1 for inherited)</i> | Especifica a dimensão do sinal de saída do bloco. |
| <i>Variable-size signal</i> | Especifica os tipos de sinal assumíveis pelo bloco. |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |
| <i>Signal type</i> | Especifica tipo numérico da saída do bloco. |

Implementação XMV

Declaração (Normalmente interpretado como parâmetro de um módulo)

$in_nome_bloco := entrada$

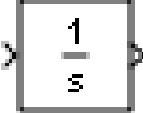
Propriedade para validação

$G(in_nome_bloco=entrada)$

A.7 Integrador pelo método de somas discretas (Integrator)

O bloco Integrador, disponível na biblioteca **Simulink / Commonly Used Blocks**, gera como saída o valor da integral de sua entrada em relação ao tempo. A saída é calculada através da soma discreta da variável de integração (integrando) a cada unidade de tempo.

Detalhamento estrutural

| Integrador pelo método de somas discretas (Integrator) | |
|--|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Valor do sinal a ser integrado. |
| Entrada 2 | Reinicia o estado do integrador às condições iniciais. |
| Entrada 3 | Condição inicial. |
| Saída 1 | O estado integrado da entrada. |
| Saída 2 | Saturação da saída. Se o valor da Saída 1 estiver acima do limite de saturação superior, gera o valor 1. Se estiver abaixo do limite de saturação inferior, gera o valor -1. Gera 0 caso esteja entre os limites. |

| | |
|--|---|
| Saída 3 | O estado integrado da entrada, com manutenção do sinal por um passo de tempo em caso de reinicialização do integrador. |
| Parâmetros | |
| Condição inicial | Especifica o estado inicial do bloco integrador. Parâmetro do bloco: <i>InitialCondition</i> Tipo: Vetor ou escalar Padrão: '0' |
| Parâmetros não tratados | |
| <i>External reset</i> | Especifica o tipo do gatilho a ser usado para o sinal externo de reinicialização. |
| <i>Initial condition source</i> | Especifica se a fonte do estado inicial é interna (dada por parâmetro) ou externa (dada por uma porta de entrada). |
| <i>Limit output</i> | Habilita a limitação do valor de saída do bloco entre os valores de saturação inferior e superior. |
| <i>Upper saturation limit</i> | Especifica o limite superior de saturação da saída do bloco. |
| <i>Lower saturation limit</i> | Especifica o limite inferior de saturação da saída do bloco. |
| <i>Wrap state</i> | Habilita o encapsulamento de estados entre os valores superior e inferior de encapsulamento para integrais de fenômenos cíclicos, periódicos ou rotacionas em natureza. |
| <i>Wrapped state upper value</i> | Especifica o limite superior de encapsulamento da saída do bloco. |
| <i>Wrapped state lower value</i> | Especifica o limite inferior de encapsulamento da saída do bloco. |
| <i>Show saturation port</i> | Habilita a porta de saturação na saída do bloco. |
| <i>Show state port</i> | Habilita a porta de estado na saída do bloco. |
| <i>Absolute tolerance</i> | Determina constantes para tolerância absoluta na computação de estados do bloco. |
| <i>Ignore limit and reset when linearizing</i> | Trata o bloco como não reinicializável e sem limites na saída para fins de linearização. |
| <i>Enable zero-crossing detection</i> | Habilita a detecção de discontinuidades que geram passos de tempo excessivamente pequenos em sua computação. |
| <i>State Name</i> | Assinala nomes únicos para cada estado do integrador. |

Implementação XMV

Declaração

`init(int_nome_bloco) : condicao_inicial`

$next(int_nome_bloco) : int_nome_bloco + integrando$

Propriedade para validação

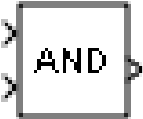
$int_nome_bloco = condicao_inicial$

$G(next(int_nome_bloco) = int_nome_bloco + integrando)$

A.8 Lógico (Logical Operator)

O bloco Lógico, disponível na biblioteca **Commonly Used Blocks** realiza a operação lógica especificada entre suas entradas. Uma entrada é considerada verdadeira se diferente de zero e falsa se igual a zero. A saída é gerada a partir da realização da operação lógica, definida em um parâmetro, entre as entradas, gerando a saída com valor lógico binário correspondente.

Detalhamento estrutural

| Lógico (Logical Operator) | |
|----------------------------------|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada N | N-ésimo sinal de entrada especificado como escalar, vetor ou matriz. |
| Saída 1 | O sinal de saída constituído em 0's e 1's, com mesma dimensão da entrada. |
| Parâmetros | |
| Operador | Seleciona a operação lógica a ser aplicada às entradas do bloco. Parâmetro do bloco: <i>Operator</i> Tipo: Vetor de caracteres Valores: 'AND' 'OR' 'NAND' 'NOR' 'XOR' 'NXOR' 'NOT' Padrão: 'AND' |

| | |
|---|---|
| Número de portas de entrada | Especifica o número de entradas no bloco. Parâmetro do bloco: <i>Inputs</i> Tipo: Vetor de caracteres Valores: Inteiro positivo Padrão: '2' |
| Parâmetros não tratados | |
| <i>Icon shape</i> | Especifica o formato do ícone do bloco. |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |
| <i>Require all inputs and output to have the same data type</i> | Exige que todas as entradas e saídas possuam o mesmo tipo de dados. |
| <i>Output data type</i> | Especifica o tipo de dado da saída entre booleano e lógico. |

Implementação XMV

Declaração

$log_nome_bloco := entrada1$ [Para cada entrada] operador $entradaN$

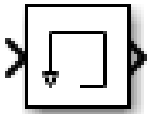
Propriedade para validação

$G(log_nome_bloco=(entrada1$ [Para cada entrada] operador $entradaN))$

A.9 Memória (Memory)

O bloco de Memória, disponível na biblioteca **Simulink / Discrete**, sustenta sua entrada como atraso por um passo de tempo. Blocos de memória são importantes em aplicações que necessitem de valores anteriores (como acumuladores) ou que lidem com problemas de sincronização.

Detalhamento estrutural

| | |
|-------------------------|---|
| Memória (Memory) | |
| Símbolo esquemático |  |
| Biblioteca | Simulink / Discrete |

| Portas | |
|---|--|
| Entrada 1 | Entrada a ser sustentada pelo bloco. |
| Saída 1 | Valor da entrada com atraso de um passo de tempo. |
| Parâmetros | |
| Condição inicial | Especifica o valor inicial armazenado pelo bloco de memória. Parâmetro do bloco: <i>InitialCondition</i> Tipo: Vetor de caracteres Valores: Escalar ou vetor Padrão: '0' |
| Parâmetros não tratados | |
| <i>Inherit sample time</i> | Especifica se o tempo de amostragem deve ser o mesmo da entrada. |
| <i>Direct feedthrough of input during linearization</i> | Especifica se a entrada deve ser apresentada como saída durante o processo de linearização e corte. |
| <i>Treat as a unit delay when linearizing with discrete sample time</i> | Especifica se a saída deve ser linearizada durante atrasos na entrada. |
| <i>Signal object class</i> | Especifica o nome do pacote de classe de armazenamento do dado no bloco. |
| <i>Code generation storage class</i> | Especifica a classe de armazenamento para geração de código no bloco, dadas aplicações com interface a códigos externos. |
| <i>TypeQualifier</i> | Especifica um tipo de qualificador de armazenamento com constante ou volátil. |

Implementação XMV

Declaração

nome_bloco: *block_memory(condicao_inicial, entrada)*

Especificação do módulo

MODULE *block_memory(condicao_inicial, entrada)*

ASSIGN

init(out) := in_init

next(out) := in_data

Propriedade para validação


$out=in_init$

$G(next(out)=(in_data))$

A.10 Multiplexador (Mux)

O bloco Multiplexador, disponível na biblioteca **Simulink / Commonly Used Blocks** combina suas entradas em um único vetor de saída. Ele possui uma função importante inclusive na organização do projeto, condensando barramentos em linhas únicas que são demultiplexadas nos sistemas que as utilizam.

Detalhamento estrutural

| Multiplexador (Mux) | |
|----------------------------|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada N | Valor do N-ésimo sinal a ser multiplexado. |
| Saída 1 | Barramento contendo os sinais de entrada como um vetor. |
| Parâmetros | |
| Number of inputs | Especifica o número de sinais de entrada do multiplexador. Parâmetro do bloco: <i>Inputs</i> Tipo: Escalar, vetor, arranjo de células, lista de nomes de sinal Valores: número, vetor de número de portas, arranjo de células ou lista de nomes de sinal Padrão: '2' |
| Parâmetros não tratados | |
| <i>Display option</i> | Especifica o modo de exibição do ícone do bloco. |

Implementação XMV

Declaração

`textitnome_bloco : block_nome_bloco([Lista de entradas do multiplexador])`

Especificação do módulo

```
MODULE block_nome_bloco([Lista de entradas do multiplexador])
  ASSIGN
    [PARA CADA ENTRADA] out[i] := entradai;
```


Propriedade para validação

```
[PARA CADA ENTRADA] G(out[i] = entradai)
```

A.11 Saída (Outport)

O bloco de saída, disponível na biblioteca **Simulink / Commonly Used Blocks**, liga um sinal de um sistema para fora desse sistema, fornecendo saídas para níveis hierarquicamente superiores. Apesar de não ser modelado como uma função em si, é extremamente importante como elemento de retorno de subsistemas.

Detalhamento estrutural

| Saída (Outport) | |
|-------------------------|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Sinal a ser enviado para um subsistema ou modelo exteno. |
| Saída 1 | O estado integrado da entrada. |
| Parâmetros | |
| Número da porta | Especifica a posição única da porta no bloco que a contém. Parâmetro do bloco: <i>Port</i> Tipo: Vetor de caracteres Valores: inteiro Padrão: '1' |
| Parâmetros não tratados | |
| <i>Signal name</i> | Especifica o nome do sinal correspondente no código gerado. |
| <i>Icon display</i> | Especifica a informação exibida no ícone do bloco. |

| | |
|---|--|
| <i>Specify output when source is unconnected</i> | Especifica se um valor constante deve ser apresentado à saída quando não houver uma fonte conectada. |
| <i>Constant value</i> | Especifica o valor constante de saída quando não houver uma fonte conectada. |
| <i>Interpret vector parameters as 1-D</i> | Trata o parâmetro <i>Constant value</i> como um vetor. |
| <i>Ensure output is virtual</i> | Especifica a saída do bloco como virtual. |
| <i>Source of initial output value</i> | Especifica uma fonte de sinal de saída inicial do bloco. |
| <i>Output when disabled</i> | Especifica o que acontece ao bloco quando o subsistema está desabilitado. |
| <i>Initial output</i> | Especifica que o sinal de saída antes que o subsistema execute subsistemas condicionalmente executáveis. |
| <i>Minimum</i> | Especifica o valor mínimo atribuível à saída pelo Simulink. |
| <i>Maximum</i> | Especifica o valor máximo atribuível à saída pelo Simulink. |
| <i>Data type</i> | Especifica o tipo do sinal dado como entrada. |
| <i>Lock output data type setting against changes by the fixed-point tools</i> | Impede a mudança de tipo da entrada devido a operações de ponto fixo. |
| <i>Output as nonvirtual bus in parent model</i> | Especifica se a entrada fornecida é virtual ou não no modelo hierarquicamente superior. |
| <i>Unit</i> | Especifica a unidade física do sinal de entrada do bloco. |
| <i>Port dimensions (-1 for inherited)</i> | Especifica a dimensão do sinal de saída do bloco. |
| <i>Variable-size signal</i> | Especifica os tipos de sinal assumíveis pelo bloco. |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |
| <i>Signal type</i> | Especifica tipo numérico da saída do bloco. |

Implementação XMV

Declaração

out_nome_bloco := entrada


Propriedade para validação

$G(out_nome_bloco=entrada)$

A.12 Controlador PID pelo método de Euler (PID Controller)

O bloco Controlador PID implementa um controlador PID, PI, PD, P ou I. Sua implementação foi baseada na realizada por Vilhena [Vilhena, 2018].

Detalhamento estrutural

| Controlador PID pelo método de Euler (PID Controller) | |
|---|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Continuous |
| Portas | |
| Entrada 1 | Valor do sinal do erro (referência subtraída da saída da planta) a ser controlado. |
| Entrada 2 | Ganho proporcional. |
| Entrada 3 | Ganho derivativo. |
| Entrada 4 | Coeficiente do filtro derivativo. |
| Entrada 5 | Gatilho de reinicialização externo. |
| Entrada 6 | Condição inicial do integrador. |
| Entrada 7 | Condição inicial do filtro derivativo. |
| Entrada 8 | Sinal de rastreamento da saída do controlador, útil para sistemas que alternam entre dois controladores. |
| Saída 1 | Saída do controlador. |

| Parâmetros | |
|--------------------------------|---|
| | Especifica quais dos termos proporcional, integral e derivativo são utilizados no controlador. |
| Tipo do controlador | Parâmetro do bloco: <i>Controller</i> Tipo: <i>String</i> , vetor de caracteres Valores: "PID", "PI", "PD", "P", "I" Padrão: 'PID' |
| | Especifica o ganho proporcional do controlador. |
| Ganho proporcional | Parâmetro do bloco: <i>P</i> Tipo: escalar ou vetor Padrão: '1' |
| | Especifica o ganho integral do controlador. |
| Ganho integral | Parâmetro do bloco: <i>I</i> Tipo: escalar ou vetor Padrão: '1' |
| | Especifica o ganho derivativo do controlador. |
| Ganho derivativo | Parâmetro do bloco: <i>D</i> Tipo: escalar ou vetor Padrão: '0' |
| | Especifica a condição inicial do integrador no controlador. |
| Condição inicial do integrador | Parâmetro do bloco: <i>InitialConditionForIntegrator</i> Tipo: escalar ou vetor Padrão: '0' |
| | Especifica a condição inicial do derivador não filtrado no controlador. |
| Condição inicial do derivador | Parâmetro do bloco: <i>DifferentiatorICPrevScaledInput</i> Tipo: escalar ou vetor Padrão: '0' |
| Parâmetros não tratados | |
| <i>Form</i> | Especifica se a estrutura do controlador é paralela ou ideal. |
| <i>Time domain</i> | Especifica o controlador como discreto ou contínuo no tempo. |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |
| <i>Integrator method</i> | Especifica o método para computação da integral no tempo discreto. |
| <i>Filter method</i> | Especifica o método para computação da derivada no tempo discreto. |

| | |
|--|--|
| <i>Source</i> | Especifica se a fonte dos ganhos, coeficientes e condições iniciais do controlador é o conjunto de entradas ou de parâmetros. |
| <i>Use filtered derivative</i> | Especifica a aplicação ou não do filtro ao termo derivativo. |
| <i>Filter coefficient</i> | Especifica o valor do filtro derivativo. |
| <i>Select tuning method</i> | Habilita a sintonia automática dos coeficientes do controlador. |
| <i>Enable zero-crossing detection</i> | Habilita a detecção de <i>zero-crossing</i> . |
| <i>Filter</i> | Determina a condição inicial do filtro derivativo no controlador. |
| <i>Initial condition setting</i> | Define quando e se aplicar as condições iniciais do filtro derivativo e do integrador. |
| <i>External reset</i> | Especifica as condições de gatilho para reinicialização dos valores do controlador. |
| <i>Ignore reset when linearizing</i> | Força a computação da linearização do Simulink a ignorar mecanismos de reinicialização do controlador enquanto executa operações de linearização. |
| <i>Enable tracking mode</i> | Habilita o modo de rastreamento no controlador para aplicações como o controle em malha fechada com múltiplos laços. |
| <i>Tracking coefficient</i> | Define o valor do ganho correspondente ao coeficiente de rastreamento. |
| <i>Limit output</i> | Restringe o valor de saída do controlador aos parâmetros de saturação superior e inferior. |
| <i>Upper limit</i> | Valor superior de saturação do controlador. |
| <i>Lower limit</i> | Valor inferior de saturação do controlador. |
| <i>Ignore saturation while linearizing</i> | Força as operações de linearização a ignorar os valores de saturação da saída do controlador. |
| <i>Anti-windup method</i> | Define o uso ou não de métodos para a contenção da geração de valores fora dos limites de saturação no controlador quando sua saída estiver restrita a valores de saturação. |
| <i>Integer rounding mode</i> | Especifica o modo de arredondamento para operações e ponto fixo. |
| <i>Saturate on integer overflow</i> | Especifica se o controlador encapsula ou satura a saída mediante <i>overflow</i> . |

| | |
|---|--|
| <i>Lock data type settings against changes by the fixed-point tools</i> | Previne que ferramentas de aritmética em ponto fixo sobreponham tipos de dados. |
| <i>State name</i> | Define nome para filtros em tempo contínuo ou discreto e estados do integrador. |
| <i>State name must resolve to Simulink signal object</i> | Exige que os estados do integrador ou filtro resolvam em um objeto de sinal do Simulink. |
| <i>Code generation storage class</i> | Seleciona a classe de armazenamento para a geração de código. |
| <i>Code generation storage type qualifier</i> | Especifica o tipo de armazenamento como constante ou volátil. |

Implementação XMV

Declaração

nome_bloco : *block_nome_bloco*(*error*)

Especificação do módulo

MODULE *block_nome_bloco*(*erro*)

VAR

integrator : Real;

previous_error : Real;

ASSIGN

init(*integrator*) := *initial_integrator*;

next(*integrator*) := $I * error + integrator$;

init(*previous_error*) := *initial_error*;

next(*previous_error*) := *error*;

DEFINE

out := $(P * error + I * error + integrator + D * (error - previous_error))$

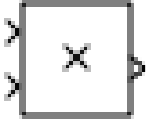
Propriedade para validação

Este módulo foi validado por [Vilhena, 2018] através da obtenção de triplas P, I, D em contraexemplos que, considerando tolerâncias variáveis no erro do controlado, correspondiam a valores encontrados por métodos clássicos na literatura.

A.13 Produto (Product)

O bloco Produto, disponível na biblioteca **Simulink / Commonly Used Blocks**, realiza a multiplicação ou divisão de duas entradas. A operação a ser realizada é definida em um dos parâmetros, bem como seus detalhes.

Detalhamento estrutural

| Produto (Product) | |
|-------------------------|--|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada N | N-ésima entrada a multiplicar ou dividir, fornecida como um escalar, vetor, matriz ou arranjo N-D. |
| Entrada N+1 | Sinal de entrada a ser multiplicado às outras entradas. |
| Entrada N+2 | Sinal de entrada para operações de divisão ou inversão. |
| Saída 1 | Computação da multiplicação, divisão ou inversão. |
| Parâmetros | |
| Condição inicial | Especifica o número de portas de entrada no bloco e se cada entrada é multiplicada ou dividida para gerar a saída. Parâmetro do bloco: <i>Inputs</i> Tipo: Vetor de caracteres Valores: '2' '**' '*' '*/*' ... Padrão: '2' |
| Parâmetros não tratados | |
| <i>Multiplication</i> | Especifica se a multiplicação realizada pelo bloco será orientada por elementos ou se será uma multiplicação de matriz. |
| <i>Multiply over</i> | Especifica se a dimensão da multiplicação será sobre todas as dimensões ou por uma dimensão especificada. |
| <i>Dimension</i> | Especifica a dimensão sobre a qual multiplicar a entrada. |

| | |
|---|--|
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |
| <i>Require all inputs to have the same data type</i> | Exige que todas as entradas possuam o mesmo tipo de dados. |
| <i>Output minimum</i> | Especifica o valor mínimo atribuível à saída pelo Simulink. |
| <i>Output maximum</i> | Especifica o valor máximo atribuível à saída pelo Simulink. |
| <i>Output data type</i> | Especifica o tipo de dado da saída, booleano ou <i>uint8</i> |
| <i>Lock output data type setting against changes by the fixed-point tools</i> | Previne que ferramentas que lidem com aritmética de ponto fixo sobreponham o tipo especificado do bloco. |
| <i>Integer rounding mode</i> | Especifica o modo de arredondamento para operações e ponto fixo. |
| <i>Saturate on integer overflow</i> | Especifica se o controlador encapsula ou satura a saída mediante <i>overflow</i> . |

Implementação XMV

$(prod_nome_bloco) := [Para\ cada\ entrada,\ seu\ nome\ e\ operação\ (multiplicação\ ou\ divisão)];$

Propriedade para validação

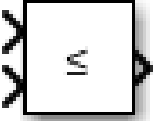
$G(prod_nome_bloco = ([Para\ cada\ entrada,\ seu\ nome\ e\ operação\ (multiplicação\ ou\ divisão)]))$

A.14 Operador relacional (Relational Operator)

O bloco Operador Relacional, disponível na biblioteca **Simulink / Commonly Used Blocks**, realiza a operação relacional especificada sobre a entrada. A operação a ser realizada é definida através de um parâmetro (igualdade, maior, menor, igual ou maior, igual e menor, diferença) e a saída é o resultado binário sobre a operação realizada.

Detalhamento estrutural

| |
|--|
| Operador relacional (Relational Operator) |
|--|

| | |
|--|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Primeiro sinal de entrada, especificado como escalar, vetor ou matriz. |
| Entrada 2 | Segundo sinal de entrada, especificado como escalar, vetor ou matriz. |
| Saída 1 | Resultado da operação relacional entre as entradas, consistindo de 0's e 1's com a mesma dimensão das entradas. |
| Parâmetros | |
| Operador relacional | Especifica a operação relacional a ser executada sobre as entradas. Parâmetro do bloco: <i>Operator</i> Tipo: Vetor de caracteres Valores: '==' '=' '<' '<=' '>=' '>' 'isInf' 'isNaN' 'isFinite' Padrão: '<=' |
| Parâmetros não tratados | |
| <i>Enable zero-crossing detection</i> | Habilita a detecção de <i>zero-crossing</i> . |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco devido a mudanças no parâmetro de Valor constante. |
| <i>Require all inputs to have the same data type</i> | Exige que todas as entradas possuam o mesmo tipo de dados. |
| <i>Output data type</i> | Especifica o tipo de dado da saída, booleano ou <i>uint8</i> |
| <i>Integer rounding mode</i> | Especifica o modo de arredondamento para operações e ponto fixo. |

Implementação XMV

Declaração

```
relop_nome_bloco := entrada1 operacao entrada2;
```

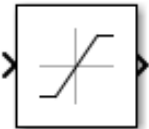
Propriedade para validação

```
G(relop_nome_bloco = (entrada1 operacao entrada2))
```

A.15 Saturação (Saturation)

O bloco Saturação, disponível na biblioteca **Simulink / Commonly Used Blocks**, produz um sinal de saída que é o sinal da entrada limitado a valores de saturação superior e inferior. Ele é importante para representar fenômenos físicos submetidos a limites, além de ser importante para verificações de violação de limiar.

Detalhamento estrutural

| Saturação (Saturation) | |
|---------------------------------------|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Valor do sinal a ser submetido à saturação. |
| Saída 1 | Valor da entrada submetido aos limites de saturação. |
| Parâmetros | |
| Limite superior | Especifica o limite superior de saturação do bloco. Parâmetro do bloco: <i>UpperLimit</i> Tipo: Vetor de caracteres Valor: Real escalar ou vetor Padrão: '0.5' |
| Limite inferior | Especifica o limite inferior de saturação do bloco. Parâmetro do bloco: <i>LowerLimit</i> Tipo: Vetor de caracteres Valor: Real escalar ou vetor Padrão: '-0.5' |
| Parâmetros não tratados | |
| <i>Treat as gain when linearizing</i> | Especifica se os comandos de linearização do Simulink devem tratar o ganho do bloco como 0 ou 1. |
| <i>Enable zero-crossing detection</i> | Habilita a detecção de <i>zero-crossing</i> . |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco devido a mudanças no parâmetro de Valor constante. |

Implementação XMV

Declaração

nome_bloco : *block_nome_bloco(entrada)*

Especificação do módulo

MODULE *block_nome_bloco(entrada)*

DEFINE

```

    out := case
        entrada > upper_limit : upper_limit;
        entrada < lower_limit : lower_limit;
    TRUE : entrada;
    esac;

```


Propriedade para validação

$G((out \leq upper_limit) \& (out \geq lower_limit))$

A.16 Soma (Sum)

O bloco de Soma, disponível na biblioteca **Simulink / Math Operations**, realiza a soma ou subtração de suas entradas. O tipo da operação é definida através de um parâmetro.

Detalhamento estrutural

| | |
|---------------------|---|
| Soma (Sum) | |
| Símbolo esquemático |  |
| Biblioteca | Simulink / Math Operations |
| Portas | |
| Entrada N | O N-ésimo operando a ser somado ou subtraído. |
| Saída 1 | Resultado das operações de soma e subtração das entradas. |
| Lista de sinais | |

| | |
|---|--|
| Condição inicial | Especifica para cada entrada se ela será somando ou subtraindo. Parâmetro do bloco: <i>Inputs</i> Tipo: Vetor de caracteres Valores: '+' '-' integer Padrão: '++' |
| Parâmetros não tratados | |
| <i>Icon shape</i> | Especifica formato do ícone do bloco. |
| <i>Sum over</i> | Especifica especifica a dimensão sobre a qual o bloco realiza a operação de soma. |
| <i>Dimension</i> | Especifica a dimensão sobre a qual somar a entrada. |
| <i>Sample time</i> | Especifica o intervalo de tempo para a atualização da saída do bloco. |
| <i>Require all inputs to have the same data type</i> | Exige que todas as entradas possuam o mesmo tipo de dados. |
| <i>Accumulator data type</i> | Escolhe o tipo de dado do acumulador. |
| <i>Output minimum</i> | Especifica o valor mínimo atribuível à saída pelo Simulink. |
| <i>Output maximum</i> | Especifica o valor máximo atribuível à saída pelo Simulink. |
| <i>Output data type</i> | Especifica o tipo de dado da saída, booleano ou <i>uint8</i> |
| <i>Lock output data type setting against changes by the fixed-point tools</i> | Previne que ferramentas que lidem com aritmética de ponto fixo sobreponham o tipo especificado do bloco. |
| <i>Integer rounding mode</i> | Especifica o modo de arredondamento para operações e ponto fixo. |
| <i>Saturate on integer overflow</i> | Especifica se o controlador encapsula ou satura a saída mediante <i>overflow</i> . |

Implementação XMV

Declaração

sum_nome_bloco := [Para cada entrada, seu nome e operação (soma ou subtração)];

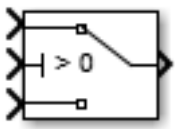
Propriedade para validação

$G(\text{sum_nome_bloco} = ([\text{Para cada entrada, seu nome e operação (soma ou subtração)]))$

A.17 Comutador (Switch)

O bloco Comutador, disponível na biblioteca **Simulink / Commonly Used Blocks**, transmite uma de suas entradas à saída dependendo do resultado de uma operação de controle. Ele é um dos principais componentes de desvio de fluxo, aparecendo predominantemente em unidades de controle.

Detalhamento estrutural

| Comutador (Switch) | |
|---------------------------------------|---|
| Símbolo esquemático |  |
| Biblioteca | Simulink / Commonly Used Blocks |
| Portas | |
| Entrada 1 | Primeiro sinal de entrada. |
| Entrada 2 | Sinal de controle. |
| Entrada 3 | Segundo sinal de entrada. |
| Saída 1 | O valor de uma das entradas dependendo do critério de controle. |
| Parâmetros | |
| Critério | Seleciona a condição sob a qual o bloco transmitirá o sinal da primeira entrada pela saída. Parâmetro do bloco: <i>Criteria</i> Tipo: Vetor de caracteres Valores: 'u2 >= Threshold' 'u2 > Threshold' 'u2 =0' Padrão: 'u2 >= Threshold' |
| Limiar | Assinala o limiar utilizado para a aplicação do critério. Parâmetro do bloco: <i>Threshold</i> Tipo: Vetor de caracteres Valores: 'off' 'on' Padrão: 'off' |
| Parâmetros não tratados | |
| <i>Enable zero-crossing detection</i> | Habilita a detecção de <i>zero-crossing</i> . |

| | |
|---|--|
| <i>Output minimum</i> | Especifica o valor mínimo atribuível à saída pelo Simulink. |
| <i>Output maximum</i> | Especifica o valor máximo atribuível à saída pelo Simulink. |
| <i>Output data type</i> | Especifica o tipo de dados da saída do bloco. |
| <i>Lock output data type setting against changes by the fixed-point tools</i> | Previne que ferramentas que lidem com aritmética de ponto fixo sobreponham o tipo especificado do bloco. |
| <i>Integer rounding mode</i> | Especifica o modo de arredondamento para operações e ponto fixo. |
| <i>Saturate on integer overflow</i> | Especifica se o controlador encapsula ou satura a saída mediante <i>overflow</i> . |
| <i>Allow different data input sizes</i> | Especifica se o bloco aceitará entradas de tamanhos diferentes. |

Implementação XMV

Declaração

nome_bloco : *block_switch_simple*(*entrada_v*,*erdadeiro*,*controle*,*entrada_f**also*);

Especificação do módulo

MODULE *block_switch_simple*(*entrada_v*,*erdadeiro*,*controle*,*entrada_f**also*)

DEFINE

```

    out := case
    controle : entradaverdadeiro;
    !controle : entradafalso;
    esac;
```

Propriedade para validação

$G((controle \& (out = entrada_v \text{erdadeiro})) | (!controle \& (out = entrada_f \text{also})))$

Apêndice B

Identificadores de estruturas do Simulink Comuns em Modelos VANT

A partir da leitura de modelos de controle VANT em Simulink [Turetta, 2018, Konig, 2018b, Konig, 2018a] identificamos blocos comuns que devem ser processados por um compilador Simulink → Linguagem de verificação formal. Apresentamos aqueles identificados que vão desde blocos mais simples, representando multiplicações, até blocos mais complexos, representando a necessidade de abertura e processamento de arquivos externos. A seguir, estruturamos os identificadores mais comuns encontrados nos modelos VANT no Simulink com seu nome, principais parâmetros e blocos instanciados (Quando múltiplos são seguidos de um [], que quando vazio representa a incapacidade de abstrair sua quantidade dada a análise sintática do modelo. Nomes de blocos entre parênteses listados entre os parâmetros indicam que aqueles atributos são encontrados em instâncias daquele tipo) e descrição sucinta.

| Nome | Principais Parâmetros | Descrição |
|-----------------------|---|------------------------------------|
| Model | Name Graphical Interface Object[] SimulationMode ParamWorkspaceSource Array(Handle)[] Simulink.ConfigSet BlockDefaults AnnotationDefaults LineDefaults MaskDefaults MaskParameterDefaults BlockParameterDefaults | Modelo de um sistema |
| ExternalFileReference | Reference Path SID Type | Importação de um arquivo externo |
| GraphicalInterface | NumExternalFileReferences ExternalFileReference[] | Configurações de interface gráfica |
| Object | PropName ObjectID ClassName IsActive Visible LoadSaveID ID Model Array(Cell)[] | Objeto genérico |

| | | |
|---------------------------|---------------------------|--|
| Array | Type | |
| | Dimension | |
| | Cell[Dimension] | |
| | PropName | |
| | (Simulink.ConfigSet) | |
| | Simulink.SolverCC | |
| | Simulink.DataIOCC | |
| | Simulink.OptimizationCC | |
| | Simulink.DebuggingCC | Arranjo que contém |
| | Simulink.HardwareCC | instâncias de componentes |
| | Simulink.ModelReferenceCC | |
| | Simulink.SFSimCC | |
| | Simulink.RTWCC | |
| SICovCC.ConfigComp | | |
| hdlcoderui.hdlcc | | |
| (Simulink.RTWCC) | | |
| Simulink.CodeAppCC | | |
| Simulink.GRTTargetCC | | |
| Simulink.ConfigSet | ObjectID | |
| | Array(Handle) | Conjunto de configurações |
| | Name | para um determinado bloco |
| Simulink.SolverCC | CurrentDlgPage | |
| | ObjectID | Configurações do resolvidor |
| | ObjectID | |
| Simulink.DataIOCC | ExternalInput | Configurações de entrada e |
| | FinalStateName | saída |
| | InitialState | |
| Simulink.OptimizationCC | ObjectID | Configurações de otimização |
| | Array(Cell) | |
| Simulink.DebuggingCC | ObjectIDm Array(Cell) | Configurações de depuração |
| Simulink.HardwareCC | ObjectID | Configurações de uso de hardware |
| Simulink.ModelReferenceCC | ObjectID | Configurações a referências internas a modelos |
| Simulink.SFSimCC | ObjectID | Configurações de simulação |

| | | |
|------------------------|---|---|
| Simulink.RTWCC | ObjectID BackupClass Array(Cell) Array(Handle) SystemTargetFile | Configurações de compilação RTW |
| Simulink.CodeAppCC | ObjectID Array(Cell) | Configurações de codificação |
| Simulink.GRTTargetCC | ObjectID BackupClass Array(Cell) | Configurações de compilação |
| SICovCC.ConfigComp | ObjectID Name CovScope | Configurações de cobertura de componentes |
| hdlcoderui.hdlcc | ObjectID Name Description | Configurações do codificador HDL |
| BlockDefaults | NamePlacement | Padrões para instâncias de blocos |
| AnnotationDefaults | - | Padrões para instâncias de anotações |
| LineDefaults | - | Padrões para instâncias de linhas |
| MaskDefaults | - | Padrões para instâncias de máscaras |
| MaskParameterDefaults | - | Padrões para parâmetros de máscaras |
| BlockParameterDefaults | Block[] | Padrões para os parâmetros de cada tipo de blocos |
| Block | BlockType Name SID Ports Port[] System | A instância de um bloco em um sistema |
| Block(BusCreator) | - | Cria um barramento |

| | | |
|---------------------------|---|---|
| Block(BusSelector) | - | Seletor de barramentos |
| Block(Concatenate) | NumInputs Mode ConcatenateDimension | Bloco de concatenação de barramentos |
| Block(Constant) | Value | Definição de constante |
| Block(DataTypeConversion) | - | Conversão de tipos de valores |
| Block(Demux) | Outputs | Demultiplexador |
| Block(From) | GotoTag TagVisibility | Captação de valores de um Goto |
| Block(Gain) | Gain RndMeth | Operador de ganho |
| Block(Goto) | GotoTag TagVisibility | Inserção de dados a serem referenciados no código |
| Block(If) | NumInputs ifExpression ShowElse ZeroCross SampleTime | Operador condicional |
| Block(Inport) | Port OutputFunctionCall SampleTime | Porta de entrada de blocos |
| Block(Logic) | Operator Inputs | Portas lógicas |
| Block(Outport) | Port InitialOutput | Porta de saída de blocos |
| Block(Product) | Inputs OutDataTypeStr SaturateOnIntegerOverflow SampleTime | Produto entre as entradas |
| Block(Reference) | SourceBlock SourceType SourceProductName | Instância de um ExternalFileReference |

| | | |
|-------------------|--|---|
| Block(SubSystem) | SystemSampleTime | Um subsistema para implementações hierárquicas |
| Block(Sum) | Inputs RndMeth SaturateOnIntegerOverflow SampleTime | Acumulador |
| Block(Switch) | Criteria Threshold InputSameDT OutMin OutMax RndMeth SaturateOnIntegerOverflow SampleTime | Chaveamento |
| Block(Terminator) | - | Terminador |
| Port | PortNumber Name | Porta de um bloco |
| Line | Name SrcBlock SrcPort DstBlock DstPort Branch[] | Linhas que interconectam blocos no fluxo de dados |
| Branch | DstBlock DstPort | Ramificações de uma linha |
| System | Name Block(SubSystem) Block(Inport) Block(Reference) Line[] | Um sistema modelado |

Anexo A

Códigos resumidos dos modelos-exemplo traduzidos

A.1 Código do cálculo de direção

O modelo de para a definição do comando utilizado para o cálculo de direção apresentado na Figura 3.19 possui como código (resumido ao texto processado pelo tradutor SMCT, excluídos os parâmetros não utilizados):

```
Model {
  Name      "calculo_direcao"
  ...
  BlockParameterDefaults {
    Block {
      BlockType      Constant
      Value          "1"
      ...
    }
    Block {
      BlockType      Inport
      Port           "1"
      ...
    }
    Block {
      BlockType      Memory
      X0             "0"
      InheritSampleTime      off
      LinearizeMemory      off
      LinearizeAsDelay      off
      StateMustResolveToSignalObject      off
      RTWStateStorageClass  "Auto"
    }
    Block {
      BlockType      Output
      Port           "1"
      ...
    }
    Block {
      BlockType      RelationalOperator
      Operator       ">="
      ...
    }
    Block {
      BlockType      Switch
      Criteria       "u2 >= Threshold"
      Threshold      "0"
      ...
    }
  }
  System {
    Name      "calculo_direcao"
    ...
    Block {
      BlockType      Inport
      Name           "dXe"
      SID            "1"
      ...
    }
  }
}
```

```

Block {
BlockType      Inport
Name           "dZe"
SID           "2"
...
Port          "2"
...
}
Block {
BlockType      Constant
Name           "0"
SID           "3"
...
}
Block {
BlockType      Constant
Name           "1"
SID           "4"
...
Value         "-1"
}
Block {
BlockType      Constant
Name           "Baixo"
SID           "5"
...
Value         "2"
}
Block {
BlockType      Constant
Name           "Cima"
SID           "6"
...
}
Block {
BlockType      Constant
Name           "Direita"
SID           "7"
...
Value         "4"
}
Block {
BlockType      Constant
Name           "Esquerda"
SID           "8"
...
Value         "3"
}
Block {
BlockType      Memory
Name           "Memory2"
SID           "9"
...
}
Block {
BlockType      Constant
Name           "Parar"
SID           "10"
...
Value         "0"
}
Block {
BlockType      RelationalOperator
Name           "Relational\nOperator1"
SID           "11"
...
Operator      "<"
}
Block {
BlockType      RelationalOperator
Name           "Relational\nOperator2"
SID           "12"
...
Operator      ">"
}
Block {
BlockType      RelationalOperator
Name           "Relational\nOperator3"
SID           "13"
...
Operator      "<"
}
Block {
BlockType      RelationalOperator
Name           "Relational\nOperator4"
SID           "14"
...
Operator      ">"
}
Block {
BlockType      Switch
Name           "Se delta Xe<0 vá para esquerda"
SID           "15"
...
Criteria      "u2 > Threshold"
}
Block {
BlockType      Switch
Name           "Se delta Xe>0 vá para direita"
SID           "16"
...
Criteria      "u2 > Threshold"
}
Block {

```

```

BlockType      Switch
Name           "Se delta Ze>0 vá para baixo.\nSe não, para cima"
SID           "17"
...
Criteria       "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Se delta Ze>0 vá para cima"
SID           "18"
...
Criteria       "u2 > Threshold"
...
}
Block {
BlockType      Outport
Name           "Comando"
SID           "19"
...
}
Line {
...
SrcBlock       "0"
SrcPort        1
...
Branch {
...
DstBlock       "Relational\nOperator1"
DstPort        2
}
Branch {
...
DstBlock       "Relational\nOperator3"
DstPort        2
}
Branch {
...
DstBlock       "Relational\nOperator2"
DstPort        2
}
}
Line {
...
SrcBlock       "Relational\nOperator2"
SrcPort        1
...
DstBlock       "Se delta Xe>0 vá para direita"
DstPort        2
}
Line {
...
SrcBlock       "Se delta Ze>0 vá para baixo.\nSe não, parar."
SrcPort        1
...
DstBlock       "Se delta Ze>0 vá para cima"
DstPort        3
}
Line {
...
SrcBlock       "Se delta Ze>0 vá para cima"
SrcPort        1
DstBlock       "Comando"
DstPort        1
}
}

```

```

Line {
...
SrcBlock      "Relational\nOperator1"
SrcPort      1
...
DstBlock      "Se delta Xe<0 vá para esquerda"
DstPort      2
}
Line {
...
SrcBlock      "Se delta Xe>0 vá para direita"
SrcPort      1
DstBlock      "Memory2"
DstPort      1
}
Line {
...
SrcBlock      "dZe"
SrcPort      1
...
Branch {
...
DstBlock      "Relational\nOperator3"
DstPort      1
}
Branch {
...
DstBlock      "Relational\nOperator4"
DstPort      1
}
}
Line {
...
SrcBlock      "Direita"
SrcPort      1
...
DstBlock      "Se delta Xe>0 vá para direita"
DstPort      1
}
Line {
...
SrcBlock      "Cima"
SrcPort      1
...
DstBlock      "Se delta Ze>0 vá para cima"

```

```

DstPort      1
}
...
SrcBlock      "Se delta Xe<0 vá para esquerda"
SrcPort      1
...
DstBlock      "Se delta Xe>0 vá para direita"
DstPort      3
}
Line {
...
SrcBlock      "dXe"
SrcPort      1
...
Branch {
...
DstBlock      "Relational\nOperator1"
DstPort      1
}
Branch {
...
DstBlock      "Relational\nOperator2"
DstPort      1
}
}
Line {
...
SrcBlock      "Baixo"
SrcPort      1
...
DstBlock      "Se delta Ze>0 vá para baixo.\nSe não, parar."
DstPort      1
}
Line {
...
SrcBlock      "Parar"
SrcPort      1
...
DstBlock      "Se delta Ze>0 vá para baixo.\nSe não, parar."
DstPort      3
}
}
}

```

A.2 Código do cálculo de orientação [Turetta, 2018]

O modelo de para a definição do comando utilizado para o cálculo de direção apresentado nas Figuras 3.25 e 3.26 possui como código (resumido ao texto processado pelo tra-

duor SMCT, excluídos os parâmetros não utilizados):

```

Model {
  Name      "orientacao_turretta"
  ...
  ...
  BlockParameterDefaults {
    Block {
      BlockType      BusSelector
    }
    Block {
      BlockType      Constant
      Value          "1"
    }
    ...
  }
  Block {
    BlockType      Inport
    Port           "1"
    ...
  }
  Block {
    BlockType      Logic
    Operator        "AND"
    Inputs          "2"
    ...
  }
  Block {
    BlockType      Memory
    X0              "0"
    ...
  }
  Block {
    BlockType      Outport
    Port           "1"
    ....
  }
  Block {
    BlockType      RelationalOperator
    Operator        ">="
    ...
  }
  Block {
    BlockType      Switch
    Criteria        "u2 >= Threshold"
    Threshold       "0"
    ...
  }
  System {
    Name      "orientacao_turretta"
    ...
    Block {
      BlockType      Inport
      Name           "GPS"
      SID            "1"
      ...
    }
    ...
    Block {
      BlockType      Inport
      Name           "Pilot_Altitude"
      SID            "2"
      ...
      Port           "2"
    }
    ...
    Block {
      BlockType      Inport
      Name           "Obstacle_Sensor"
      SID            "3"
      ...
      Port           "3"
    }
    ...
    Block {
      BlockType      Inport
      Name           "BACK_CMD"
      SID            "4"
      ...
      Port           "4"
    }
    ...
    Block {
      BlockType      Inport
      Name           "FWD_CMD"
      SID            "5"
      ...
      Port           "5"
    }
    ...
    Block {
      BlockType      Inport
      Name           "Steady_Lat_CMD"
      SID            "6"
      ...
      Port           "6"
    }
    ...
    Block {
      BlockType      BusSelector
      Name           "Bus\nSelector"
      SID            "7"
      ...
      Port {
        PortNumber 1
        Name       "<X>"
      }
      Port {
        PortNumber 2
  
```

```

Name "<Z>"
}
}
Block {
BlockType BusSelector
Name "Bus\nSelector1"
SID "8"
...
Port {
PortNumber 1
Name "<Obs_Above>"
}
Port {
PortNumber 2
Name "<Obs_Below>"
}
Port {
PortNumber 3
Name "<Obs_Right>"
}
Port {
PortNumber 4
Name "<Obs_Left>"
}
}
Block {
BlockType Reference
Name "Compare\nTo Constant"
SID "9"
...
SourceType "Compare To Constant"
...
relop "=="
const "1"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant1"
SID "10"
...
SourceType "Compare To Constant"
...
relop "=="
const "1"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant2"
SID "11"
...
SourceType "Compare To Constant"
...
relop "=="
const "1"
...
}
const "1"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant3"
SID "12"
...
SourceType "Compare To Constant"
...
relop "=="
const "0"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant4"
SID "13"
...
SourceType "Compare To Constant"
...
relop "=="
const "0"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant5"
SID "14"
...
SourceType "Compare To Constant"
...
relop "=="
const "1"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant6"
SID "15"
...
SourceType "Compare To Constant"
...
relop "=="
const "0"
...
}
Block {
BlockType Reference
Name "Compare\nTo Constant7"
...
SourceType "Compare To Constant"
...
relop "=="
const "0"
...
}

```



```

...
}
Block {
BlockType      Reference
Name           "Compare\nTo Constant8"
SID            "17"
...
SourceType     "Compare To Constant"
...
relop          "=="
const          "1"
...
}
Block {
BlockType      Constant
Name           "Constant2"
SID            "18"
...
Value          "0"
}
Block {
BlockType      Constant
Name           "Constant3"
SID            "19"
...
Value          "-0.01"
}
Block {
BlockType      Constant
Name           "Constant4"
SID            "20"
...
Value          "0.01"
}
Block {
BlockType      Logic
Name           "Logical\nOperator10"
SID            "21"
...
Operator       "NOT"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator11"
SID            "22"
...
Operator       "NOT"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator12"
SID            "23"
...
Operator       "NOT"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator13"
SID            "50"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator14"
SID            "51"
...
Operator       "NOT"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator6"
SID            "24"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator7"
SID            "25"
...
Inputs         "3"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator8"
SID            "26"
...
Inputs         "3"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator9"
SID            "27"
...
Inputs         "3"
...
}
Block {
BlockType      Memory
Name           "Memory1"
SID            "52"
...
}
Block {
BlockType      Memory

```

```

Name      "Memory2"
SID      "28"
...
}
Block {
BlockType      Memory
Name      "Memory3"
SID      "29"
...
}
Block {
BlockType      Memory
Name      "Memory4"
SID      "30"
...
}
Block {
BlockType      Memory
Name      "Memory5"
SID      "31"
...
}
Block {
BlockType      RelationalOperator
Name      "Relational\nOperator2"
SID      "32"
...
Operator      "<"
...
}
Block {
BlockType      RelationalOperator
Name      "Relational\nOperator3"
SID      "33"
...
Operator      ">"
...
}
Block {
BlockType      Switch
Name      "Switch"
SID      "40"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name      "Switch1"
SID      "41"
...
Criteria      "u2 > Threshold"
...
}
Block {
Name      "Memory2"
BlockType      Switch
Name      "Switch2"
SID      "42"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name      "Switch3"
SID      "43"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name      "Switch4"
SID      "44"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name      "Switch5"
SID      "45"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name      "Switch6"
SID      "46"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Outputport
Name      "Pitch_Target"
SID      "47"
...
}
Block {
BlockType      Outputport
Name      "Target_Altitude"
SID      "48"
...
Port      "2"
...
}
Line {
Name      "<Obs_Left>"

```

```

...
SrcBlock      "Bus\nSelector1 "
SrcPort       4
...
}
Line {
Name          "<Obs_Right>"
...
SrcBlock      "Bus\nSelector1 "
SrcPort       3
...
}
Line {
Name          "<X>"
...
SrcBlock      "Bus\nSelector"
SrcPort       1
...
}
Line {
...
SrcBlock      "Memory3"
SrcPort       1
...
DstBlock      "Switch2"
DstPort       3
}
Line {
...
SrcBlock      "Switch2"
SrcPort       1
...
DstBlock      "Switch"
DstPort       3
}
Line {
Name          "<Z>"
...
SrcBlock      "Bus\nSelector"
SrcPort       2
...
Branch {
...
Branch {
...
DstBlock      "Switch6"
DstPort       1
}
Branch {
...
DstBlock      "Relational\nOperator3"
DstPort       2
}
}
Branch {
...
DstBlock      "Relational\nOperator2"
DstPort       2
}
}
Line {
...
SrcBlock      "Constant2"
SrcPort       1
...
DstBlock      "Switch1"
DstPort       1
}
Line {
...
SrcBlock      "Logical\nOperator11"
SrcPort       1
DstBlock      "Logical\nOperator9"
DstPort       2
}
Line {
...
SrcBlock      "Switch6"
SrcPort       1
...
Branch {
...
DstBlock      "Memory4"
DstPort       1
}
Branch {
...
DstBlock      "Switch5"
DstPort       3
}
}
Line {
...
SrcBlock      "Switch1"
SrcPort       1
...
Branch {
...
DstBlock      "Memory3"
DstPort       1
}
Branch {
...
DstBlock      "Pitch_Target"
DstPort       1
}
}
Line {
...
SrcBlock      "Switch5"

```

```

SrcPort      1
...
DstBlock     "Switch4"
DstPort      3
}
Line {
...
SrcBlock     "Compare\nTo Constant5"
SrcPort      1
DstBlock     "Logical\nOperator7"
DstPort      1
}
Line {
...
SrcBlock     "BACK_CMD"
SrcPort      1
DstBlock     "Memory2"
DstPort      1
}
Line {
...
SrcBlock     "Compare\nTo Constant7"
SrcPort      1
DstBlock     "Logical\nOperator8"
DstPort      1
}
Line {
...
SrcBlock     "Logical\nOperator13"
SrcPort      1
...
DstBlock     "Switch6"
DstPort      2
}
Line {
...
SrcBlock     "Memory2"
SrcPort      1
DstBlock     "Compare\nTo Constant2"
DstPort      1
}
Line {
ZOrder      25
SrcBlock     "Constant4"
SrcPort      1
...
DstBlock     "Switch2"
DstPort      1
}
Line {
...
SrcBlock     "Pilot_Altitude"
SrcPort      1
...
Branch {
...
Branch {
...
DstBlock     "Relational\nOperator3"
DstPort      1
}
Branch {
...
DstBlock     "Relational\nOperator2"
DstPort      1
}
}
Branch {
...
Branch {
...
Branch {
...
DstBlock     "Switch3"
DstPort      1
}
Branch {
...
DstBlock     "Switch4"
DstPort      1
}
}
Branch {
...
DstBlock     "Switch5"
DstPort      1
}
}
Line {
Name         "<Obs_Above>"
...
SrcBlock     "Bus\nSelector1"
SrcPort      1
...
Branch {
...
Branch {
...
DstBlock     "Compare\nTo Constant3"
DstPort      1
}
Branch {
...
DstBlock     "Compare\nTo Constant5"
DstPort      1
}
}
Branch {
...

```

```

DstBlock "Compare\nTo Constant7"
DstPort 1
}
}
Line {
...
SrcBlock "Switch3"
SrcPort 1
DstBlock "Target_Altitude"
DstPort 1
}
Line {
...
SrcBlock "Switch4"
SrcPort 1
...
DstBlock "Switch3"
DstPort 3
}
Line {
...
SrcBlock "Compare\nTo Constant1"
SrcPort 1
DstBlock "Switch1"
DstPort 2
}
Line {
...
SrcBlock "Obstacle_Sensor"
SrcPort 1
DstBlock "Bus\nSelector1"
DstPort 1
}
Line {
...
SrcBlock "Compare\nTo Constant2"
SrcPort 1
DstBlock "Switch2"
DstPort 2
}
Line {
...
SrcBlock "Logical\nOperator10"
SrcPort 1
DstBlock "Logical\nOperator9"
DstPort 1
}
Line {
...
SrcBlock "Compare\nTo Constant8"
SrcPort 1
DstBlock "Logical\nOperator8"
DstPort 2
}
Line {
...
SrcBlock "Logical\nOperator12"
SrcPort 1
DstBlock "Logical\nOperator9"
DstPort 3
}
Line {
...
SrcBlock "Logical\nOperator6"
SrcPort 1
...
Branch {
...
DstBlock "Switch3"
DstPort 2
}
Branch {
...
DstBlock "Logical\nOperator12"
DstPort 1
}
}
Line {
...
SrcBlock "Relational\nOperator3"
SrcPort 1
DstBlock "Logical\nOperator8"
DstPort 3
}
Line {
...
SrcBlock "Compare\nTo Constant"
SrcPort 1
DstBlock "Switch"
DstPort 2
}
Line {
...
SrcBlock "Relational\nOperator2"
SrcPort 1
DstBlock "Logical\nOperator7"
DstPort 3
}
Line {
...
SrcBlock "Logical\nOperator8"
SrcPort 1
...
Branch {
...
DstBlock "Switch5"
DstPort 2
}
Branch {
...

```



```

}
Block {
BlockType      Inport
Name           "Z"
SID           "2"
...
Port          "2"
...
}
Block {
BlockType      Constant
Name           "Altura de cruzeiro"
SID           "3"
...
Value         "304"
}
Block {
BlockType      Constant
Name           "Altura de motor horizontal"
SID           "4"
...
Value         "60"
}
Block {
BlockType      Constant
Name           "Comando descer"
SID           "5"
...
Value         "2"
}
Block {
BlockType      Logic
Name           "Comando horizontal"
SID           "6"
...
Operator      "OR"
...
}
Block {
BlockType      Constant
Name           "Comando para direita"
SID           "7"
...
Value         "4"
}
Block {
BlockType      Constant
Name           "Comando para esquerda"
SID           "8"
...
Value         "3"
}
Block {
BlockType      Constant
Name           "Comando parar"
SID           "9"
...
Value         "0"
}
Block {
BlockType      Constant
Name           "Comando subir"
SID           "10"
...
}
Block {
BlockType      Switch
Name           "Condição de\ndecolagem"
SID           "11"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Condição de cruzeiro"
SID           "12"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Condição de nave estática"
SID           "13"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Condição de nave pousada"
SID           "14"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Condição de pouso"
SID           "15"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Condição de recusa\nhorizontal"
SID           "16"
...
}

```



```

Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Switch
Name           "Condição de transição"
SID           "17"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Constant
Name           "Cruzeiro"
SID           "18"
...
Value         "2"
}
Block {
BlockType      Constant
Name           "Decolagem"
SID           "19"
...
Value         "3"
}
Block {
BlockType      Constant
Name           "Desconhecido"
SID           "20"
...
Value         "7"
}
Block {
BlockType      Logic
Name           "Logical\nOperator"
SID           "21"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator1"
SID           "22"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator2"
SID           "23"
...
Operator      "NOT"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator3"
...
Criteria      "u2 > Threshold"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator6"
SID           "25"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator7"
SID           "26"
...
Operator      "NOT"
...
}
Block {
BlockType      Logic
Name           "Logical\nOperator8"
SID           "27"
...
}
Block {
BlockType      Constant
Name           "Nave em pouso"
SID           "28"
...
Value         "0"
}
Block {
BlockType      Constant
Name           "Nave estática"
SID           "29"
...
}
Block {
BlockType      Constant
Name           "Nave pousada"
SID           "30"
...
Value         "0"
}
Block {
BlockType      Constant
Name           "Pouso"
SID           "31"
...
Value         "5"
}
Block {
BlockType      Constant
Name           "Recusa\nhorizontal"
SID           "32"
...

```

| | | | |
|-----------|-------------------------|-----------|-----------------------------|
| Value | "6" | BlockType | RelationalOperator |
| } | | Name | "Relational\nOperator8" |
| Block { | | SID | "40" |
| BlockType | RelationalOperator | ... | |
| Name | "Relational\nOperator1" | Operator | "==" |
| SID | "33" | ... | |
| ... | | } | |
| Operator | "==" | Block { | |
| ... | | BlockType | Constant |
| } | | Name | "Transição" |
| Block { | | SID | "41" |
| BlockType | RelationalOperator | ... | |
| Name | "Relational\nOperator2" | Value | "4" |
| SID | "34" | } | |
| ... | | Block { | |
| Operator | "==" | BlockType | Outport |
| ... | | Name | "Estado" |
| } | | SID | "42" |
| Block { | | ... | |
| BlockType | RelationalOperator | } | |
| Name | "Relational\nOperator3" | Line { | |
| SID | "35" | ... | |
| ... | | SrcBlock | "Z" |
| } | | SrcPort | 1 |
| Block { | | ... | |
| BlockType | RelationalOperator | Branch { | |
| Name | "Relational\nOperator4" | ... | |
| SID | "36" | Branch { | |
| ... | | ... | |
| } | | DstBlock | "Relational\nOperator3" |
| Block { | | DstPort | 1 |
| BlockType | RelationalOperator | } | |
| Name | "Relational\nOperator5" | Branch { | |
| SID | "37" | ... | |
| ... | | DstBlock | "Relational\nOperator2" |
| Operator | "==" | DstPort | 2 |
| ... | | } | |
| } | | } | |
| Block { | | Branch { | |
| BlockType | RelationalOperator | ... | |
| Name | "Relational\nOperator6" | DstBlock | "Relational\nOperator4" |
| SID | "38" | DstPort | 1 |
| ... | | } | |
| Operator | "==" | } | |
| ... | | Line { | |
| } | | ... | |
| Block { | | SrcBlock | "Condição de nave estática" |
| BlockType | RelationalOperator | SrcPort | 1 |
| Name | "Relational\nOperator7" | ... | |
| SID | "39" | DstBlock | "Condição de nave pousada" |
| ... | | DstPort | 3 |
| Operator | "==" | } | |
| ... | | Line { | |
| } | | ... | |
| Block { | | SrcBlock | "Comando" |

```

...
Branch {
...
Branch {
...
Branch {
...
Branch {
...
DstBlock      "Relational\nOperator8"
DstPort      1
}
}
Branch {
...
DstBlock      "Relational\nOperator7"
DstPort      1
}
}
Branch {
...
DstBlock      "Relational\nOperator6"
DstPort      1
}
}
Branch {
...
DstBlock      "Relational\nOperator5"
DstPort      1
}
}
Branch {
...
DstBlock      "Relational\nOperator1"
DstPort      1
}
}
Line {
...
SrcBlock      "Logical\nOperator3"
SrcPort      1
DstBlock      "Condição de cruzeiro"
DstPort      2
}
}
Line {
...
SrcBlock      "Relational\nOperator4"
SrcPort      1
...
DstBlock      "Logical\nOperator6"
DstPort      2
}
}
Line {
...
SrcBlock      "Logical\nOperator"
SrcPort      1
...
Branch {
...
DstBlock      "Logical\nOperator2"
DstPort      1
}
}
Branch {
...
DstBlock      "Condição de nave pousada"
DstPort      2
}
}
Line {
...
SrcBlock      "Logical\nOperator8"
SrcPort      1
DstBlock      "Condição de recusa\nhorizontal"
DstPort      2
}
}
Line {
...
SrcBlock      "Comando horizontal"
SrcPort      1
...
Branch {
...
Branch {
...
DstBlock      "Logical\nOperator8"
DstPort      2
}
}
Branch {
...
DstBlock      "Logical\nOperator6"
DstPort      1
}
}
}
Branch {
ZOrder      ...
DstBlock      "Logical\nOperator3"
DstPort      1
}
}
Line {
...
SrcBlock      "Relational\nOperator1"
SrcPort      1
...
Branch {
...
DstBlock      "Logical\nOperator1"
DstPort      2
}
}
Branch {
...

```

| | | | |
|----------|-----------------------------|----------|------------------------------|
| DstBlock | "Logical\nOperator" | DstPort | 1 |
| DstPort | 1 | } | |
| } | | Line { | |
| } | | ... | |
| Line { | | SrcBlock | "Transição" |
| ... | | SrcPort | 1 |
| SrcBlock | "Altura de cruzeiro" | ... | |
| SrcPort | 1 | DstBlock | "Condição de transição" |
| DstBlock | "Relational\nOperator3" | DstPort | 1 |
| DstPort | 2 | } | |
| } | | Line { | |
| Line { | | ... | |
| ... | | SrcBlock | "Comando parar" |
| SrcBlock | "Cruzeiro" | SrcPort | 1 |
| SrcPort | 1 | DstBlock | "Relational\nOperator1" |
| ... | | DstPort | 2 |
| DstBlock | "Condição de cruzeiro" | } | |
| DstPort | 1 | Line { | |
| } | | ... | |
| Line { | | SrcBlock | "Condição de cruzeiro" |
| ... | | SrcPort | 1 |
| SrcBlock | "Comando para direita" | ... | |
| SrcPort | 1 | DstBlock | "Condição de nave estática" |
| DstBlock | "Relational\nOperator6" | DstPort | 3 |
| DstPort | 2 | } | |
| } | | Line { | |
| Line { | | ... | |
| ... | | SrcBlock | "Altura de motor horizontal" |
| SrcBlock | "Relational\nOperator2" | SrcPort | 1 |
| SrcPort | 1 | DstBlock | "Relational\nOperator4" |
| ... | | DstPort | 2 |
| DstBlock | "Logical\nOperator" | } | |
| DstPort | 2 | Line { | |
| } | | ... | |
| Line { | | SrcBlock | "Relational\nOperator6" |
| ... | | SrcPort | 1 |
| SrcBlock | "Comando subir" | ... | |
| SrcPort | 1 | DstBlock | "Comando horizontal" |
| DstBlock | "Relational\nOperator7" | DstPort | 2 |
| DstPort | 2 | } | |
| } | | Line { | |
| Line { | | ... | |
| ... | | SrcBlock | "Condição de transição" |
| SrcBlock | "Nave estática" | SrcPort | 1 |
| SrcPort | 1 | ... | |
| ... | | DstBlock | "Condição de cruzeiro" |
| DstBlock | "Condição de nave estática" | DstPort | 3 |
| DstPort | 1 | } | |
| } | | Line { | |
| Line { | | ... | |
| ... | | SrcBlock | "Comando para esquerda" |
| SrcBlock | "Logical\nOperator2" | SrcPort | 1 |
| SrcPort | 1 | ... | |
| ... | | DstBlock | "Relational\nOperator5" |
| DstBlock | "Logical\nOperator1" | DstPort | 2 |

```

}
Line {
...
SrcBlock      "Nave em pouso"
SrcPort       1
...
DstBlock      "Relational\nOperator2"
DstPort       1
}
Line {
...
SrcBlock      "Logical\nOperator6"
SrcPort       1
...
Branch {
...
DstBlock      "Logical\nOperator7"
DstPort       1
}
Branch {
...
DstBlock      "Condição de transição"
DstPort       2
}
}
Line {
...
SrcBlock      "Logical\nOperator1"
SrcPort       1
DstBlock      "Condição de nave estática"
DstPort       2
}
Line {
...
SrcBlock      "Condição de recusa\nhorizontal"
SrcPort       1
...
DstBlock      "Condição de transição"
DstPort       3
}
Line {
...
SrcBlock      "Relational\nOperator8"
SrcPort       1
DstBlock      "Condição de pouso"
DstPort       2
}
Line {
...
SrcBlock      "Pouso"
SrcPort       1
...
DstBlock      "Condição de pouso"
DstPort       1
}
}
Line {
...
SrcBlock      "Relational\nOperator3"
SrcPort       1
...
DstBlock      "Logical\nOperator3"
DstPort       2
}
Line {
...
SrcBlock      "Condição de pouso"
SrcPort       1
...
DstBlock      "Condição de\ndecolagem"
DstPort       3
}
Line {
...
SrcBlock      "Logical\nOperator7"
SrcPort       1
...
DstBlock      "Logical\nOperator8"
DstPort       1
}
Line {
...
SrcBlock      "Desconhecido"
SrcPort       1
...
DstBlock      "Condição de pouso"
DstPort       3
}
Line {
...
SrcBlock      "Recusa\nhorizontal"
SrcPort       1
...
DstBlock      "Condição de recusa\nhorizontal"
DstPort       1
}
Line {
...
SrcBlock      "Condição de nave pousada"
SrcPort       1
DstBlock      "Estado"
DstPort       1
}
Line {
...
SrcBlock      "Relational\nOperator5"
SrcPort       1
...
DstBlock      "Comando horizontal"
DstPort       1
}
}

```

```

Line {
...
SrcBlock      "Comando descer"
SrcPort       1
DstBlock      "Relational\nOperator8"
DstPort       2
}
Line {
...
SrcBlock      "Condição de\ndecolagem"
SrcPort       1
...
DstBlock      "Condição de recusa\nhorizontal"
DstPort       3
}
Line {
...
SrcBlock      "Decolagem"
SrcPort       1
...
}

DstBlock      "Condição de\ndecolagem"
DstPort       1
}
Line {
...
SrcBlock      "Nave pousada"
SrcPort       1
...
DstBlock      "Condição de nave pousada"
DstPort       1
}
Line {
...
SrcBlock      "Relational\nOperator7"
SrcPort       1
DstBlock      "Condição de\ndecolagem"
DstPort       2
}
}
}

```