Gerenciamento de configuração de uma linha de produtos de software de veículos aéreos não tripulados

Eduardo Miranda Steiner

Gerenciamento de configuração de uma linha de produtos de software de veículos aéreos não tripulados

Eduardo Miranda Steiner

Orientador: **Prof. Dr. Paulo Cesar Masiero**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Maio de 2012

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Steiner, Eduardo
S818g Gerenciamento de configuração de uma linha de produtos de software de veículos aéreos não tripulados / Eduardo Steiner; orientador Paulo Masiero. -- São Carlos, 2012.
101 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2012.

1. Linha de Produtos de Software. 2. Sistemas Embarcados. 3. Veículos Aéreos não Tripulados. 4. Simulink. I. Masiero, Paulo, orient. II. Título.

Agradecimentos

Agradeço aos meus pais, Eliana e João, pelo carinho e esforço que empregaram na minha criação e educação e pelos inúmeros ensinamentos e exemplos.

Ao meu orientador, Prof. Paulo Masiero, pela confiança, conselhos e conhecimentos transmitidos durante o trabalho.

Ao Jorge e ao Luciano da AGX, aos professores Kalinka, Rosana, Onofre (do ICMC), Rodrigo Bonifácio (da UnB) e Itana (da UEM) pelo envolvimento, ajuda e conselhos dados no trabalho.

Um agradecimento aos parentes e amigos que sempre me apoiaram e inspiraram: meus irmãos Renato e Ronaldo, minhas primas Carol e Rubia, os velhos amigos Matheus, Karla, Johnny, Daniel, Leo, Tuzi, Gugs, André, Manu, Flavia, Alexey, Dedo, Zacani, Monteiro, Rods, Lucas, Claudinha, Coxa, Jason, Victor, Esper, Mari; aos amigos feitos em São Carlos: Indio, Messi, Nalbert, Leozao, Chicken, Billy, Ale, Fresh, Fran, Xotz e a todos os amigos do LABES, em especial Draylson, Harry, Cabeça, Luciano, Neiza, Silvana, Joice, Rodolfo, Marcão e André.

Finalmente, agradeço à CAPES pelo apoio financeiro.

“Depende de nós praticarmos atos nobres ou vis;
e se é isso que se entende por ser bom ou mau,
então depende de nós sermos virtuosos ou viciosos.”

Aristóteles

Resumo

Veículos Aéreos não Tripulados (VANTs) são aeronaves que voam sem tripulação e são capazes de realizar diversos tipos de missões, como vigilância, coleta de dados topográficos e monitoramento ambiental. Este é um domínio que tem muito a ganhar com a aplicação da abordagem de Linha de Produtos de Software (LPS), uma vez que é rico em variabilidades e cada modelo de VANT tem também muitas partes comuns. Neste trabalho é apresentada uma infraestrutura tecnológica e de configuração de ativos em Simulink, gerenciados pelas ferramentas Pure::variant e Hephaestos para uma LPS de VANTs. Um conjunto de padrões para especificação de variabilidades em Simulink é proposto, bem como uma extensão para a ferramenta Hephaestus. Uma comparação entre as ferramentas Pure::variants e Hephaestus é apresentada.

Abstract

Unmanned Aerial Vehicles (UAVs) are aircrafts that can fly without any crew and are capable to realize several types of missions such as surveillance, topographic data collection and environmental monitoring. This is a domain which can benefit very much with the adoption of the Software Product Lines (SPL) approach, as each UAV model is rich in variabilities and has many common parts. In this work it is presented a software asset configuration infrastructure for the Simulink environment, managed by the tools Pure::variants and Hephaestus for a UAV SPL. A set of patterns of variability specification in Simulink is proposed as well as an extension to Hephaestus to support a SPL product engineering for Simulink. A comparison between Pure::variants and Hephaestus is also presented.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo

1

Introdução

1.1 Contextualização

Sistemas embarcados (SE) estão assumindo uma importância muito grande atualmente e estão presentes de muitas maneiras na vida das pessoas nos mais variados tipos de aplicação como, por exemplo, sistemas que controlam os programas de uma máquina de lavar louça, telefones celulares e componentes de carros como pilotos automáticos, bombas de injeção eletrônica e freios ABS. Os SE estão também sendo aplicados em outros problemas complexos, como automação de plantas industriais e automação de residências (Cetina et al., 2009; Ebert e Salecker, 2009; Liggesmeyer e Trapp, 2009). São sistemas que têm restrições importantes em relação a diferentes requisitos não funcionais, tais como desempenho, uso de memória e segurança, e também devem resistir a ambientes de uso hostis (alta temperatura, por exemplo).

Os sistemas embarcados podem ser de diferentes tipos, entre os quais estão os sistemas embarcados críticos (SEC), sistemas caracterizados por possuírem restrição crítica em relação ao tempo de atendimento de suas requisições (sistemas de tempo real rígido) e em relação à segurança e confiabilidade (Bouhraoua et al., 2010). Normalmente eles estão dentro de produtos físicos que são vendidos em diferentes modelos e versões. Apesar de serem constantemente reusados, o reúso propriamente dito é uma atividade complexa no desenvolvimento desses sistemas, uma vez que pode existir grande variabilidade em

hardware, software, requisitos, etc (Clements e Northrop, 2001; Faust e Verhoef, 2003; Karsai et al., 2010).

Um tipo de aplicação no domínio de sistemas embarcados críticos que tem crescido em importância nos últimos anos é o de veículos autônomos, em especial veículos aéreos não tripulados (VANTs). Um VANT pode ser definido como uma estrutura de aeronave e um sistema computacional que são integrados a sensores, GPS, atuadores e processadores. Todos esses elementos combinados devem controlar uma aeronave sem qualquer intervenção humana (Pastor et al., 2006). Essas aeronaves foram concebidas inicialmente para aplicações militares, pelo fato de trazerem soluções seguras e de relativo baixo custo para diversos tipos de missões como, por exemplo, vigilância, espionagem e bloqueio ou interferência em radares, mas hoje em dia estão sendo cada vez mais utilizadas para aplicações civis, científicas e comerciais como medições meteorológicas em altas altitudes, coleta de dados topográficos, avaliação de situações catastróficas e monitoração de agricultura, pesca e meio ambiente.

Entre as diversas ferramentas e ambientes para o desenvolvimento de sistemas embarcados destaca-se a ferramenta de modelagem e simulação *MATLAB/Simulink*. Segundo Ebert e Jones (2009), essa ferramenta tem participação no mercado superior a 50%. Ela possibilita a modelagem de sistemas dinâmicos lineares, não-lineares, contínuos ou discretos no tempo. Essa ferramenta possui uma biblioteca de blocos bastante completa, cada qual com características e funções diferentes. Os modelos são construídos por meio da composição e conexão de diferentes blocos. Além de se modelar, analisar, simular e testar, com a ferramenta também é possível gerar código C e VHDL automaticamente a partir dos seus modelos com o plugin *Real-Time Workshop* (MathWorks, 2007).

Uma abordagem que tem obtido sucesso no reúso de software em sistemas embarcados é a de LPS (Linha de Produtos de Software), também referido como "família de software" por alguns autores (Bayer et al., 1999; Polzer et al., 2009). Essa abordagem tem sido usada desde meados dos anos 90 e tem como ideia central desenvolver o software embarcado como uma linha de produtos (Clements e Northrop, 2001; Sugumaran et al., 2006; Van Der Linden et al., 2007). Segundo essa técnica, o reúso é controlado e guiado pelos produtos que se pretende desenvolver. Assim, há uma parte do software que é comum a todos os produtos – compondo, portanto, a arquitetura reusável da LPS – e outra que varia conforme o produto e aparece em um ou mais produtos da mesma linha.

Um conceito importante em LPS é o de *feature* (que pode ser traduzido para o português como "característica", mas não se trata de uma tradução amplamente adotada) que refere-se a partes (artefatos) do sistema em questão que representam as variantes funcionais que atendem às necessidades de clientes. Essas partes de sistema podem ser comuns

ou variáveis em relação aos produtos da LPS e constituem os ativos (ou *assets*, em inglês) ou componentes da linha.

Nas duas últimas décadas, a abordagem de LPS tem recebido muita atenção por levar à redução efetiva em tempo e esforço no desenvolvimento de produtos e ao grande número de casos de sucesso da sua aplicação (Pohl et al., 2005; Van Der Linden et al., 2007); diversas ferramentas (Beuche, 2003; Bonifácio et al., 2009; Cirilo et al., 2007; Kästner et al., 2008), técnicas (Botterweck et al., 2010; Krueger, 2002a; Myllymäki, 2002) e estudo de casos foram feitos de modo a evoluir e refinar a teoria e a prática para desenvolver LPS (Buhrdorf et al., 2004; Lutz, 2008; Pohl e Metzger, 2006; Yoshimura et al., 2006).

1.2 Motivação

O crescente uso de VANTs em aplicações civis foi uma das principais motivações para a parceria entre a empresa AGX (AGX, 2012) e o INCT-SEC (Instituto Nacional de Ciência e Tecnologia de Sistemas Embarcados Críticos) (INCT-SEC, 2012). Dessa parceria surgiram alguns projetos de VANTs, entre os quais há um que está na fase inicial de produção e venda, o VANT Tiriba. Esse VANT foi desenvolvido usando o ambiente MATLAB/Simulink, o que facilitou o reúso, diminuindo o tempo e esforço na manutenção e na implementação da aeronave e também permitiu geração automática de código (todo o código utilizado nos processadores do VANT foi gerado automaticamente a partir do seu modelo Simulink).

Uma das ideias que está na base da criação do INCT-SEC é o desenvolvimento de uma LPS de VANTs. Essa ideia é factível por diversos motivos: o domínio de VANTs possui um considerável número de variabilidades, as aeronaves possuem muitas características comuns entre si e no Instituto estão em curso ou foram finalizados projetos de VANTs em MATLAB/Simulink (por exemplo SarVANT, SGA3 e Arara).

Apesar de o desenvolvimento de LPS de modelos Simulink parecer muito promissor, considerando as vantagens da abordagem de LPS e do desenvolvimento de sistemas segundo o desenvolvimento dirigido a modelos do Simulink, existem poucos trabalhos e ferramentas nesse contexto. A única ferramenta comercial usada nesse domínio é a Pure::variants (Beuche, 2003; Dziobek et al., 2008). A instanciação de produtos por essa ferramenta se dá por meio de atribuição de valores a blocos de controle, o que faz com que sejam executados apenas blocos relacionados a variantes funcionais selecionadas para a instância de produto.

1.3 Objetivo

Considerando o contexto e a motivação apresentados nas seções anteriores, este trabalho tem dois objetivos. O primeiro objetivo, mais geral, é estudar e definir como pode ser realizado o gerenciamento de configurações em uma LPS desenvolvida em MATLAB/Simulink. Essa prova de conceito é realizada com base em uma LPS simplificada, criada a partir do modelo Simulink do VANT Tiriba. Pretende-se usar e avaliar ferramentas de apoio ao gerenciamento de configuração em LPS em Simulink, a Pure::variants e também o Hephaestus (desenvolvida por pesquisadores brasileiros) (Bonifácio et al., 2009).

Um segundo objetivo do trabalho é realizar uma extensão na ferramenta Hephaestus, que é usada na fase de engenharia do produto no desenvolvimento de LPS. Nessa extensão pretende-se fazer com que a ferramenta Hephaestus apoie o desenvolvimento e geração de produtos em LPS modelados em Simulink.

Espera-se que este trabalho sirva como guia para o desenvolvimento e configuração de LPS em MATLAB/Simulink, independentemente do domínio do sistema em questão.

1.4 Organização

No Capítulo 2 os sistemas embarcados são introduzidos. Nele são apresentadas as terminologias, principais conceitos básicos, características dos projetos de sistemas deste domínio e o ambiente MATLAB/Simulink. Além disso, também são exploradas as principais características de sistemas embarcados críticos e do domínio de VANTs.

No Capítulo 3 é apresentada uma revisão bibliográfica sobre LPS, destacando-se os principais conceitos, técnicas, abordagens de adoção, processos e ferramentas para gerenciamento de configuração de LPS. No final do capítulo é explorada a utilização da abordagem de LPS para o domínio de veículos autônomos.

No Capítulo 4 é introduzido um conjunto de padrões para implementações de variabilidades de diferentes tipos de *features* e diferentes relações entre *features* em modelos Simulink. Além disso, as *features* criadas no Tiriba e no seu modelo Simulink são apresentadas. Por fim, é mostrado como as *features* foram estruturadas no modelo Simulink seguindo os padrões introduzidos no começo do capítulo.

No Capítulo 5 é mostrado o funcionamento das ferramentas Pure::variants (com o conector para Simulink) e Hephaestus para o gerenciamento de configuração da LPS criada neste trabalho a partir do modelo Simulink do VANT Tiriba. No final do capítulo é apresentada uma comparação entre as duas ferramentas.

Por fim, no Capítulo 6 são apresentadas as considerações finais, principais contribuições do trabalho e trabalhos futuros.

Capítulo

2

Sistemas Embarcados

2.1 Considerações Iniciais

Este capítulo caracteriza o domínio de sistemas embarcados. Na Seção 2.2 é apresentada uma introdução aos sistemas embarcados, com suas principais características e restrições. Na Seção 2.3, o projeto de sistemas embarcados é introduzido; nesta seção é também descrito o desenvolvimento dirigido por modelo e as ferramentas e ambientes que tem sido usadas para o desenvolvimento de sistemas no domínio, com destaque para o ambiente MATLAB/Simulink – o qual será utilizado na parte prática deste trabalho, conforme exposto no Capítulo 1. Na Seção 2.4, os sistemas embarcados críticos são introduzidos; nesta seção o destaque é dado para o domínio dos veículos aéreos não tripulados (VANTs) – domínio que é foco deste trabalho.

2.2 Introdução aos Sistemas Embarcados

Sistemas embarcados são definidos por Marwedel (2003) como "*Sistemas de processamento de informação que estão integrados em um produto maior e que normalmente não são diretamente visíveis pelo usuário*". De acordo com Lee (2006), software embarcado é definido como "*um software integrado a processos físicos*". Exemplos de sistemas embarcados são bens eletrônicos de consumo (como câmeras digitais, televisões e tocadores mp3), sistemas

de automação industrial, sistemas aviônicos e eletrônicos automotivos (como freios ABS e *airbags*).

Ao contrário de sistemas computacionais de propósito geral, os sistemas embarcados tem em foco uma aplicação específica. Além disso, eles são caracterizados por um grande número de restrições que normalmente devem ser levadas em conta em seus projetos, entre as quais estão:

- *Limitação de recursos computacionais* (como capacidade de processamento e memória): sistemas embarcados costumam ter recursos computacionais limitados e em muitos casos proibitivos, o que exige de projetistas e programadores de aplicação experiência e conhecimento sobre as tecnologias envolvidas no desenvolvimento do sistema.

- *Custo*: Para projetos de sistemas embarcados com foco em altos volumes de venda (caso principalmente de eletrônicos de consumo), o mercado costuma ser bastante competitivo e, para se garantir sucesso na venda de produtos, é preciso haver gerenciamento eficiente do orçamento e do esforço de desenvolvimento de hardware e software.

- *Energia*: o consumo de energia é uma restrição-chave no projeto de sistemas embarcados. Componentes de hardware costumam gerar calor, o que implica necessidade de resfriamento, que por sua vez significa integrar ao produto dispositivos caros ou que ocupem um espaço indesejado. No caso de sistemas embarcados cuja energia é fornecida por bateria, o consumo de energia se torna uma restrição ainda mais crítica, uma vez que está diretamente ligado ao tempo de vida do sistema.

Tempo real é uma outra restrição muito importante existente na maior parte dos SE. Sistemas embarcados de tempo real podem ser vistos como um conjunto de tarefas ou processos associados a um sistema operacional de tempo real ou a um executivo cíclico (em inglês *cyclic executive*). A corretude de tais tarefas depende não só do resultado lógico dos processamentos, mas também do momento em que os resultados são produzidos.

Esses sistemas atribuem prazos a suas tarefas para determinar o tempo limite que elas têm para terminar sua execução. Segundo Li e Yao (2003), as tarefas desses sistemas costumam ser classificadas em dois tipos:

- Tempo real rígido (*hard real-time*): Um sistema que possua tarefas de tempo real rígido deve ser projetado para que em hipótese alguma suas tarefas percam prazos. Esse tipo de tarefa é comum em sistemas embarcados críticos (descritos na Seção 2.4), em que uma perda de prazo pode causar até mesmo perda de vidas humanas.

Exemplos desse tipo de tarefa podem ser encontrados nos sistemas que controlam os freios *ABS* de carros.

- Tempo real moderado (*soft real-time*): são tarefas com restrições temporais, mas há um certo grau de flexibilidade em relação à perda de prazos; geralmente a perda de prazos resulta em uma queda na qualidade da aplicação. Tarefas desse tipo podem ser encontradas, por exemplo, em aparelhos de DVD: caso a decodificação de algum frame seja perdida, o resultado pode nem mesmo ser percebido pelo usuário.

Nem todos os sistemas embarcados são de tempo real e nem todos os sistemas de tempo real são embarcados (conforme é ilustrado na Figura 2.1). Porém, uma vez que quase todos os sistemas embarcados precisam interagir com o mundo real, tempo real passa a ser um item presente na maior parte desses sistemas, desde os pequenos sistemas para uso pessoal como mp3 players e aparelhos de DVD, até sistemas embarcados críticos, como os de aviões, carros ou de automação industrial.

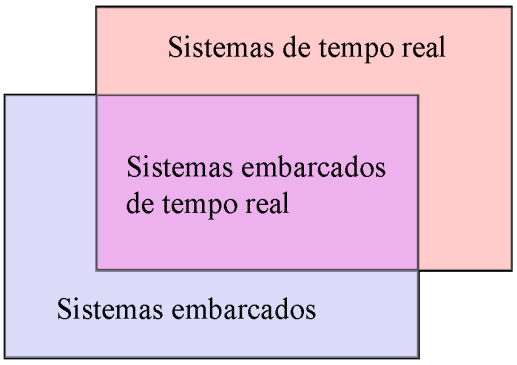


Figura 2.1: Conjunto dos sistemas embarcados, sistemas de tempo real e sistemas embarcados de tempo real.

A Figura 2.2 mostra uma visão alto nível de comportamento típico de grande parte dos sistemas embarcados: informações sobre o ambiente no qual os sistemas embarcados estão inseridos são captadas por sensores e, depois de algum processamento da informação, uma resposta apropriada deverá ser dada por meio de atuadores.

Um exemplo de sistema embarcado que possui um comportamento como o descrito da Figura 2.2 é o marca-passo, que é um dispositivo que tem o objetivo de regular batimentos cardíacos por meio de estímulos elétricos. Fazendo uso de sensores ele analisa o ritmo, comprimento e amplitude dos batimentos cardíacos. Essas informações são digitalizadas a partir de conversores AD (analógico/digital) e o computador integrado ao marca-passo verifica se há alguma anormalidade no batimento e, se houver, calcula um sinal elétrico adequado que deverá ser convertido por meio de conversores DA (digital/analógico) e depois será aplicado pelos atuadores ao coração, corrigindo a anormalidade detectada.

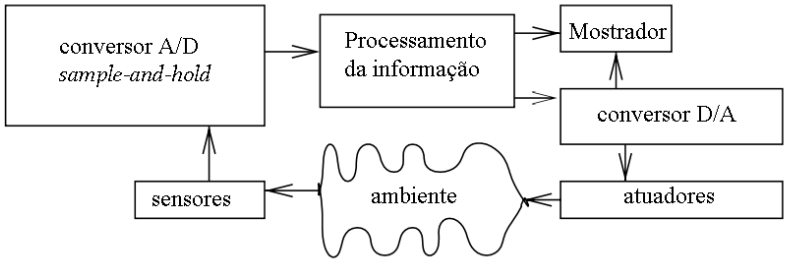


Figura 2.2: Visão alto nível do comportamento de um sistema embarcado (adaptada de Marwedel (2003)).

Sistemas embarcados são cada vez mais comuns no dia a dia das pessoas. Segundo Ebert e Jones (2009), em 2008 havia 30 microprocessadores embarcados por pessoa em países desenvolvidos com ao menos 2.5 milhões de pontos de função de software embarcado. Os computadores de propósito geral, grandes centros de TI (tecnologia da informação) e aplicações online possuem menos de 2% dos microprocessadores fabricados no mundo – todos os outros encontram-se em sistemas embarcados (Ebert e Salecker, 2009). O mercado mundial de sistemas embarcados está orçada em cerca de 160 bilhões de euros e há um crescimento anual de 9%. De acordo com Marwedel (2003), sistemas embarcados são considerados a área de aplicação mais importante da TI para os próximos anos.

O crescimento da complexidade e quantidade de software existente em sistemas embarcados é uma tendência observada e exposta em alguns trabalhos (Ebert e Jones, 2009; Marwedel, 2003; Muller, 2003; Obbink et al., 2000). Estudos mostram que a quantidade de software existente em aparelhos de televisão têm tido um crescimento exponencial: de 1979 para 1990 houve um crescimento na ordem de 2^6 e de 1990 para 2000, um crescimento na ordem de 2^5 (Obbink et al., 2000).

A Figura 2.3 mostra a evolução de sistemas embarcados em termos do tamanho de software ao longo dos últimos anos; os sistemas considerados são softwares embarcados de veículos espaciais, sistemas de comutação de telecomunicações, sistemas automotivos e o kernel do Linux que é usado como base para muitos sistemas embarcados.

O maior problema trazido pelo aumento da quantidade de software para sistemas embarcados é que isso é uma ameaça à confiabilidade. A ameaça existe pelo fato de que software costuma ter erro: segundo Muller (2003), a densidade comum de erros encontrados em código é da ordem de 3 erros para cada 1000 linhas (em casos de processos de desenvolvimento muito bons para organizações maduras, essa ordem é de 1 ou 2 erros para cada 1000 linhas).

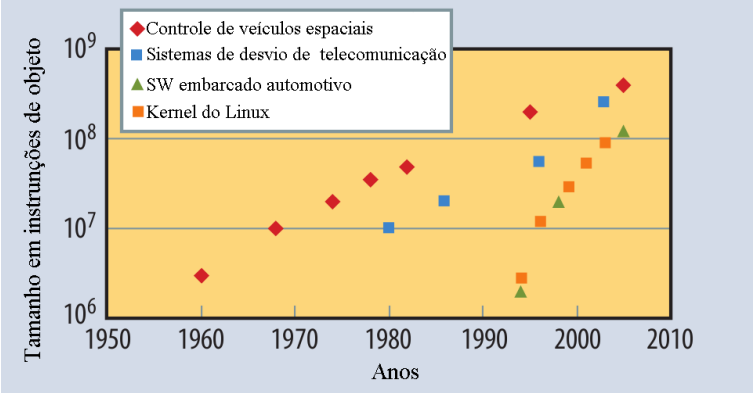


Figura 2.3: Crescimento da complexidade do software de diferentes tipos de sistemas embarcados ao longo dos últimos anos (adaptada de Ebert e Jones (2009)).

2.3 Projeto de Sistemas Embarcados

Projetar sistemas embarcados costuma ser uma tarefa muito complexa. Segundo Ebert e Jones (2009), a maior fonte de complexidade é o grande número de interações sutis e muitas vezes inesperadas entre várias partes dos sistemas, as quais possuem as seguintes características:

- Funcionalidade representada por estados e eventos
- Comportamento de tempo real de eventos e ações esperadas
- Combinação de sistemas de hardware e software com computadores, sensores e atuadores distribuídos
- Altas demandas por disponibilidade, segurança e interoperabilidade
- Sistemas de longa duração, cujo software embarcado deve funcionar com segurança

Uma maneira clássica de definir estados e eventos no projeto de sistemas embarcados é com *diagramas de transição de estados*. Esses diagramas descrevem máquinas de estados finitos comunicantes que se baseiam no conceito de comunicação por memória compartilhada. A Figura 2.4 mostra um exemplo de máquina de estados finitos. Os círculos representam estados, as arestas, eventos (que implicam em transições de estados). É possível observar que os rótulos dos estados não se repetem (uma vez que os estados são únicos), mas os rótulos das arestas podem repetir (já que um determinado evento pode ocorrer em diferentes estados em que a máquina esteja). Normalmente as máquinas de estados estão em apenas um estado em cada instante, mas existem outros tipos de diagramas de transição de estados nos quais é possível que haja ao mesmo tempo mais de um estado ativo (Marwedel, 2003).

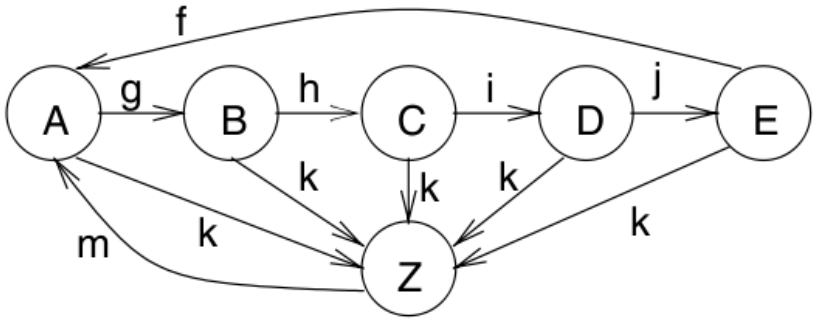


Figura 2.4: Diagrama de transição de estados (Marwedel, 2003).

2.3.1 Desenvolvimento Dirigido por Modelo de Sistemas Embarcados

Um modelo é uma descrição, especificação ou abstração de um sistema e seu ambiente que está mais próximo dos conceitos do domínio do que os conceitos computacionais (ou algorítmicos). Os modelos são representados como combinação de diagramas e textos e expressos por meio de uma linguagem de modelagem ou linguagem natural (OMG, 2005).

Gerenciar a complexidade crescente do desenvolvimento de sistemas embarcados é um dos desafios mais importantes para aumentar a qualidade dos produtos, redução do tempo de desenvolvimento e redução dos custos de desenvolvimento. Desenvolvimento Dirigido a Modelos (em inglês, *Model Driven Development* ou MDD) é uma abordagem de desenvolvimento de sistemas que surgiu nesse contexto, com a qual é possível automatizar o processo de desenvolvimento por meio da execução de modelos, transformações e geração de código.

Em vez de desenvolverem o software diretamente usando linguagens de programação, desenvolvedores modelam sistemas de software usando notações gráficas que são mais intuitivas e expressivas e provêm um nível de abstração mais alto que de linguagens de programação convencionais. Nessa abordagem, geradores criam automaticamente o código que implementa as funcionalidades do sistema. Para gerenciar a complexidade crescente de sistemas embarcados, esse tipo de modelagem provavelmente irá substituir codificação manual de desenvolvimento a nível de aplicação, da mesma maneira que linguagens de programação de alto nível substituiram quase completamente o uso da linguagem assembly (Liggesmeyer e Trapp, 2009).

De acordo com Bunse et al. (2007), as promessas do desenvolvimento dirigido a modelos podem ser obtidas no desenvolvimento de sistemas embarcados. Seus estudos indicam que a abordagem de desenvolvimento dirigido a modelos aplicada ao desenvolvimento de

sistemas embarcados atingiu um nível sistemático de reúso, levando à redução do esforço de desenvolvimento e aumento da qualidade do sistema.

2.3.2 Ferramentas e Ambientes de Desenvolvimento

De acordo com Ebert e Jones (2009), uma característica comum a todo o domínio de software embarcado é o uso de linguagens de programação que permitem acesso direto a interfaces, memória e outros componentes de hardware. Mais de 80% das companhias do domínio utilizam a linguagem *C* e um pouco de *C++*. Mais de 40% usam *assembler* para interfaces de baixo nível. O uso da linguagem *Java* está crescendo para interfaces gráficas do usuário e programação de aplicação. Ferramentas de desenvolvimento baseadas no ambiente *Eclipse* dominam bancadas de engenharia porque muitas ferramentas diferentes estão integradas na IDE, como ferramentas de modelagem e simulação (por exemplo, *Matlab/Simulink*, *Rose* e *Tau*), ambientes de teste (*LabView*, *CANoe*, *HIL/SIL* e *emuladores*), ambientes de gerenciamento do ciclo de vida do produto (*Teamcenter* e *eASEE*), ferramentas de gerenciamento de configuração (*SVN* e *CVS*), ferramentas para apoio à engenharia de requisito (*DOORS* e *Caliber*), além de compiladores e depuradores. Devido à intensiva interação com fornecedores e colaboradores, que é bem maior do que na TI tradicional, ferramentas como *DOORS* ou *Matlab/Simulink* têm participação no mercado superiores a 50%.

MATLAB/Simulink

MATLAB/Simulink é uma ferramenta baseada em modelos para análise, modelagem e simulação de sistemas dinâmicos desenvolvida pela Mathworks (Dabney e Harman, 1997; MathWorks, 2012). A ferramenta usa a estrutura matemática do MATLAB e possibilita o desenvolvimento de sistemas dinâmicos lineares, não-lineares, contínuos ou discretos no tempo; exemplos de sistemas físicos que podem ser projetados e simulados com MATLAB/Simulink são robôs, carros e aviões.

Com o Simulink, desenvolvedores podem especificar completamente sistemas de software embarcado usando modelos alto nível. Essa modelagem é feita por meio de modelagem de blocos e a execução e comunicação entre os blocos usa um modelo temporal síncrono.

A ferramenta possui uma biblioteca de blocos em que cada um representa um processo dinâmico diferente. Os blocos são classificados nos seguintes tipos:

- **Blocos Fontes**: são usados para gerar sinais. Exemplos de blocos desse tipo são: *número aleatório* e `sequência de repetição`.

- **Blocos Sumidouros**: são usados como saída ou para mostrar saída de sinais. Exemplos de blocos desse tipo são: `arquivo` e `gráfico XY`.

- **Blocos Discretos**: são elementos de tempo discreto de sistemas dinâmicos. Exemplos de blocos desse tipo são: `integrador de tempo discreto` e `atrasador de unidade` (ou `delay`) – a saída desse bloco é igual à entrada, porém com algum atraso a ser determinado pelo desenvolvedor.

- **Blocos Lineares**: são elementos de tempo linear e contínuo de sistemas dinâmicos. Exemplos de blocos desse tipo são `integração` e `derivação`.

- **Blocos não Lineares**: são elementos de tempo contínuo não linear de sistemas dinâmicos. Exemplos de blocos desse tipo são matemática elementar (como multiplicação) e operadores lógicos (como `AND`, `OR` e `<=`).

- **Blocos de Conexão**: são usados para organizar e combinar sinais e sistemas. Exemplos de blocos desse tipo são `multiplexadores` e `demultiplexadores`.

A comunicação entre os blocos se dá por meio de linhas que transmitem sinais. A Figura 2.5 mostra um exemplo de um modelo Simulink modelado no ambiente MATLAB/ Simulink. Esse sistema é um loop simples, e os blocos presentes nele têm as seguintes caracterizações:

- O bloco `In1` é um bloco fonte do tipo `iput` (entrada) responsável por receber entradas do sistema;

- O bloco `Out1` é um bloco sumidouro do tipo `output` (saída) responsável por capturar os sinais de saída do sistema;

- Os blocos `Sum1` e `Sum2` são blocos lineares do tipo `sum` (soma); esses blocos podem ter sinais positivos ou negativos associados a cada uma de suas entradas, no caso do exemplo, ambos os blocos possuem um sinal positivo e um negativo, o que significa que o valor de sua saída será o valor que está na porta com sinal positivo menos o valor que estiver na porta com sinal negativo;

- Os blocos `G1` e `G3` são blocos lineares do tipo `gain` (ganho); a saída de um bloco desse tipo é igual à entrada multiplicada por um parâmetro definido pelo desenvolvedor (no caso desse exemplo, `G1` tem um ganho de 3x e o `G3` de 5x);

- O bloco `SW2` é um bloco do tipo `switch`; blocos `switch` têm três portas de entrada, e o seu valor de saída é igual ao valor de sua primeira ou terceira entrada, dependendo de seus parâmetros e do valor da segunda porta. No caso desse sistema, se o valor

na segunda porta for maior ou igual a zero, sua saída será igual ao valor da primeira porta e, caso contrário, será igual ao valor da terceira porta;

- O bloco `Delay` é do tipo `delay` e a sua saída é igual a sua entrada, mas atrasada em um número de unidade definido em seus parâmetros.

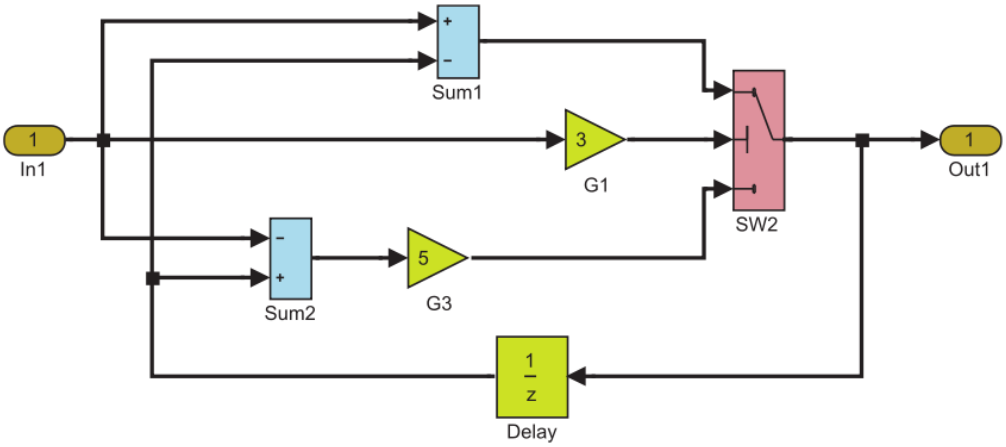


Figura 2.5: Exemplo de um modelo Simulink (MathWorks, 2007).

Um segundo produto da MathWorks usado no desenvolvimento de sistemas embarcados é o Real-Time Workshop, que é acoplado ao MATLAB/Simulink para gerar código, traduzindo modelos para linguagens de projeto de sistemas de software e hardware como C e VHDL. O Real-Time Workshop é uma ferramenta muito popular para projeto de sistemas embarcados por causa da sua flexibilidade, capacidade de rápidas interações e possibilidade de geração de código eficiente para aplicações com restrições de tempo real ou não. Além disso, é possível monitorar e ajustar o código gerado usando blocos Simulink e a análise de capacidade embutida, ou executar e interagir com código fora dos ambientes MATLAB/Simulink (MathWorks, 2007).

A Figura 2.6 mostra o código em C gerado automaticamente pelo Real-Time Workshop para o sistema da Figura 2.5. O código é gerado para uma simulação que ocorrerá em 10 unidades de tempo (valor colocado como condição de parada do loop da estrutura `for`); o `if` do código representa o `switch SW2` do modelo, em caso de a condição ser verdadeira, a saída atual do `SW2` (i.e., a variável `rtb_SW2_c`) terá como valor a lógica implementada pelo bloco `Sum1` (o valor produzido pela entrada da simulação – variável `rtU.In1` – menos o valor de saída do Delay (variável `rtDWork.DELAY_DSTATE`)) e, caso contrário, terá a lógica do bloco `Sum2` (variável `rtDWork.DELAY_DSTATE` menos o valor de entrada da simulação) vezes cinco. Depois disso, o `Delay` (variável `rtDWork.DELAY_DSTATE`) e a saída (variável `rty.Out1`) terão o seu valor igual à saída do SW2.

```
/* Switch: '<Root>/SW2' incorporates:
 *  Sum: '<Root>/Sum1'
 *  Gain: '<Root>/G1'
 *  Sum: '<Root>/Sum2'
 *  Gain: '<Root>/G3'
 */
for(i1=0; i1<10; i1++) {
  if(rtU.In1[i1] * 3.0 >= 0.0) {
    rtb_SW2_c[i1] = rtU.In1[i1] - rtDWork.Delay_DSTATE[i1];
  } else {
    rtb_SW2_c[i1] = (rtDWork.Delay_DSTATE[i1] - rtU.In1[i1]) * 5.0;
  }

  /* Outport: '<Root>/Out1' */
  rtY.Out1[i1] = rtb_SW2_c[i1];

  /* Update for UnitDelay: '<Root>/Delay' */
  rtDWork.Delay_DSTATE[i1] = rtb_SW2_c[i1];
}
```

Figura 2.6: Código gerado automaticamente pelo Real-Time Workshop a partir do sistema da Figura 2.5 (MathWorks, 2007).

Apesar de o MATLAB/Simulink ser uma poderosa ferramenta gráfica para projeto de sistemas, é importante observar que ela não cobre todo o espectro do desenvolvimento de sistemas embarcados. O processo de desenvolvimento de sistemas embarcados inclui diversas outras atividades complexas como, por exemplo, especificação de requisitos, verificação, mapeamento para plataforma distribuída, escalonamento, análise de desempenho e síntese (Balasubramanian et al., 2006).

2.4 Sistemas Embarcados Críticos

Sistema embarcado crítico é um tipo de sistema embarcado em que uma falha ou mau funcionamento pode resultar em:

- Sérios danos físicos a indivíduos ou perda de vidas humanas
- Altos prejuízos por danificação ou destruição de equipamentos
- Danos ao meio ambiente

O projeto de sistemas embarcados críticos costuma ser muito mais complexo do que os não críticos, uma vez que, além da especificação do comportamento funcional do sis-

tema, é necessário um grande esforço para implementar os requisitos de confiabilidade do sistema. Entre esses requisitos se encontram: manutenibilidade, disponibilidade e segurança, no sentido de que nenhum mal será causado pelo sistema (*safety*) e no sentido da confidencialidade e autenticidade nas comunicações (*security*).

Os engenheiros de software embarcado devem conhecer e usar uma combinação de técnicas de prevenção de defeitos mais rica do que engenheiros de outros domínios de software. Segurança e desempenho não podem ser projetados e testados isoladamente. Esses itens têm influência um sobre o outro, assim como os requisitos funcionais e de interface (Ebert e Jones, 2009).

Tendo em vista as falhas de projeto e erros humanos, Kopetz (1997) propôs doze princípios de projeto de sistemas embarcados críticos:

1. Considerações de segurança devem ser utilizadas para conduzir o projeto;
2. Especificações precisas das hipóteses de projeto devem ser feitas desde o começo. Isso inclui as falhas esperadas e a probabilidade de acontecerem;
3. Regiões de confinamento de falhas (RCF) devem ser consideradas. Uma RCF não deve afetar outra;
4. Uma noção consistente de tempo e estado deve ser estabelecida. Caso contrário, será impossível diferenciar os erros originais e os decorrentes dos originais
5. Interfaces bem definidas devem esconder comportamentos internos de componentes;
6. Deve ser garantido que componentes falhem independentemente;
7. Componentes devem se considerar corretos, a menos que dois ou mais componentes tratem o contrário como verdade (principio da auto-confiança);
8. Mecanismos de tolerância a falha devem ser projetados para não criar nenhuma dificuldade adicional ao explicar o comportamento do sistema. O mecanismo deve ser desacoplado da função normal do sistema;
9. O sistema deve ser projetado para diagnose. Por exemplo, deve ser possível identificar erros mascarados existentes;
10. A interface homem-máquina deve ser intuitiva. A segurança do sistema deve ser mantida apesar de erros cometidos por humanos;
11. Toda anomalia deve ser gravada. Essa gravação deve envolver efeitos internos, caso contrário pode acontecer de as anomalias serem mascaradas por mecanismos de tolerância a falha;

12. Uma estratégia de nunca desistir deve ser adotada. Sistemas embarcados podem ter que fornecer serviços ininterruptos. A queda do serviço, por exemplo, é algo inaceitável para esse tipo de sistema.

Exemplos de aplicações típicas de sistemas embarcados críticos são: sistemas aviônicos, usinas nucleares e equipamentos médicos, industriais e militares.

2.4.1 Veículos Aéreos Não Tripulados (VANTs)

VANT é uma expressão que se refere a aeronaves que podem voar sem piloto. Trata-se de uma estrutura de aeronave, um sistema computacional, sensores, GPS, atuadores (*servo-mechanism*) e CPUs. Todos esses elementos combinados devem pilotar uma aeronave sem qualquer intervenção humana. Uma outra definição comum para VANTs é uma aeronave capaz de voar de modo autônomo e operar em diversos tipos de missões e que, em casos de emergência, pode ser controlada de alguma estação-base (Pastor et al., 2006).

Os VANTs foram concebidos inicialmente para missões militares, pelo fato de trazerem soluções seguras e de relativo baixo custo para diversos tipos de missões como, por exemplo, vigilância, espionagem e bloqueio ou interferência em radares. Além das aplicações militares, essas aeronaves estão sendo cada vez mais utilizadas para aplicações civis, científicas e comerciais como medições meteorológicas em altas altitudes, coleta de dados topográficos, avaliação de situações catastróficas e monitoração de agricultura, pesca e meio ambiente.

Os VANTs podem ser classificados por suas diferentes missões. Também é possível classificá-los quantitativamente de acordo com a magnitude de diferentes parâmetros. Entre os parâmetros essenciais envolvidos no voo de um VANT estão: duração de voo, velocidade, altitude de voo, etc. Os itens a seguir fornecem uma breve descrição e mostram variações comuns de cada um desses parâmetros (Bhat et al., 2010):

- Duração (ou autonomia) de voo: esse parâmetro refere-se ao tempo total da decolagem à aterrissagem. Para VANTs de longa duração, os voos podem durar de 2 a 3 horas e para os de curta duração, de 30 minutos a 1 hora.

- Altitude de voo: pode ser definido como a altitude na qual os equipamentos de coleta de dados (e.g. câmeras fotográficas) têm o desempenho desejado para o modo de operação. Tipicamente os VANTs conseguem voar a uma altura de 5 a 6 Km e também a apenas alguns metros do chão.

- Velocidade de voo: a velocidade do veículo depende da potência do motor e de seu projeto aerodinâmico. Velocidades típicas variam de 72 a 540 km/h.

- Ângulo de guinada: esse ângulo depende principalmente da missão e da faixa de cobertura do terreno. Seus valores tipicamente estão entre 40 e 50 graus.

- Alcance: é a distância total coberta pelo veículo da decolagem ao pouso. A distância do alcance varia entre 30 e 1000 kms.

Os VANTs possuem seis módulos principais que trabalham coordenadamente para alcançar autonomia de voo e cumprir os objetivos das missões. Cada módulo é definido a seguir, segundo Pastor et al. (2006).

- **Estrutura de aeronave**: trata-se de uma plataforma estável, simples, leve e aerodinamicamente eficiente com espaço limitado para aviônicos (eletrônicos de aviação);

- **Computador de voo**: é o coração dos VANTs; um sistema computacional que serve para direcionar o voo da aeronave para seu plano de voo. Para isso ele utiliza informações aerodinâmicas coletadas por sensores (como acelerômetros, giroscópios, sensores de pressão, GPS, etc), dados da missão e diversas superfícies de controle presentes na estrutura da aeronave;

- **Equipamentos de coleta de dados**: são um conjunto de sensores composto de câmeras, sensores infravermelhos, sensores térmicos, etc, para coletar informações que podem ser parcialmente processadas a bordo ou transmitidas diretamente para a estação base para serem analisadas futuramente;

- **Controlador de missão**: Um sistema de computador a bordo da aeronave que deve controlar a operação dos sensores existentes nos equipamentos de coleta de dados. Essa operação deve ser executada de acordo com o andamento do plano de voo e com a missão atual atribuída ao VANT;

- **Estação base**: Um sistema de computador em terra projetado para monitorar o andamento da missão e, eventualmente, operar o VANT e os seus equipamentos de coleta de dados;

- **Infraestrutura de comunicação**: Uma mistura de mecanismos de comunicação (modems a rádio, comunicação por satélite, etc) que devem garantir comunicação contínua entre o VANT e a estação-base.

Nas subseções seguintes serão descritos alguns VANTs: dois usados para aplicações civis (ARARA e Tiriba) e um usado para aplicações militares (Predator).

ARARA

O projeto ARARA (Aeronaves de Reconhecimento Assistidas por Rádio e Autônomas) começou em 1999 por iniciativa do Laboratório de Computação de Auto Desempenho (LCAD) do Instituto de Ciências Matemáticas e de Computação da USP São Carlos em parceria com a EMBRAPA-CNPDIA e teve como objetivo o desenvolvimento de um VANT para monitoramento de plantações e reservas ecológicas (Neris, 2001). Entre as possibilidades de monitoramento, a aeronave pode verificar a presença de pragas, espécies invasoras e o andamento do crescimento das plantações.

Figura 2.7: VANT ARARA

A aeronave tem autonomia de voo de até três horas, pode fotografar e filmar em alta resolução áreas rurais e transmitir em tempo real essas informações. Também é possível pré-programar suas missões inserindo dados como altitude, percurso e área a ser sobrevoada e fotografada.

Predator RQ-1

O VANT *Predator RQ-1* é classificado por Agostino et al. (2006) como um VANT de propósitos múltiplos – VANTs que geralmente são modificações de aeronaves de reconhecimento e que possuem armamento. Suas principais missões costumam ser interditar ou conduzir reconhecimento armado contra alvos críticos.

Essas aeronaves também podem atacar com mísseis *AGM-114C/K Hellfire*. Caso não sejam requeridas armas para suas missões, esses VANTs também são capazes de fazer reconhecimento, vigilância e aquisição de alvos.

Figura 2.8: VANT Predator RQ-1 (Airforce, 2010)

Tiriba

Tiriba é um pequeno VANT elétrico usado em missões predefinidas como monitoramento ambiental e de agronegócio. O projeto da aeronave foi feito em parceria pela empresa AGX (AGX, 2012) e o INCT-SEC (INCT-SEC, 2012). Entre as principais características do Tiriba estão: autonomia de voo entre 40 minutos e uma hora e meia, capacidade de carga de 3 kg (podendo ser usado, por exemplo, com câmeras de video e foto), propulsão elétrica (1.2 KW) e velocidade de cruzeiro entre 60 e 100 Km/h.

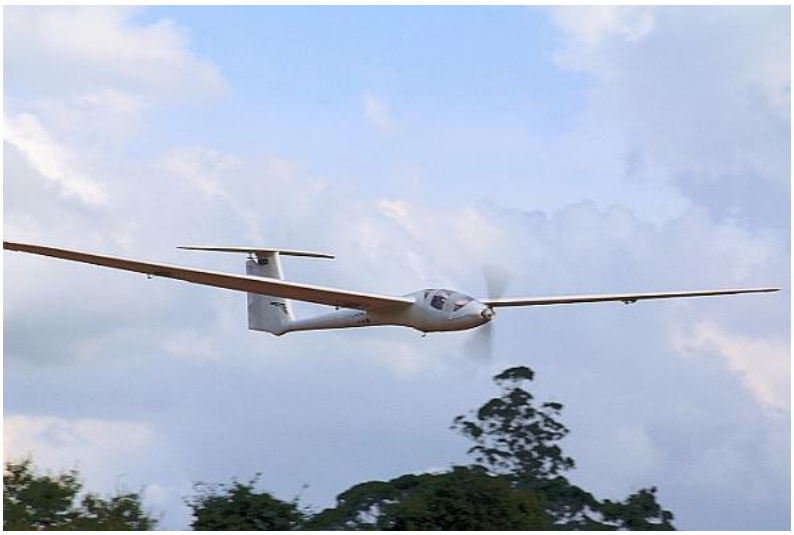


Figura 2.9: VANT Tiriba

O VANT foi desenvolvido seguindo a abordagem de desenvolvimento dirigido por modelo com apoio do ambiente MATLAB/Simulink. O código que implementa o controle e a navegação da aeronave foi todo gerado automaticamente pelo MATLAB/Simulink, contabilizando mais de 30 mil linhas de código C no total.

O modelo Simulink do VANT Tiriba foi usado como estudo de caso neste trabalho. A Figura 2.10 mostra uma visão de alto nível da arquitetura do VANT Tiriba. Nela é possível ver os quatro processadores responsáveis por controlar o vôo e executar a missão da aeronave. Para cada um desses processadores é alocado código gerado automaticamente a partir de diferentes subsistemas escritos em Simulink. Esses subsistemas são:

- Unidade inercial: esse subsistema determina a orientação (posicionamento no espaço) do avião nos três eixos.
- Unidade de pressão(ou barométrica): subsistema responsável por monitorar a velocidade vertical, horizontal e atitude do avião.
- Unidade de navegação: é responsável pelo controle e gerenciamento da missão e da navegação da aeronave. Com base na rota que deve ser feita e na posição em que a aeronave está, deve calcular a direção para a qual a aeronave deve realizar manobras.
- Unidade de controle: esse subsistema recebe comandos de direção da unidade de navegação e deve calcular o melhor parâmetro para os atuadores com o objetivo de colocar a aeronave na direção desejada.

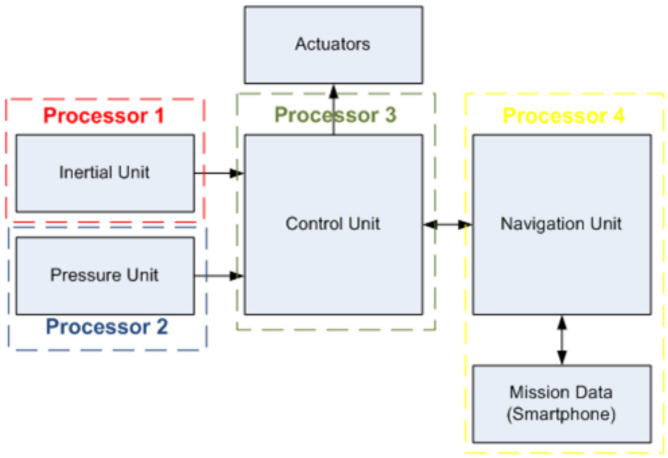


Figura 2.10: Visão alto nível da arquitetura do Tiriba (Branco et al., 2011)

A missão do Tiriba é definida por scrips escritos em uma linguagem específica de domínio desenvolvida pela empresa AGX. Esses *scripts* são compilados para uma linguagem de máquina que é interpretada pelo processador da unidade de navegação da aeronave. As informações da missão definidas nos *scripts* são referentes à rota que o avião deve tomar na sua missão (descrita por meio de uma ordem específica de *waypoints* de coordenadas

geográficas) e intervalo de espaço em que as fotos do avião devem ser tiradas (por exemplo, a cada 500 metros).

A Figura 2.11 mostra a imagem gerada pelo MATLAB/Simulink como resultado da execução de uma simulação do Tiriba. Essa imagem representa um mapa por onde a aeronave passou; nesse mapa estão denotados os *waypoints* que formam a rota da aeronave em vermelho, pontos de fotografia como pontos pretos com contornos em cinza, a rota que a aeronave deveria seguir em azul escuro, as rotas de manobras calculadas pela aeronave em azul claro e a rota que a aeronave efetivamente realizou durante a simulação em amarelo.

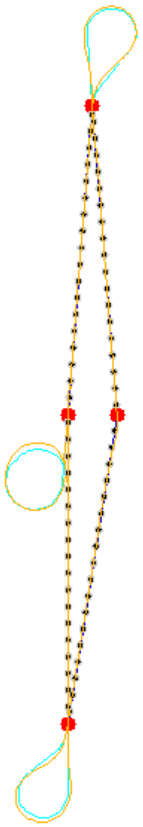


Figura 2.11: Imagem gerada pelo MATLAB/Simulink como resultado da execução de uma simulação do Tiriba

2.5 Considerações Finais

Neste capítulo as principais características e restrições de sistemas embarcados, sistemas embarcados de tempo real e sistemas embarcados críticos são introduzidas. O projeto desses sistemas também foi um tópico explorado no capítulo, e técnicas e ferramentas foram expostas. Os tópicos mais explorados neste capítulo foramo ambiente modelagem e simulação de sistemas MATLAB/Simulink e o domínio de VANTs.

Capítulo

3

Linha de Produtos de Software

3.1 Considerações Iniciais

Os principais conceitos sobre a abordagem de linha de produtos de software (LPS) são introduzidos na Seção 3.2; esses conceitos envolvem os que caracterizam a abordagem, variabilidade em LPS, modelo de *features* e técnicas para implementação de variabilidades. As fases ou atividades de alto nível para o desenvolvimento de LPS, segundo as principais publicações da área, são exploradas na Seção 3.3. Na Seção 3.4 são descritos os métodos de desenvolvimento de LPS FAST, FODA e KobrA. Os modelos de adoção de LPS são discutidos em 3.5 e algumas orientações encontradas na literatura para realizar a adoção e migração para a abordagem de LPS são também discutidas. Na Seção 3.6, as principais características do gerenciamento de variabilidade em LPS são apresentadas e algumas ferramentas utilizadas para tal propósito são descritas. Por fim, na Seção 3.7 alguns trabalhos relacionados e suas contribuições são descritos; esses trabalhos estão no contexto de LPS e veículos autônomos.

3.2 Linha de Produtos de Software: uma Introdução

3.2.1 Conceitos Básicos

Surgida em meados de 1990, LPS é uma abordagem promissora para o desenvolvimento de software, com a qual se desenvolve um conjunto ou família de produtos com foco em um domínio específico. Os produtos de uma mesma família possuem uma arquitetura reusável comum (uma vez que são destinados a um mesmo domínio), e também possuem características variáveis para satisfazer a diferentes requisitos específicos.

A abordagem convencional de desenvolvimento de sistemas que costuma ser comparada com a de LPS é a de sistemas únicos (*single systems*). Essa abordagem, ao contrário da LPS, concentra-se no desenvolvimento de um único produto de software. A diferença no foco das abordagens implica diferentes estratégias de negócio: visão *ad-hoc* de "próximo-contrato" (no caso da abordagem de sistemas únicos) e visão estratégica no campo dos negócios (abordagem de LPS) (Van Der Linden et al., 2007).

No desenvolvimento de uma LPS, os requisitos, projetos e casos de teste devem ser considerados para toda a linha de produto. Como nem todos os requisitos (ou feaures) são igualmente relevantes para cada sistema, as comunalidades e variabilidades dos artefatos da linha de produto precisam ser explicitadas. Artefatos explicitados como comunalidades compõem uma parte da arquitetura que será reusada por todos os produtos da linha de produtos, ao contrário dos explicitados como variabilidades.

Ativo (do inglês *asset*) é o termo utilizado na literatura para denotar artefatos que possam ser reusados em projetos e sistemas (El Kaim, 2000). Um ativo pode ser um documento de requisito, componente, caso de teste, arquitetura, código fonte, etc. O conjunto de ativos de uma linha de produtos (também denominado ativos do núcleo ou *core assets*) pode ser reusado por diferentes membros da LPS.

A abordagem de LPS já está bem consolidada na indústria - existe um grande número de métodos de desenvolvimento e ferramentas bem estabelecidas e evoluídas no contexto da abordagem. De acordo com Pohl et al. (2005), os principais benefícios associados a LPS são: redução no tempo e esforço de desenvolvimento e aumento da qualidade.

Além dessas vantagens, a abordagem também traz impacto positivo em outras áreas, como redução no esforço de manutenção, eficiência técnica, gerenciamento e satisfação dos clientes (Decker e Dager, 2007).

A Figura 3.1 mostra o esforço acumulado no desenvolvimento de sistemas em relação ao número de sistemas a serem desenvolvidos, comparando as abordagens de sistemas únicos e de linha de produtos. No caso da LPS, existe a necessidade de um investimento e esforço inicial relativamente altos para o desenvolvimento de ativos reusáveis, bem como

transformações na organização (Van Der Linden et al., 2007). No entanto, depois do desenvolvimento de um certo número de sistemas, o esforço necessário para desenvolver mais sistemas com a abordagem de LPS passa a ser cada vez menor. De acordo com alguns estudos empíricos, existe um ponto de equilíbrio em torno de três sistemas a partir do qual a abordagem de linha de produtos passa a exigir um esforço acumulado menor do que a abordagem de sistemas únicos (Clements e Northrop, 2001).

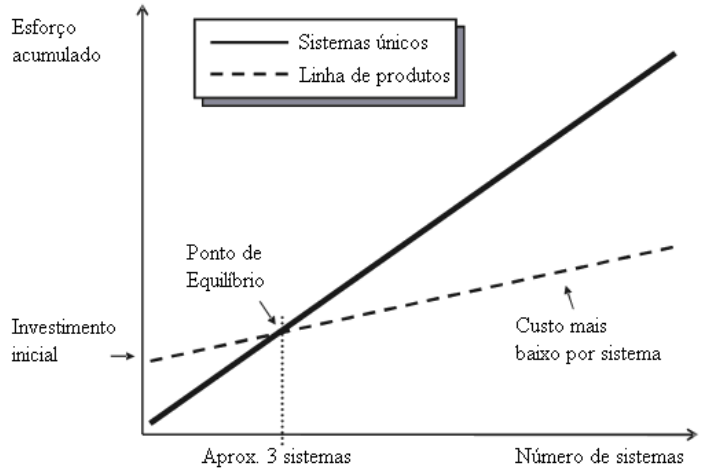


Figura 3.1: Esforço acumulado no desenvolvimento de sistemas com a abordagem de sistemas únicos e LPS (adaptada de Van Der Linden et al. (2007)).

LPS têm sido utilizadas em diversos domínios, como os de software básico (Bonifácio et al., 2009), sistemas de informação (Buhrdorf et al., 2004; Torres et al., 2010) e sistemas embarcados (Bunse et al., 2007; Polzer et al., 2009; Yoshimura et al., 2006). Desses domínios, o que produziu mais aplicações e recebeu mais atenção nos últimos anos, sem dúvida, é o de sistemas embarcados; algumas das empresas que adotam a abordagem no desenvolvimento de produtos com a abordagem de LPS nesse domínio são: *Siemens*, *Nokia* e *Philips* (Van Der Linden et al., 2007).

Pohl et al. (2005) mostram um exemplo de LPS no domínio de sistemas embarcados: uma LPS de automação residencial. Alguns recursos ou funções existentes nesse tipo de aplicação são: lâmpadas, portas, janelas e persianas inteligentes. No caso das portas inteligentes, elas podem, por exemplo, destravar-se por meio de algum mecanismo de identificação como cartões de RFID ou sensores de impressão digital. A variabilidade nesse tipo de aplicação é causada por necessidade dos clientes (que podem preferir menos características por motivos econômicos) e por hardware (pode haver várias opções de hardware para cada componente necessário na aplicação).

Apesar das inúmeras possibilidades de criação de linha de produtos para sistemas embarcados, nenhum trabalho que desenvolveu uma no domínio de veículos aéreos não

tripulados (um tipo de sistema embarcado caracterizado na Seção 2.4.1) foi encontrado na literatura, o que mostra a relevância deste trabalho.

3.2.2 Variabilidade em Linhas de Produtos de Software

Uma linha de produtos pode ser descrita como um conjunto de abstrações chamadas *features* (em português, características). De acordo com Pohl et al. (2005) *features* são definidas por requisitos funcionais e de qualidade do sistema em consideração. Fazendo uso do conceito de *features*, Van Der Linden et al. (2007) definiram três tipos principais de variabilidades para LPS:

- *Comuns*: comunalidade refere-se à *features* que aparecem exatamente na mesma forma em cada um dos produtos da LPS; ou seja, é o conjunto das *features* comuns a todos os produtos da LPS.

- *Variáveis*: são as *features* que são comuns a alguns produtos, mas não a todos eles. Cada *feature* deve ser explicitamente modelada como uma possível variabilidade e deve ser implementada de modo que permita que apenas os produtos selecionados a apresentem.

- *Específica de produto*: uma *feature* que pode ser parte de apenas um produto. Essas especialidades normalmente não são requeridas pelo mercado, mas sim por um cliente específico.

Durante o ciclo de vida de uma LPS, uma variabilidade pode mudar de tipo. Por exemplo, uma *feature* específica de produto pode ser requerida por um segundo produto, o que a faria se tornar uma *feature* variável da LPS.

Descrever as comunalidades e variabilidades por meio de modelagem de *features* é uma prática natural e intuitiva, que facilita a compreensão da LPS tanto do ponto de vista dos desenvolvedores como dos clientes (Kang et al., 2002).

3.2.3 Modelo de Features

Ao se modelar as *features* de uma LPS, é comum fazer-se uso de um modelo de *features* – estrutura que organiza de maneira hierárquica as *features* de um sistema e na qual é possível decompor-se cada *features* em diversas *sub-features*. A Figura 3.2 mostra um modelo de *features* para uma loja virtual (*e-shop*) (Bonifácio e Borba, 2009).

As *features* as de um sistema podem ser classificadas em dois tipos:

- **Obrigatórias**: Fazem, obrigatoriamente parte de todos os produtos da LPS.

- **Opcionais**: Podem ou não fazer parte de um produto da LPS.

Além disso, no modelo de *features* pode haver quatro tipos de relações entre *features*:

- **OU inclusivo**: Qualquer uma das sub-*features* pode fazer parte do produto.
- **OU exclusivo**: Apenas uma das sub-*features* pode fazer parte do produto.
- **Dependência**: Uma *feature* que possua uma relação de dependência com outra *feature* só poderá ser selecionada se a *feature* da qual ela "depende" for selecionada.
- **Hierarquia**: Uma *feature* que possua essa relação com uma outra *feature* automaticamente tornará ela uma *sub-feature* sua. Essa relação organiza e agrega semântica ao modelo. No que diz respeito à instância de produtos, uma *sub-feature* só poderá ser selecionada se sua *feature* "mãe" também for.

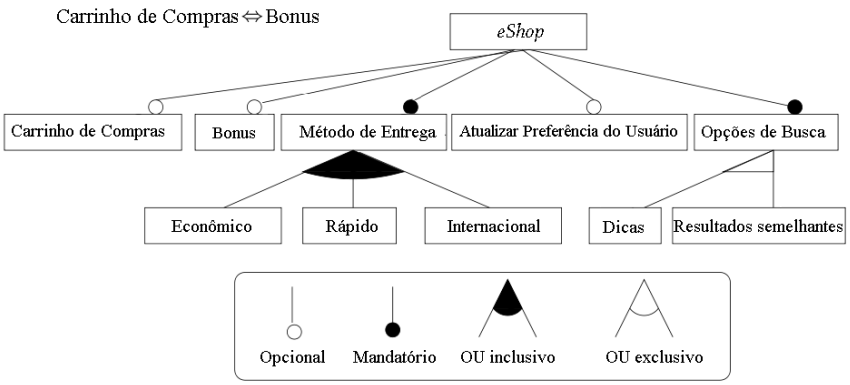


Figura 3.2: Modelo de *features* de uma loja virtual (adaptada de Bonifácio e Borba (2009)).

Na Figura 3.2 é possível ver os dois tipos de *features* e os quatro tipos de relações. Também é possível observar que há uma relação de dependência mútua entre as *features* *Carrinho de Compras* e *Bônus* (i.e. a *feature* *Carrinho de Compras* possui uma relação de dependência com a *feature* *Bônus* e vice versa).

Por ser representado de maneira visual e hierárquica, os modelos de *features* facilitam a compreensão da importância, dependência e função de cada uma das *features* e também oferecem uma visão geral clara sobre a arquitetura do sistema.

3.2.4 Técnicas para Implementação de Variabilidades

Existem diversas técnicas e métodos com os quais é possível implementar e configurar a variabilidade de *features* nos ativos de linhas de produtos. Além da implementação e configuração de variabilidades, esses métodos e técnicas visam a minimizar duplicação de código e o esforço de reúso e manutenção (Muthig e Patzke, 2009).

Compilação Condicional

Compilação condicional permite controle sobre segmentos de código a serem incluídos ou excluídos da compilação de um programa. Essa técnica separa variabilidade em diferentes seções de código que são selecionadas por meio de *flags* de compilação. As funcionalidades comuns são postas fora das áreas de compilação condicional (sendo sempre selecionadas), e ao estabelecer valores adequados para as *flags* de compilação, seções de código referentes à variabilidade desejada para uma determinada instância de produto serão incluídas (Myllymäki, 2002).

Apesar de ser uma solução simples, a compilação condicional deve ser expressa no código fonte, o que, dependendo da quantidade de expressões de compilação condicional, pode poluir visualmente o código, dificultando a compreensão e a sua manutenção (Kästner et al., 2008; Spencer e Collyer, 1992).

A Figura 3.3 mostra um exemplo de compilação condicional. No caso deste trecho de código, está sendo implementado um método que retorna o nome do sistema operacional; Se a *flag* de compilação WIN32 for definida como verdadeira, o trecho de código compilado será o $s =$ *"Windows"*, e caso a VXWORKS seja verdadeira, o trecho compilado será $s =$ *"VXWORKS"*.

```
string getOperatingSystemName(){

        string s;

#ifdef WIN32
        s = "Windows";
#ifdef VXWORKS
        s = "VXWORKS";
#endif

        return s;
};
```

Figura 3.3: Exemplo de código com compilação condicional com anotações '*#ifdef*' (Myllymäki, 2002).

Aspectos

Programação orientada a aspectos (POA) é uma técnica de desenvolvimento de software que proporciona meios para modularizar interesses transversais. Interesses são conjuntos particulares de comportamento de um software; eles podem ser gerais como interação com banco de dados ou específicos como execução de determinados cálculos. O propósito do

paradigma é separar funções secundárias ou de suporte da lógica de negócio principal do programa.

Interesses referem-se a características funcionais e não funcionais de sistemas de informação. Exemplos de interesses funcionais são: interesses relacionados a dados, características ou regras de negócio; exemplos de interesses não funcionais são sincronização, distribuição, tratamento de erros e gerenciamento de transação.

Na POA, os interesses são implementados como aspectos em uma única área física e depois são compostos no código automaticamente, geralmente com apoio de alguma ferramenta (Muthig e Patzke, 2009). Com essa modularização de interesses, é possível alcançar um baixo acoplamento e alta coesão no software em desenvolvimento.

Os interesses podem ser classificados em transversais ou não:

- Interesses transversais entrecortam o código fonte em pontos de diferentes módulos. Um *log* é um bom exemplo de interesse transversal; *log* é um histórico das operações realizadas e erros ocorridos em um sistema, e normalmente ele é necessário em diferentes módulos de um mesmo sistema.

- Interesses não-transversais são interesses de apenas uma parte bem definida ou um módulo do sistema. No contexto da POA, eles podem ser implementados tanto como aspectos, como classes ou métodos.

A técnica de POA é uma boa alternativa para o desenvolvimento de LPS, pois ela permite expressar e modularizar explicitamente variabilidade em modelos, código e gabarito; além disso, a técnica possui poderosos mecanismos de composição de código que podem ser utilizadas na geração de instâncias de produtos.

Variabilidade Positiva

Variabilidade positiva é uma técnica de derivação de produtos em linhas de produtos. A geração de produto começa com um conjunto mínimo de ativos (comum a todos os produtos) e seleciona partes adicionais para serem agregadas ao produto conforme a presença ou ausência de *features* nos modelos de configuração (Groher e Voelter, 2007).

Variabilidade Negativa

Variabilidade negativa também é uma outra técnica de derivação de instâncias de produtos de LPS que consiste na criação de um modelo ou código fonte que contenha a união de todas as *features* da LPS, e a partir do qual é possível gerar instâncias de produtos com todas as configurações válidas. Ao se derivar uma instância específica, o modelo ou código fonte principal é modificado removendo ou desativando as *features* que não foram selecionadas para o produto (Groher e Voelter, 2007).

3.3 Desenvolvimento de Linhas de Produto de Software

Há um consenso na comunidade de engenharia de software de que o desenvolvimento de uma LPS pode ser dividido em duas atividades principais: Engenharia de domínio e Engenharia de aplicação (Northrop et al., 2007; Pohl et al., 2005; Van Der Linden et al., 2007). Nas duas próximas subseções, as duas atividades serão descritas de acordo com o framework apresentado por Pohl et al. (2005).

3.3.1 Engenharia de Domínio

Pohl e Metzger (2006) definem a engenharia de domínio como "*processo responsável pela definição da comunalidade e variabilidade da LPS e, portanto, por estabelecer os artefatos reusáveis. A engenharia de domínio deve garantir que a variabilidade disponível seja apropriada para produzir as aplicações da LPS; para isso deve fazer uso de mecanismos para compor variabilidade nos respectivos artefatos de desenvolvimento (e.g., por meio do projeto de componentes configuráveis)*".

De acordo com Pohl et al. (2005), os principais objetivos dessa atividade são:

- Definir e refinar a comunalidade e variabilidade da LPS.
- Definir o conjunto de aplicações para os quais a LPS é planejada (i.e. definir o escopo da linha de produtos).
- Definir e construir artefatos reusáveis que completam a variabilidade desejada.

A Figura 3.4 dá uma visão geral da engenharia de domínio e mostra os artefatos que compõem a plataforma da LPS (requisitos, arquitetura, componentes e testes). Além desses artefatos, de acordo com Pohl et al. (2005), outros artefatos desenvolvidos na engenharia de domínio são roteiro de produtos (*product roadmap*), que é um plano para o desenvolvimento e introdução de produtos no mercado e que também descreve e categoriza as *features* das aplicações da lista de produtos, e o modelo de variabilidade do domínio que define a variabilidade da LPS (introduzindo pontos de variação para a LPS).

3.3.2 Engenharia de Aplicação

Pohl e Metzger (2006) definem a engenharia de aplicação como "*o processo responsável pela derivação das aplicações da LPS a partir dos artefatos reusáveis. A engenharia de aplicação explora a variabilidade dos artefatos reusáveis ao interligar as variabilidades de acordo com as necessidades específicas das aplicações*".

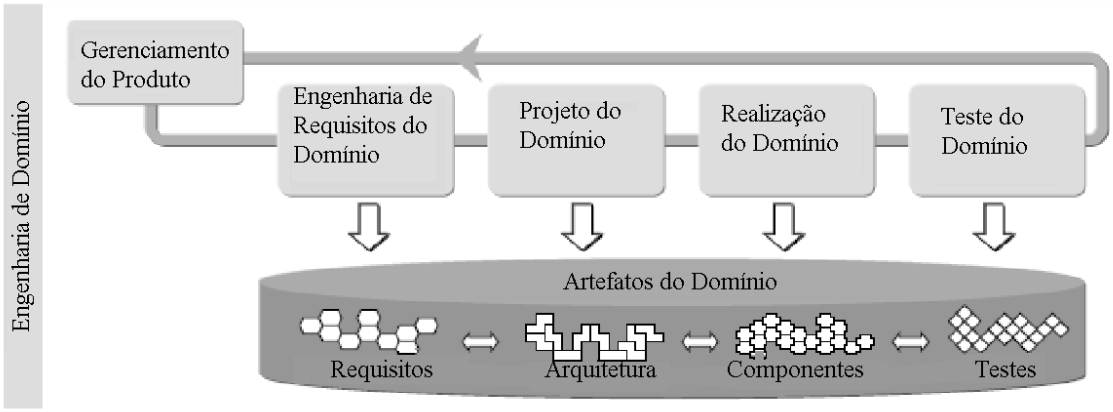


Figura 3.4: Atividades da engenharia de domínio (adaptada de Pohl et al. (2005)).

Em outras palavras, a engenharia de aplicação escolhe e usa os ativos da LPS para conceber as diferentes aplicações previstas para a LPS. Para a geração das diferentes aplicações, os engenheiros de aplicação combinam os artefatos reusáveis desenvolvidos na engenharia de domínio com artefatos específicos de aplicação desenvolvidos por eles mesmos.

A Figura 3.5 mostra uma visão geral do processo de engenharia de aplicação. Os artefatos produzidos na engenharia de aplicação compreendem todos os artefatos desenvolvidos para uma aplicação específica, incluindo as próprias aplicações configuradas e testadas (Pohl e Metzger, 2006).

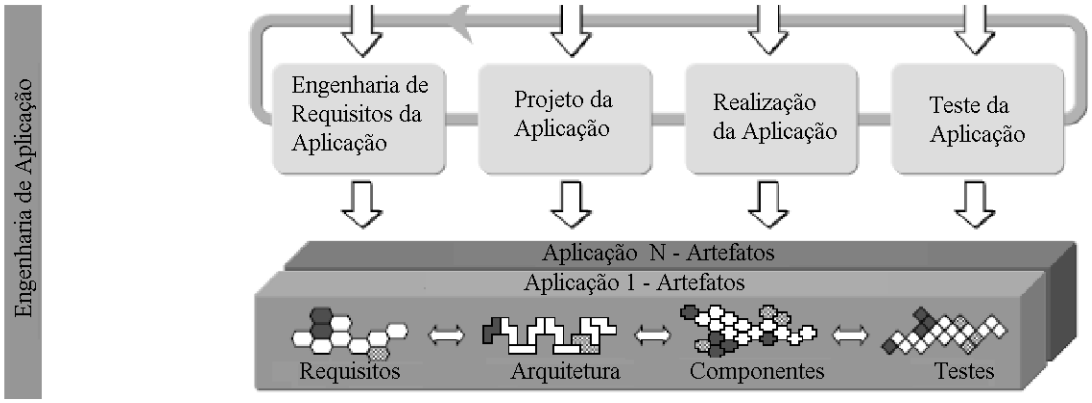


Figura 3.5: Atividades da engenharia de aplicação (adaptada de Pohl et al. (2005)).

3.3.3 Framework for Software Product Line Practice

Uma estratégia bem estabelecida e evoluída para o desenvolvimento de LPS é o *Framework for Software Product Line Practice* (FSPLP) criada pelo SEI (*Software Engineering Institute*). Northrop et al. (2007) apresentam o FSPLP na forma de um documento Web,

com o intuito de guiar a comunidade de software em empreendimentos com LPS. Cada versão representa um esforço incremental para capturar as últimas informações sobre o sucesso de práticas de LPS.

Essa estratégia possui três atividades consideradas essenciais: **Desenvolvimento do ativo do núcleo**, **desenvolvimento do produto** e **gerenciamento**.

De acordo com Northrop et al. (2007) e também com Catal (2009), a atividade de desenvolvimento do ativo do núcleo é equivalente à atividade de engenharia de domínio e a de desenvolvimento do produto, à de engenharia de aplicação. A atividade de gerenciamento compreende duas sub-atividades: o gerenciamento organizacional, que identifica restrições e as estratégias de produção e o gerenciamento técnico, que supervisiona o desenvolvimento, garantindo que ele esteja de acordo com as atividades requeridas, além de decidir o método de produção e fornecer os elementos do gerenciamento de projeto do plano de produção.

3.4 Métodos para Desenvolvimento de Linha de Produtos

Os métodos existentes para desenvolvimento de LPS variam no grau de customização e no nível de abstração usados para as *features* variáveis e comuns de uma família de software (Atkinson et al., 2002). Há propostas de diferentes métodos na literatura, mas como nenhum deles foi utilizado no trabalho, nessa seção descreveremos superficialmente apenas três deles.

FAST (*Family Oriented Abstraction, Specification and Translation*) aplica os princípios de arquitetura de LPS em um processo de engenharia de software. O desenvolvimento de LPS com FAST foi criado para conseguir alcançar dois objetivos: produção rápida e desenvolvimento cuidadoso (Harsu, 2002).

A Figura 3.6 mostra uma visão geral das atividades do FAST. Ele pode ser dividido em três subprocessos:

- **Qualificação de domínio**: Neste subprocesso, a família de software é analisada do ponto de vista econômico. São estimados número, valor e custo de produção dos diferentes membros da família e assim escolhem-se aqueles em que vale a pena receber investimento.

- **Engenharia de domínio**: A engenharia de domínio estuda como os membros da mesma família compartilham a base comum e como diferem uns dos outros. Na metodologia FAST, a engenharia de domínio é dividida em duas partes: **análise de domínio**, que tem como objetivo gerar um documento chamado modelo de domínio

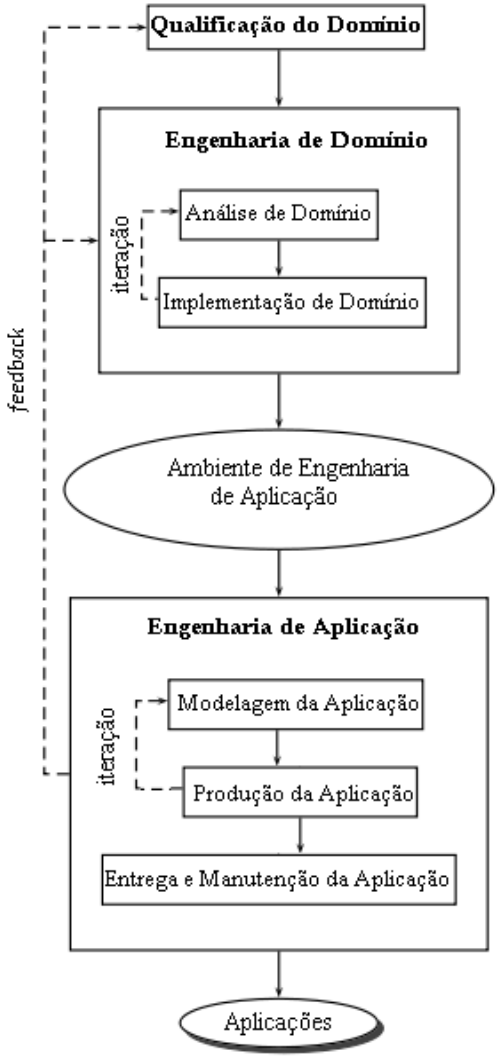


Figura 3.6: Visão geral do método FAST (adaptada de (Harsu, 2002))

(uma representação precisa para especificação do sistema e conceitos de implementação) e **implementação de domínio**, que desenvolve ou refina um ambiente que satisfaça o modelo de domínio. O ambiente desenvolvido na implementação de domínio deve apoiar o sub-processo de engenharia de aplicação.

- **Engenharia de aplicação**: O objetivo deste subprocesso é gerar um conjunto de código e documentação entregáveis, a partir do ambiente de engenharia de aplicação. Os engenheiros de aplicação modelam membros da família de produto na atividade de **modelo de aplicação** e os desenvolvem na atividade de **produção de aplicação**.

FODA (*Feature Oriented Domain Analysis*) é o método de análise de domínio que introduziu a modelagem de *features* na engenharia de domínio. O método tem três fases principais: 1) *Análise de contexto*, que define os limites do domínio para análise; 2)

modelagem do domínio, que descreve os problemas do domínio que serão tratados pelo software; 3) *modelagem da arquitetura*, que cria as arquiteturas de software que implementam a solução dos problemas do domínio (Kang et al., 1990).

Um outro método que é mais recente mas que tem sido muito aplicado é o KobrA (*Komponentenbasierte Anwendungsentwicklung*, em português "desenvolvimento de aplicação baseado em componentes"). Esse método foi criado no contexto da engenharia de software moderna, uma vez que inclui diversas tecnologias avançadas como frameworks, inspeções centradas na arquitetura, modelagem da qualidade, modelagem de processos e desenvolvimento de software baseado em componentes. O principal foco do método KobrA é a criação de uma sinergia entre as abordagens de desenvolvimento de software baseado em componentes e LPS (Atkinson et al., 2002).

3.5 Adoção de Linha de Produto de Software

Apesar das vantagens existentes no desenvolvimento com LPS, a adoção dessa abordagem não é trivial; é uma tarefa complexa que tem altos riscos, barreiras e desafios.

Para introduzir com sucesso a abordagem de LPS, é necessário haver mudanças na organização que pretende fazer isso, bem como no seu processo de desenvolvimento. Isso implica um sobrecusto não apenas econômico, mas também de tempo. Essa necessidade de mudança está entre as principais razões que impedem a adoção ou migração rápida para a abordagem de LPS (Catal, 2009). Um outro problema enfrentado pelas organizações é a falta de especialistas em LPS e o alto custo de treinamento técnico (que também contribui para o pequeno número de especialistas em LPS). Uma dificuldade adicional é que apenas o treinamento não é o suficiente para garantir o sucesso da iniciativa de LPS, pois a prática é também muito importante.

Outro obstáculo para a adoção de LPS mencionada por Yoshimura et al. (2006) é referente à dificuldade de estimar os benefícios econômicos resultantes da migração: a abordagem de LPS claramente traz benefícios em termos de esforço e tempo de desenvolvimento, mas não há modo confiável de se determinar se esses benefícios compensam os gastos envolvidos na migração para a abordagem.

3.5.1 Modelos de Adoção

Há uma variedade de modelos de adoção de LPS e cada um deles possui diferentes *features* e estratégias. Essa variedade faz com que organizações possam selecionar uma ou mais estratégias que melhor se encaixem nos seus objetivos de negócio, realidades de desenvolvimento e estilos de gerenciamento. Os três modelos de adoção principais são: proativo,

reativo e extrativo; cada um deles é descrito nas próximas subseções de acordo com as definições de Krueger (2002b).

Modelo Proativo

O modelo proativo é parecido com a abordagem cascata no desenvolvimento convencional de software. Todas as variações de produtos imaginadas para a LPS são totalmente analisadas, arquitetadas e projetadas antes de implementar a LPS.

Esse modelo pode ser usado quando os requisitos da LPS são bem definidos e estáveis. O modelo requer mais esforço, tempo e custo inicial que os outros modelos; para organizações que estão em uma realidade na qual esses sobrecustos são proibitivos, é preferível adotar o modelo reativo.

O domínio principal de aplicação do modelo proativo é o de sistemas embarcados, nos quais ele tem sido usado para desenvolvimento de LPS de sistemas com altos volumes de vendas como, por exemplo, de impressoras, televisões e eletrônicos de consumo etc.

Modelo Extrativo

O modelo extrativo costuma ser utilizado para migrar um conjunto de sistemas existentes para a abordagem de LPS.

O modelo é mais adequado para casos em que os sistemas existentes possuam uma quantidade significativa de comunalidade, bem como diferenças consideráveis. De acordo com Krueger (2002a), não é necessário executar a extração de todos os sistemas existentes de uma vez. Por exemplo, pode ser mais interessante extrair inicialmente um subconjunto dos sistemas que dão mais lucro e deixar o restante para ser incrementalmente extraído quando necessário.

Para ser uma escolha efetiva, o modelo extrativo requer tecnologias simples de LPS e técnicas que permitem reusar softwares existentes sem um grande esforço de reengenharia. Esse modelo exige um esforço e tempo relativamente pequenos na transição do desenvolvimento convencional de software para LPS.

Modelo Reativo

Esse modelo é semelhante às abordagens de desenvolvimento espiral ou XP (programação extrema). As variações de um ou mais produtos da LPS são analisadas, arquitetadas, projetadas e implementadas em cada ciclo da espiral, fazendo com que a LPS seja desenvolvida em passos incrementais;

O modelo funciona em situações em que não se podem prever os requisitos das variações dos produtos com antecedência, ou em situações nas quais as organizações devem manter

um cronograma agressivo de produção com pouco recurso adicional durante a transição para a abordagem da LPS. A customização da LPS cresce à medida que novos produtos ou novos requisitos de produtos existentes aparecem. O código fonte comum e variável, juntamente com a declaração de características e definições de produtos são estendidos incrementalmente em reação a novos requisitos (Krueger, 2002a).

Uma possibilidade de adicionar um novo produto a uma LPS seguindo o modelo reativo pode ser feito por meio das seguintes tarefas de alto nível:

1. Caracterizar os requisitos para o novo produto em relação ao que é mantido aos ativos da linha de produção atual;

2. É possível que o novo produto esteja dentro do escopo da linha de produção atual. Se for o caso, pular para o passo 4;

3. Se o novo produto não estiver no escopo, adicionar declarações, autômatos, software comum e definições da linha de produção de modo a estender o escopo para incluir os novos requisitos;

4. Criar a definição do novo produto selecionando valores para cada um dos parâmetros das *features*.

De acordo com Faust e Verhoef (2003), os modelos reativo e proativo são extremos e, na prática, o modelo de migração adotado costuma ser uma mistura das duas. Buhrdorf et al. (2004) discutem a experiência da *Salion* (uma companhia de software empresarial que provê soluções em gerenciamento de receita para fornecedores automotivos) na adoção do modelo reativo de LPS. A transição para a abordagem de LPS foi feita com um esforço de duas pessoas/mês, o equivalente a 1% do esforço necessário para desenvolver a versão inicial do software, e é duas ordens de magnitude menor do que o esforço relatado nos esforços de transição com o modelo proativo.

3.5.2 Orientações para Adoção de Linha de Produtos de Software

Diversos trabalhos foram desenvolvidos nos últimos anos propondo práticas e orientações para adoção de LPS (Breivold et al., 2008; Krueger, 2002a,b; Schmid, 2004). A maior parte dos trabalhos propõe orientações mais genéricas para a adoção de LPS e não orientações no escopo de um modelo específico de adoção.

Há uma série de fatores envolvidos na migração para a abordagem de LPS e, conforme exposto no começo da Seção 3.5, riscos de vários tipos podem aparecer no processo de migração. Desse modo, o gerenciamento cuidadoso e abrangente de risco é essencial para garantir o sucesso da migração. Uma outra recomendação que ajuda a diminuir os riscos

é a execução da migração por meio de transições incrementais. A maior vantagem dessa prática é que, ao segui-la, não há um rompimento brusco dos projetos em andamento, o que reduz impactos econômicos negativos nas organizações que estão em processo de migração para LPS (Breivold et al., 2008).

Uma dúvida comum das organizações que estão nos primeiros passos da adoção de LPS é saber quando cada variabilidade deve ser implementada. As orientações para essa dúvida, oferecidas por Schmid (2004), sugerem não implementar apenas as variabilidades que tiverem uma probabilidade baixa de serem requeridas e implementar todas as outras. Seguindo essa orientação, alcança-se um menor custo e esforço de desenvolvimento.

O desenvolvimento de uma arquitetura de LPS que modularize as variabilidades tem um papel-chave na diminuição do custo total de desenvolvimento. Se um ou vários sistemas existentes possuírem uma boa representação, projeto e implementação de arquitetura, então eles podem ser usados com quase nenhum esforço para criar a versão inicial dos ativos do núcleo para a LPS.

Uma outra recomendação é a utilização de tecnologias, ferramentas e técnicas simples focadas especificamente em apoiar o desenvolvimento da LPS (Krueger, 2002b). Essas técnicas e tecnologias minimizam a mudança de paradigma entre o desenvolvimento convencional de software e o desenvolvimento da LPS, além de reduzir o tempo e esforço inicial de transição. Isso acontece por meio do reúso de software, ferramentas, pessoal e processos já existentes. Outras recomendações em relação às ferramentas e tecnologias são: utilizar ferramenta para resolver análises de dependência, utilizar a documentação da arquitetura para melhorar sua integridade e consistência e definir cuidadosamente os pontos de variação e mecanismos de realização.

3.6 Gerenciamento de Variabilidade

No desenvolvimento de sistemas únicos de software, o gerenciamento de variabilidade (muitas vezes referido como gerenciamento de configuração) lida com a variação do software ao longo do tempo. No escopo das LPS, o gerenciamento de variabilidade é uma atividade muito mais complexa, uma vez que ela é multi-dimensional: lida com variabilidade em tempo e espaço (Krueger, 2002c).

O gerenciamento de variabilidade é a atividade responsável por definir, representar, explorar, implementar e evoluir as variabilidades da LPS. Segundo Pohl et al. (2005) essa atividade engloba as seguintes sub-atividades:

- Gerenciar artefatos variáveis.
- Apoiar atividades que estão relacionadas à definição de variabilidades.

- Apoiar atividades focadas em resolver variabilidade.
- Coletar, armazenar e gerenciar informações de rastreabilidade necessárias para cumprir essas tarefas.

Segundo Krueger (2002c), elevar o gerenciamento de variabilidade a um papel central no desenvolvimento de LPS oferece diversas vantagens, entre as quais:

- *Volta ao desenvolvimento de sistemas únicos*: Artefatos comuns, variáveis e de infraestrutura em uma linha de produto podem ser tratados como uma linha de produção de software único que evolui ao longo do tempo;
- *Diminuição da barreira da adoção*: As estratégias incrementais de adoção e o reúso de tecnologias e ativos reduzem tempo, custo e esforço requeridos para adotar a abordagem de linha de produtos;
- *Metodologia de linha de produtos flexível*: É possível adotar métodos diferentes para diferentes condições de negócio. Isso inclui o reúso de ativos legados versus reengenharia, engenharia de variação proativa versus reativa e refatoração de comunalidade e variação.
- *Opções de formalidade*: Engenheiros podem escolher níveis apropriados de formalidade, incluindo realizar ou não: análise formal de domínio, arquitetura formal e "componentização".

3.6.1 Ferramentas para Apoiar o Desenvolvimento de Linha de Produtos de Software

Conforme dito na Seção 3.5.2, as ferramentas para apoiar o desenvolvimento de LPS (e realizar o gerenciamento de variabilidade) possuem uma grande importância na adoção da abordagem.

De acordo com Gomaa e Shin (2004), para uma ferramenta proporcionar um gerenciamento de variabilidade e derivação de produtos efetivos em LPS ela precisa possuir as seguintes propriedades: **múltiplas visões da linha de produto**, **modelo de features**, **meta-modelo** (contém meta-classes da linha de produto e seus relacionamentos), **repositório da linha de produto** (onde os artefatos são armazenados), **verificação de consistência** e **derivação da linha de produto**.

Diversas ferramentas foram propostas nos últimos anos para apoiar o desenvolvimento de LPS. Dentre as ferramentas comerciais bem estabelecidas no mercado, encontram-se: Pure::variants – desenvolvida pela PureSystems, GmbH (Beuche, 2003, 2008), e GEARS – desenvolvida pela BigLever (Krueger, 2002a).

Além das ferramentas comerciais, existe um número grande de ferramentas para LPS desenvolvida no contexto acadêmico: CIDE – desenvolvida pela Universidade de Magdeburg (Alemanha) (Kästner et al., 2008), GenArch – desenvolvida pelo LES da PUC-RIO (Cirilo et al., 2007), Captor-AO – desenvolvida pelo ICMC/USP (Júnior et al., 2008), e Hepheastus desenvolvidas pelo CIn da UFPE (Bonifácio et al., 2009) .

Nas próximas subseções, as ferramentas Pure::variants, CIDE e Hephaestus serão discutidas com mais detalhes.

Pure::variants

Pure::variants é uma ferramenta baseada em modelos de derivação de LPS que considera dois modelos principais: modelo de *features* e modelo de família.

- ***Modelo de features***: A Pure::variants faz uso do modelo de *features* para representar variabilidades e comunalidades dos produtos, assim como as demais ferramentas de apoio ao desenvolvimento de LPS. No entanto,a ferramenta possui uma extensão do conceito de modelo de *features* que acrescenta expressividade ao modelo. Ele apoia hierarquias de modelos, o que permite diferentes representações dos problemas dependendo do tipo de usuário (cliente, desenvolvedor, vendas, etc).

- ***Modelo de família:*** Descreve uma implementação específica de uma LPS, por meio da descrição da estrutura interna de componentes e suas dependências das *features*. Esse modelo é estruturado em vários níveis; o mais alto é formado pelos componentes e o mais baixo pelo código fonte (conforme pode ser visto na Figura 3.7). Cada um dos componentes representa uma (ou mais) *feature* funcional e consiste em uma parte lógica do software (ativos reusáveis).

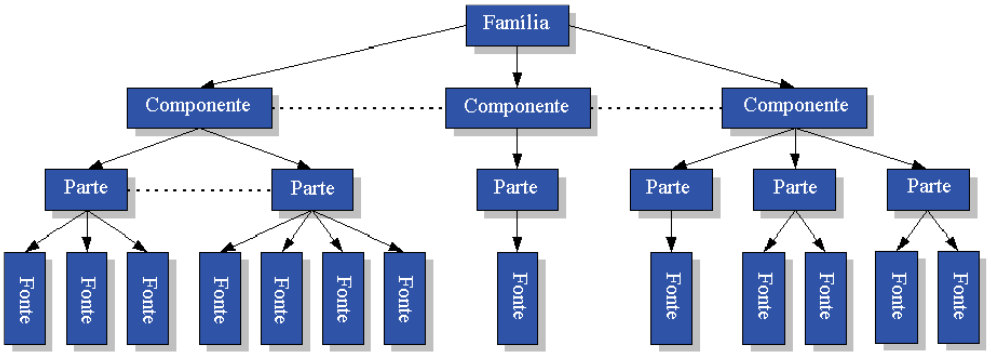


Figura 3.7: Modelo de família da Pure::variants (adaptada de Beuche (2003)).

Os modelos de *features* e modelos de família são desenvolvidos de modo separado e independente pelas tecnologias Pure::variants. Com isso, o reúso de soluções e modelos se torna simplificado.

A Figura 3.8 apresenta uma visão geral do processo básico da criação de variantes de produtos com o Pure::variants. O usuário seleciona as *features* do modelo de *features* e o Pure::variants checa se a seleção é válida ou não (se possível, resolvendo conflitos de dependência automaticamente). A vantagem disso é que mesmo dependências estruturais complexas podem ser eficientemente transformadas em sistemas válidos.

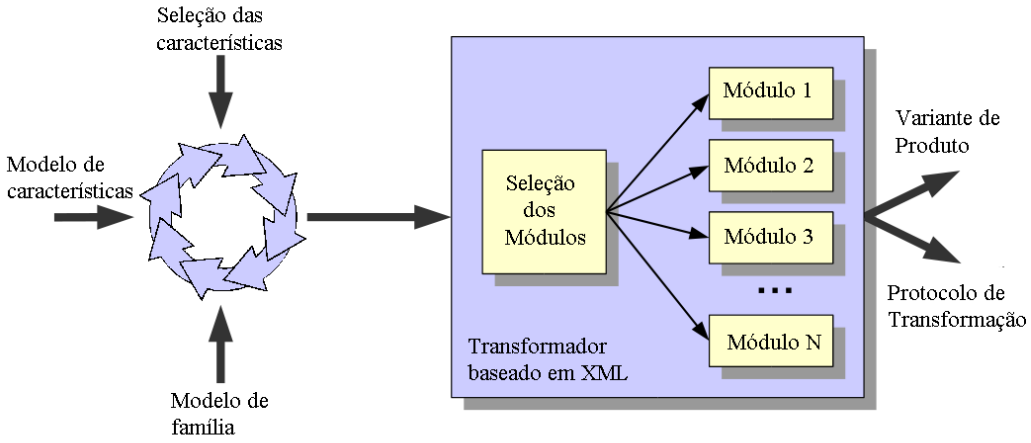


Figura 3.8: Visão geral do processo básico de criação de variantes com o Pure::variants (adaptada de Beuche (2003)).

Baseando-se na definição da configuração do produto e fazendo uso da informação contida no modelo de família, a seleção das *features* é analisada para cada componente e seus elementos lógicos e físicos. O produto final é criado por um processo de transformação controlado pela descrição de componente. Durante o processo, todos os módulos de transformação necessários são ativados e executam as conversões especificadas na descrição de componente.

Alguns pontos positivos do Pure::variants, segundo Chong (2008), são:

- Possui um nível moderado de documentação (guia de usuário), mas o suporte técnico da ferramenta possui alta disponibilidade e resposta rápida; também há treinamento disponível.
- Existem muitos clientes europeus com histórias de sucesso de uso da ferramenta.
- Pode rodar nas plataformas *Windows* e *Linux*.
- Usa "conectores" para apoiar um conjunto rico de aplicações de programas de desenvolvimento como *DOOR*, *CVS*, *SAP*, *ClearCase* e *MATLAB/Simulink*.

Pontos negativos do Pure::variants, segundo Chong (2008), são:

- Requer um esforço médio para configurar e começar a usar a ferramenta.
- O usuário deve configurar o sistema *Java Eclipse* primeiramente, antes de instalar o Pure::variants como *plug-in* Java. Seria útil se tutoriais e demonstrações online sobre este assunto fossem disponibilizadas.
- Uma abordagem de modelagem gráfica (como modelagem operacional, modelagem de objeto/componente) permitiria que os usuários visualizassem mais facilmente as condições e status do projeto da LPS.

CIDE

CIDE (*Colored Integrated Development Environment*) (Kästner et al., 2008) é uma ferramenta para apoiar o gerenciamento de variabilidade em LPS que tem como objetivo principal decompor código legado em *features* para o desenvolvimento da LPS.

Ao contrário dos pré-processadores tradicionais como os das linguagens *C/C++*, com *CIDE*, o código fonte não é marcado com anotações adicionais do tipo '*#ifdef*'. Em vez disso, uma camada de representação do editor indica *features* associadas a diferentes cores de fundo. Os fragmentos de código referentes a alguma *features* só serão selecionados se eles possuírem a "cor" (i.e. associação) de alguma *feature* selecionada na configuração.

Caso um fragmento de código seja associado a mais de uma *feature*, a cor desse fragmento será uma mistura das cores de suas *features*; por exemplo, um fragmento de código referente a uma *feature* de cor azul e uma de cor vermelho teria roxo como cor de fundo. Essa funcionalidade da ferramenta é interessante em casos em que há pouca sobreposição de *feature* no código fonte. Caso contrário, muitas misturas de cores diferentes passam a ser observadas na ferramenta, o que atrapalha a legibilidade do código.

Uma funcionalidade da ferramenta desenvolvida para melhorar a compreensão do código é a de se esconder código de uma determinada *feature* - desse modo, todo o código restante da aplicação pode ser visualizado isoladamente. A Figura 3.9 mostra uma captura de tela do ambiente CIDE em que é possível ver as principais funções da ferramenta.

Para LPS com granularidade fina, abordagens anotativas (como compilação condicional, discutida na Seção 3.2.4) implicam anotações confusas no código fonte e a implementação da LPS tende a se tornar complicada, ilegível e de difícil manutenção (Kästner et al., 2008; Spencer e Collyer, 1992). CIDE é uma ótima ferramenta para trabalhar com LPS com granularidade fina, pois seus mecanismos para separação e definição de *feature* aumentam a organização, legibilidade e compreensão do código.

LPS com granularidade fina costumam ser típicas no modelo de adoção extrativo (apresentado com os outros modelos na subSeção 3.5.1). Isso ocorre porque a LPS é desenvolvida a partir de uma ou mais aplicações legadas; as *features* não são planejadas

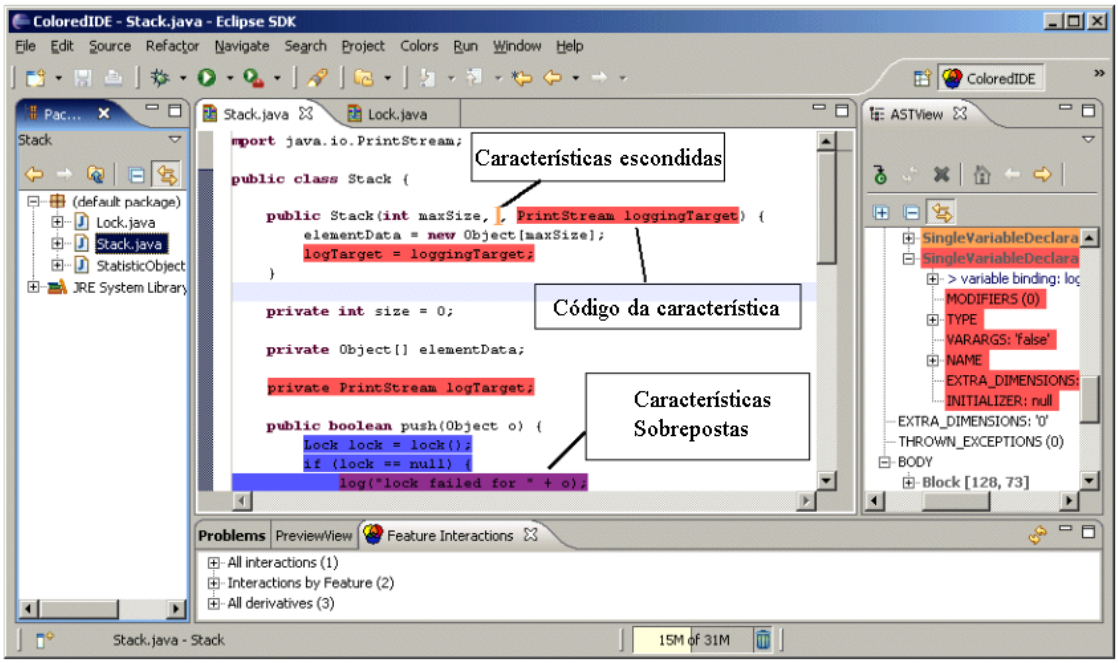


Figura 3.9: Captura de tela do ambiente CIDE (adaptada de (Kästner et al., 2008)).

na fase de projeto e reprojetar completamente uma aplicação legada normalmente está fora de questão. Uma LPS desenvolvida com o modelo reativo costuma ter um número de extensões de granularidade fina de médio para alto e uma LPS desenvolvida com o modelo proativo costuma ter uma granularidade grossa. A razão disso é que, no modelo proativo, ao contrário do reativo, todas as *features* da LPS são implementadas do zero.

Hephaestus

Hephaestus (Bonifácio et al., 2009) é uma ferramenta para gerar artefatos de instâncias específicas de LPS. Em sua versão atual, o Hephaestus apoia o gerenciamento de variabilidade em LPS com diferentes tipos de ativos como: cenários de casos de uso, documentos de requisitos, processos de negócio e mapeamento entre nomes e arquivos de código fonte, para geração de arquivos executáveis em um formato adequado para compilação de programas Java e AspectJ.

Hephaestus possui uma interface gráfica simples, que permite a desenvolvedores de produto selecionarem os artefatos e modelos de entrada para o processo de derivação de produto. A ferramenta avalia os artefatos e modelos e então gera artefatos de instâncias específicas da LPS. Os artefatos e modelos de entrada são:

- **Modelo de *features* (MC)**: contém as *features* da LPS e suas relações. Esse tipo de modelo foi introduzido na SubSeção 3.2.3;

- **Modelo de instância (MI)**: representa os produtos da LPS. Geralmente são representados por um conjunto de *features* selecionadas que satisfazem as restrições do modelo de *features*;

- **Ativos da linha de produto (ALP)**: representam os artefatos da LPS (podem ser de diferentes tipos, como código fonte, casos de uso, etc);

- **Conhecimento de configuração (CC)** (em inglês *configuration knowledge*): artefato responsável por relacionar as configurações de *features* a ativos da LPS. Atualmente existem diferentes representações do conhecimento de configuração, das quais a mais simples, basicamente, mapeia uma *feature* a um ativo.

Apesar de fazer uso de diferentes modelos no processo de geração de produtos, os modelos não são definidos pelo *Hephaestus*. Eles podem ser definidos manualmente ou com apoio de outras ferramentas. Por exemplo, modelos de *features* e modelos de instância podem ser definidos por meio de ferramentas como *Feature Modeling Plugin* ou *Feature IDE* (ambas plugins do *Eclipse*). A Figura 3.10 mostra uma visão de alto nível da geração de produtos com o *Hephaestus*.

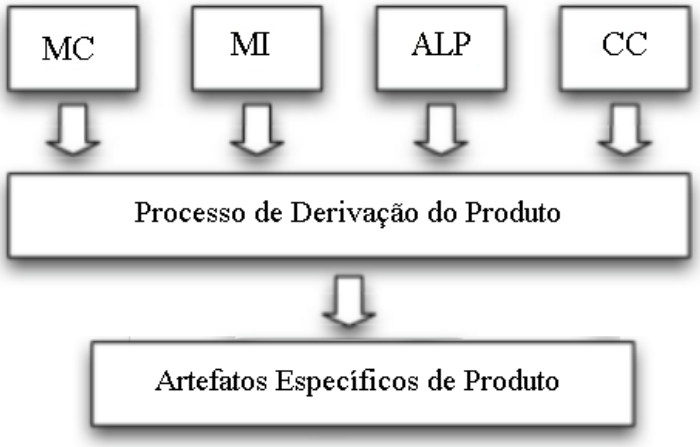


Figura 3.10: Visão de alto nível da geração de produtos com o Hephaestus (adaptada de Bonifácio et al. (2009)).

3.7 Linhas de Produto de Veículos Autônomos

Para encontrar trabalhos envolvendo LPS de veículos autônomos foram realizadas buscas nas bibliotecas digitais ACM, IEEE, ISI Web Of Knowledge, Scopus e Scirus. As palavras-chave da busca foram divididas em duas categorias, uma no escopo de linha

de produtos ("*product line*","*product family*") e a outra no de veículos autônomos ("*unmanned*", "*uncrewed*", "*autonomous*", "*crewless*"). Com as palavras-chave e operadores lógicos, a seguinte string de busca foi criada:

("*product line*" **OR** "*product family*") **AND** ("*unmanned*" **OR** "*uncrewed*" **OR** "*autonomous*" **OR** "*crewless*")

Essas palavras foram escolhidas para abranger o maior número de trabalhos possíveis. Poderia ter sido colocado "*software product line*" em vez de "*product line*", mas optou-se pela segunda alternativa porque o número de artigos retornado por esta é sempre, no mínimo, igual ao número retornado com "*software product line*". A string foi utilizada em buscas no resumo de trabalhos das bases IEEE, ACM e Scopus e no título de trabalhos das bases Scirus e ISI Web of Knowledge (nessas duas bases não há opção de procurar nos resumos).

A tabela 3.1 mostra o número de resultados em cada uma das bases. Apesar de o número de trabalhos retornados ser razoável, poucos trabalhos de fato possuem foco em linhas de produtos de veículos autônomos. Muitas das palavras "*product line*" encontradas nos resultados não se referem à abordagem de desenvolvimento de linha de produtos, ou é um tópico explorado muito superficialmente, e algumas das ocorrências da palavra "*autonomous*" (e seus sinônimos) não se referem a sistemas autônomos (o que também era esperado).

Base eletrônica	Número de artigos retornados
ACM	6
IEEE	3
ISI Web of Knowledge	0
Scirus	0
Scopus	32

Tabela 3.1: Resultado da busca nas diferentes bases

Dos trabalhos encontrados na busca, o mais relacionado a este projeto é o de Polzer et al. (2009). Em seu trabalho, eles combinaram técnicas de desenvolvimento dirigido por modelo com sistemas de controle de prototipação rápida (*rapid control prototyping systems*) para o desenvolvimento de uma linha de produtos com a aplicação de manobras automáticas para estacionamento de um veículo autônomo experimental.

A abordagem proposta no trabalho melhora o desenvolvimento de linha de produtos por meio de um processo baseado em modelos que é dividido em três etapas (modelo, protótipo e produto). Além disso, para proporcionar adaptação simples do sistema de controle a variabilidades nos sensores e atuadores, foi usada uma *camada de abstração de hardware*; essa camada isola o comportamento do controlador principal da aplicação

(desenvolvido em Simulink) dos parâmetros específicos de hardware que são armazenados em um arquivo de configuração XML.

O veículo é controlado por um microcontrolador, que recebe entrada de uma variedade de sensores (informações como velocidade do veículo, distância de obstáculos e direção do veículo). Além disso, ele pode receber comandos de motoristas via controle de rádio sem fio. Por fim, o controlador age no ambiente por meio de vários atuadores, incluindo acelerador de motor, freios e direção.

A Figura 3.11 mostra o diagrama de *features* da aplicação. A maior parte dos atuadores são obrigatórios, exceto o freio traseiro. Os sensores possuem mais variabilidade: para sensores de distância podem ser escolhidas tecnologias infravermelho ou ultrasônica. Além desses, o sensor de direção também é opcional. No contexto do controlador (que executa o algoritmo para estacionar o veículo), deve ser decidido se será usado ou não um algoritmo que leva a direção em conta. As *features* pertencentes à categoria *plant* são opcionais, uma vez que, em algumas situações, pode se querer simular a aplicação inteira dentro da ferramenta de modelagem. A *feature* de *debug* permite integrar mecanismos de depuração no modelo e, por fim, a *feature* de comando do usuário dá ao veículo a funcionalidade de receber comandos do usuário (por meio de um controle remoto, por exemplo).

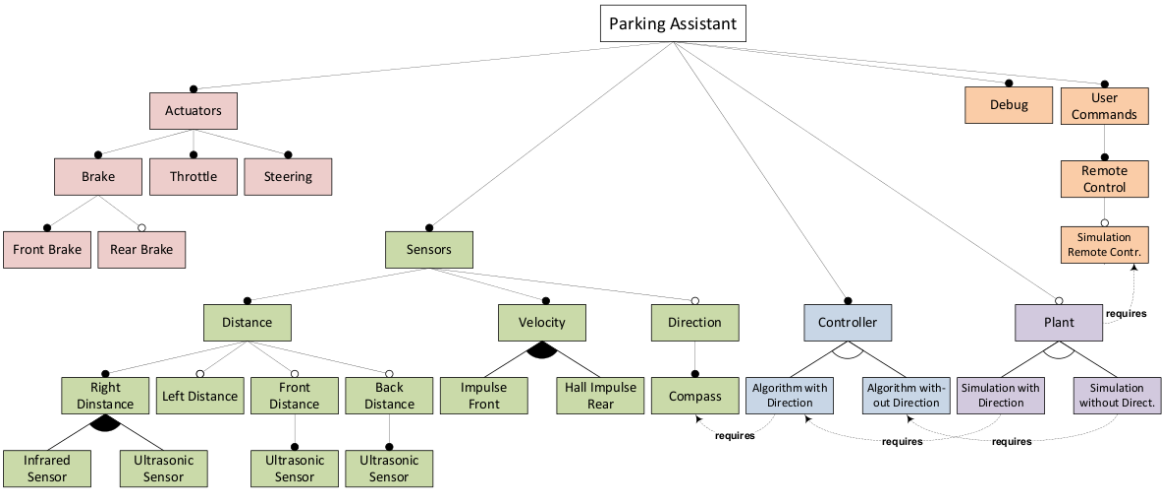


Figura 3.11: Modelo de *features* da LPS do veículo autônomo do assistente de estacionamento (Polzer et al., 2009).

Segundo Lutz (2008), a NASA (*National Aeronautics and Space Administration*) possui a intenção de adotar a abordagem de LPS para o desenvolvimento da alguns dos seus sistemas. Lutz (2008) resumiu resultados de experiências industriais reunindo um conjunto de possibilitadores de verificação para uso na engenharia de aplicações das LPS da NASA. De acordo com o autor, as lições aprendidas podem ser úteis no desenvolvi-

mento de LPS críticas em relação à segurança, de longa duração ou de sistemas altamente autônomos.

Nesse artigo são descritos quatro conjuntos de sistemas que possuem *features* de LPS mas que, de acordo com o autor e as definições formais de LPS, não são de fato uma LPS. Entre os sistemas, o autor cita o TechSat21, um conjunto de pequenos satélites sensíveis a contexto desenvolvidos para testar tecnologia de formação de voo de veículos espaciais, e que podem mudar rapidamente a formação de acordo com os requisitos da missão. O projeto foi cancelado, mas muito do software foi reusado em uma missão subsequente.

Uma outra aplicação citada no trabalho refere-se a um conjunto de ativos que compõem sistemas interferômetros (componente importante de telescópios espaciais). A Figura 3.12 mostra como ativos dos sistemas foram usados em múltiplos interferômetros. O projeto não foi desenvolvido como uma LPS, e a arquitetura real acabou evoluindo de maneira diferente da arquitetura documentada.

O projeto *ASys*, proposto por Sanz et al. (2007), tem como objetivo o desenvolvimento de ciência e tecnologia para construção de sistemas autônomos. A maior diferença entre este projeto e outros da mesma área é que ele abrange todo o domínio de sistemas autônomos. O plano de desenvolvimento do projeto Asys segue uma estratégia de linha de produtos.

A abordagem do projeto ASys é definida pelos autores como uma abordagem para engenharia de LPS de sistemas cognitivos de tempo real. Os autores planejam proporcionar ativos tecnológicos para engenheiros construírem sistemas cognitivos de tempo real. Isso traz implicações aos mais diferentes aspectos dos sistemas, variando de controladores embarcados de tempo real até mecanismos pensantes (*thinking mechanisms*) de alto nível ou infraestrutura de integração. O aspecto central que é pesquisado atualmente é um conjunto de ativos arquiteturais para apoiar a auto-consciência e o meta-controle do sistema.

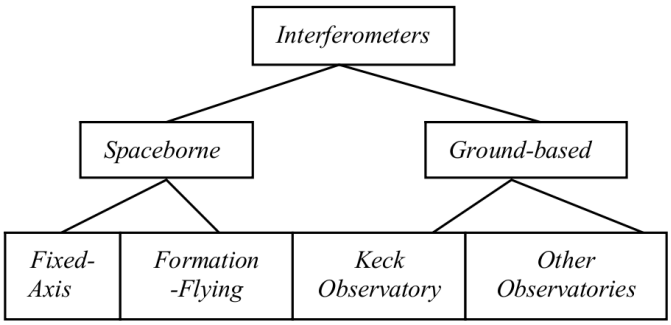


Figura 3.12: Modelo de *features* dos ativos de sistemas de interferômetros da NASA (Lutz, 2008).

Svahnberg e Mattsson (2002) apresentaram um estudo de casos de uma empresa no domínio de veículos automaticamente guiados, a qual migrou uma geração de LPS com foco em hardware para uma nova geração com foco em software. Os autores caracterizaram itens que motivaram a transição, fatores que a complicaram e desafios relativos às gerações futuras de LPS; entre os desafios descritos estão: aumentar o uso de hardware *off-the-shelf*, permitir software de terceiros coexistir e colaborar com o sistema, reduzir o tempo de aprendizado para novos desenvolvedores de software e considerar diferentes tamanhos de hardware. No trabalho também é apresentado um processo em três estágios para a migração de uma LPS antiga para uma nova: criação de uma arquitetura de LPS, avaliação da arquitetura de LPS e desenvolvimento de um plano de migração.

Outro trabalho relevante encontrado é o de Botterweck et al. (2010), que encontra-se no contexto de LPS de sistemas embarcados modelados em linguagens específicas de domínio, mais especificamente Simulink. As maiores contribuições do trabalho são a análise de mecanismos de variabilidade no ambiente MATLAB/Simulink, conceitos para realizar as variabilidades com transformações nos modelos, um mecanismo de mapeamento que modifica o modelo de acordo com configurações e o conceito de "*pruning*" (poda, em português), a limpeza de componentes que são indiretamente influenciados por decisões de configuração.

Para a modelagem de variabilidades e geração de instâncias específicas de LPS, Botterweck et al. (2010) propõem uma abordagem dirigida a modelo que se baseia em transformações de ordem maior. Essa abordagem considera diferentes processos (como análise de *features*, implementação de *features* e configuração do produto) e diferentes artefatos (como modelo de *features* e modelo de implementação do domínio). Para introduzir variabilidades em modelos Simulink, Botterweck et al. (2010) usam uma técnica chamada *variabilidade negativa* (descrita na Seção 3.2.4). Em Simulink, a variabilidade de um modelo seguindo essa técnica pode ser definida e manipulada de duas maneiras:

- **Manipular o modelo, descrevendo as informações sobre variabilidade externamente**: A manipulação do modelo é feita por uma ferramenta externa. Para esse fim, durante a derivação de produto é necessário analisar a estrutura do arquivo do modelo e remover blocos que representam as *features* desativadas (bem como seus sinais de entrada e saída) para obter a configuração desejada. Um exemplo de modelagem de *features* variáveis com uma relação do tipo ou-exclusivo em Simulink, seria agrupar suas saídas na mesma porta de entrada do próximo bloco no modelo de união; quando esse modelo fosse configurado pela ferramenta externa, um dos blocos seria obrigatoriamente excluído, resultando em um modelo Simulink válido.
- **Embarcar mecanismos de variabilidade internamente no modelo**: Elementos artificiais que não implementam nenhuma funcionalidade são inseridos no mo-

delo e servem apenas como mecanismo para modelar variabilidades. A Figura 3.14 mostra os blocos do Simulink que podem ser utilizados para modelar variabilidade (nesse caso, seguindo o modelo de *features* da Figura 3.13) de *features* com diferentes relações e tipos. *Features* obrigatórias não necessitam de mecanismos para serem definidas, já as opcionais podem ser definidas com o bloco `triggered subsystem`, um bloco especial que ativa ou desativa os blocos com os quais ele está interligado, dependendo de um sinal booleano. *Features* com relações do tipo ou-exclusivo podem ser agrupadas utilizando-se o bloco `Switch`, o qual terá o valor de saída exatamente igual a uma das entradas (no caso de uma LPS, a saída será referente ao bloco da *feature* selecionada); por fim, *features* com relações do tipo ou-inclusivo podem ser definidas usando-se o bloco `integration`, o qual pode usar diferentes funções para integrar os sinais de entrada (por exemplo, mínimo ou máximo).

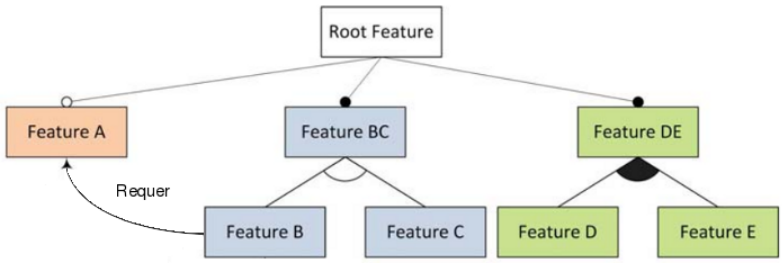


Figura 3.13: Modelo de *features* referente ao modelo Simulink da Figura 3.14 (adaptada de Botterweck et al. (2010)).

Na abordagem proposta, Botterweck et al. (2010) usaram uma combinação das duas possibilidades de manipulação de variabilidade. A razão para isso é que, por um lado, a variabilidade introduzida internamente no modelo mantém as características do desenvolvimento baseado em modelo (como teste e simulação simplificada e possibilidade de captura de dependências) e, por outro lado, um produto derivado não deve conter nenhum mecanismo de variabilidade. Na derivação de um produto nessa abordagem, primeiramente são removidos os blocos não selecionados, e depois é executado um processo com operações de *pruning*, que removem os mecanismos de variabilidades e todas as linhas que não devem fazer parte do modelo que está sendo gerado.

Fragal et al. (2011) apresentam uma abordagem para mapear *features* de linhas de produtos para modelos Simulink. Para apoiar a abordagem foi desenvolvido no trabalho uma ferramenta chamada SimulinkImport. Essa ferramenta importa um modelo Simulink e seus subsistemas importantes e realiza um meapeamento dos elementos desse modelo para *features* e expressões de *features* de uma LPS.

A abordagem proposta por Fragal et al. (2011) segue um processo de seis passos no qual, ao final, é gerado uma instância de produto da LPS – i.e, um modelo Simulink

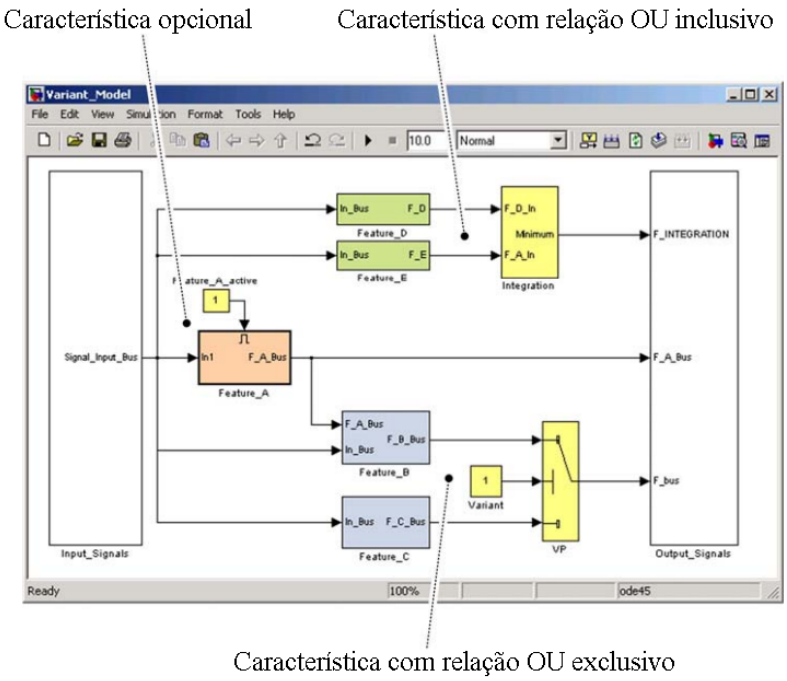


Figura 3.14: Modelo Simulink com a variabilidade definida com blocos do próprio ambiente (adaptada de Botterweck et al. (2010)).

contendo apenas os blocos associados às *features* selecionadas para o produto. Além da ferramenta SimulinkImport, essa abordagem usa a ferramenta Pure::variants para apoiar esse processo de mapeamento e geração de produto. A abordagem é ilustrada no trabalho com um exemplo de um mini-VANT, um modelo Simulink *opensource* de um VANT simples.

Braga et al. (2011) realizaram uma evolução no projeto do VANT Tiriba (aeronave descrita na Seção 2.4.1) baseando-se nos conceitos de LPS. Para essa reengenharia de projeto, foi criado um modelo de *features* de 108 *features*, cuja maior parte é obrigatória e referente às características físicas da aeronave (que são, na sua maior parte, implementadas em hardware sem nenhuma relação com software). Além do modelo de *features*, no trabalho foi criado a arquitetura desse projeto de LPS, a qual pode ser vista na Figura 3.15. Todos os blocos do diagrama são obrigatórios, com a exceção daqueles que aparecem hachurados na figura, que são opcionais. É possível ver que a maior parte das variabilidades na LPS dessa aeronave se encontra no *payload* (carga), mas também há variabilidade em seus componentes aviônicos (sensores opcionais *margnectic field 3D* e *optical measumen unit*) e nos sistemas de lançamento e recuperação – o lançamento pode ser feito a mão ou com estilingues e o pouso pode ser feito por procedimento de aterrissagem automática ou paraquedas

A maior diferença entre este trabalho de mestrado e o trabalho de Braga et al. (2011) é que neste trabalho foi investigado como as variabilidades podem ser implementadas

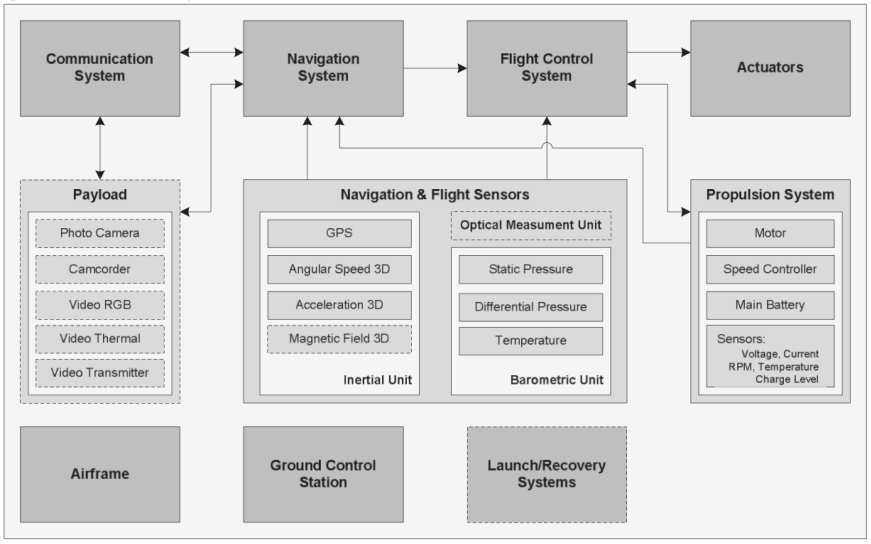


Figura 3.15: Arquitetura do projeto de LPS do Tiriba Braga et al. (2011).

em modelos Simulink e como duas ferramentas de gerenciamento de configuração podem ser usadas para apoiar a engenharia de LPS em Simulink. Para realizar uma prova de conceito foi criada uma pequena linha de produtos com oito *features* baseada no Tiriba (será descrita em maiores detalhes no capítulo 4). A maior diferença entre as *features* deste trabalho e do trabalho de Braga et al. (2011) é que elas possuem um foco maior no software e as de Braga et al. (2011) ao hardware; como todo o software do Tiriba foi desenvolvido em Simulink, para todas as *features* definidas neste trabalho existe alguma parte no modelo referente a elas.

3.8 Considerações finais

Neste capítulo a abordagem de LPS foi caracterizada; os principais conceitos da abordagem, algumas técnicas e mecanismos de implementação, atividades envolvidas no desenvolvimento de LPS e alguns métodos foram descritos. Os assuntos que receberam mais atenção foram ferramentas comerciais e acadêmicas que apoiam o desenvolvimento de LPS, modelos de adoção/migração para a abordagem, orientações para realizar a adoção e, finalmente, trabalhos envolvendo LPS de veículos autônomos.

Capítulo

4

Modelagem de Variabilidades em Simulink

4.1 Considerações Iniciais

Este capítulo apresenta um conjunto de padrões que mostram maneiras pelas quais variabilidades de diferentes *features* podem ser modeladas em modelos Simulink. As *features* criadas no VANT usado como estudo de caso desta dissertação são descritas, bem como os mecanismos de variabilidade usados para modelar cada uma delas no modelo Simulink da aeronave.

Na Seção 4.2 é mostrado como os blocos da biblioteca de blocos Simulink podem ser usados como mecanismos para modelar variabilidades de diferentes tipos de *features* e variabilidades de diferentes tipos de relações em Simulink, tanto na parte de fluxo de dados como de máquinas de estados finitos. Na Seção 4.3 são descritas as *features* que foram criadas neste trabalho, a partir do modelo Simulink do VANT Tiriba. Por fim, na seção 4.4 é mostrado como os padrões para modelagem de variabilidades (introduzidos na seção 4.2) foram usados para reestruturar o modelo de modo a comportar corretamente as variabilidades projetadas.

4.2 Padrões de Modelagem de Variabilidades em Simulink

Modelos Simulink que possuem variabilidades podem necessitar de alguns mecanismos para estruturá-las. Essa necessidade existe, pois muitas das variabilidades denotadas diretamente no código (sem auxílio de qualquer mecanismo) resultariam em um modelo inválido. A Figura 4.1 mostra um exemplo de um modelo Simulink com dois subsistemas referentes a variabilidades de duas *features* alternativas (ou-exclusivas) e ambos são ligados à mesma porta de um subsistema. Um modelo com essas características é inválido, pois cada porta de entrada de bloco só pode ser conectada a apenas um bloco.

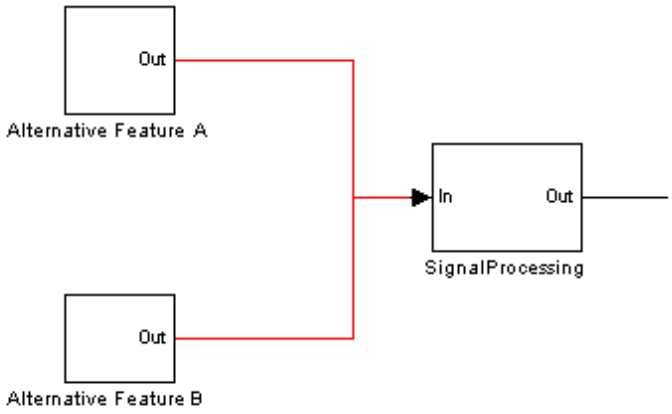


Figura 4.1: Modelo Simulink inválido. Modelagem de variabilidades sem o uso de nenhum mecanismo

Para modelar adequadamente variabilidades em modelos Simulink é possível utilizar os próprios blocos do ambiente. Alguns desses mecanismos de variabilidades foram descritos superficialmente na Seção 3.7. A seguir, esses mecanismos e alguns outros serão descritos com mais detalhes.

4.2.1 Mecanismos de variabilidade para a parte de fluxo de dados de modelos Simulink

Organização de features com subsistemas

Subsistema é um tipo de bloco Simulink que substitui um conjunto de blocos (e suas ligações) por apenas um bloco no modelo. Caso todos os blocos relacionados a uma *feature* estejam conectados e em uma mesma área espacial do modelo, eles podem ser isolados em um único subsistema. Caso os blocos estejam espalhados em diferentes áreas do modelo, eles podem ser isolados em mais de um subsistema. Uma convenção de nome

pode ser usada para facilitar a distinção entre subsistemas usados para organizar *features* e os demais subsistemas do modelo. Fazer uso de subsistemas para isolar *features* no modelo Simulink faz com que seja possível alcançar uma visão de alto nível mais clara do papel de cada uma das *features* no modelo e facilita a maneira como as *features* podem ser manipuladas.

Features opcionais modeladas com Enabler subsystems

Se um bloco do tipo `enabler` for colocado dentro de um subsistema, ele se transformará em um bloco do tipo `enabler subsystem`. Esse tipo de subsistema é praticamente igual aos Subsistemas simples, com a única diferença de que ele possui uma porta a mais. Se o valor de entrada nessa porta for igual a 0, todo o subsistema será desabilitado, e todas as saídas do subsistema terão o valor 0. Caso a entrada seja maior que 0, o subsistema será executado normalmente.

Esse tipo de subsistema pode ser usado como mecanismo de variabilidade para *features* opcionais. Se todos os blocos referentes a uma *feature* opcional estiverem em `enabler subsystems`, basta modificar o valor das portas `enabler` para habilitar ou desabilitar uma *feature*. A Figura 4.2 mostra esse bloco usado como um mecanismo de variabilidade. O subsistema `Optional Feature Subsystem` possui uma única porta de entrada, que é uma porta do tipo `enabler`; essas portas sempre possuem o símbolo característico que pode ser verificado na figura e são, por padrão, sempre na parte de cima dos blocos que a possuem (ao contrário das portas comuns de sinais que ficam aos lados dos blocos).

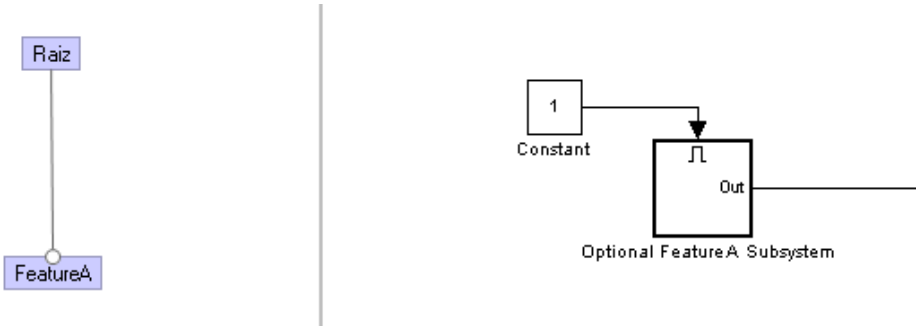


Figura 4.2: Blocos do tipo `enabler subsystems` usados como mecanismo de variabilidade para *features* opcionais

Features com relação do tipo ou-exclusivo modeladas com blocos do tipo switch

Features alternativas (que possuem relação ou-exclusivo) podem ser estruturadas no Simulink com um bloco do tipo `switch`. Esse bloco serve para rotear sinais Simulink e possui três portas de entrada. O seu valor de saída é igual ao valor da sua porta de entrada 1 ou 3 (chamadas de "entradas de dados"), dependendo do valor da porta de entrada

2 (chamada de "entrada de controle"), dos critérios e dos valores de limiar configurados no bloco.

A Figura 4.3 mostra um exemplo de bloco `switch` usado como mecanismo de variabilidade para duas *features* com relação do tipo ou-exclusivo (isoladas nos subsistemas `Alternative Variability A` e `Alternative Variability B`). Se a entrada de controle for conectada a um bloco do tipo constante, apenas uma das *features* será executada, garantindo assim a execução de uma configuração válida do modelo de *features* relacionado às variabilidades do modelo Simulink.

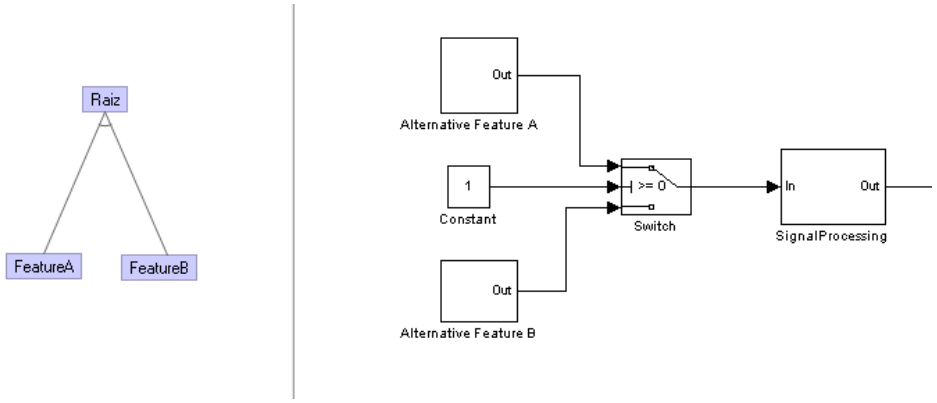


Figura 4.3: Bloco `switch` usado como mecanismo de variabilidade para *features* alternativas

Caso haja mais de duas *features* alternativas em um mesmo agrupamento, pode ser usado um bloco do tipo `multiport switch`, que comporta diversas entradas de dados.

Features com relações do tipo ou-exclusivo modeladas com enabler subsystems, blocos do tipo and e not

Uma possibilidade de modelagem da variabilidade de *features* com relação do tipo ou-exclusivo é utilizar `enabler subsystems` e uma combinação de blocos do tipo `AND` e `NOT`. Cada `enabler subsystem` deverá ter a sua porta `enable` ligada à saída de um bloco de comparações lógicas do tipo `and`. Esse bloco `and` deverá possuir uma porta de entrada para cada *feature* agrupada na relação ou-exclusivo. Todas essas portas de entrada serão ligadas a um bloco do tipo `NOT` exceto uma, a que será conectada ao bloco do tipo `constant` responsável por ativar o `enabler subsystems` quando ele possuir o valor 1; os blocos `NOT` serão conectados aos outros blocos `constant` responsáveis por ativar os `enabler subsystems` referentes às outras *features*.

Esse mesmo padrão deverá se repetir para os demais `enabler subsystems`, conforme ilustra a Figura 4.4. Desse modo é garantido que nenhum ou apenas um `enabler subsystem` ficará habilitado durante toda a execução ou simulação do sistema, garantindo assim, uma configuração válida do modelo no que tange às *features*.

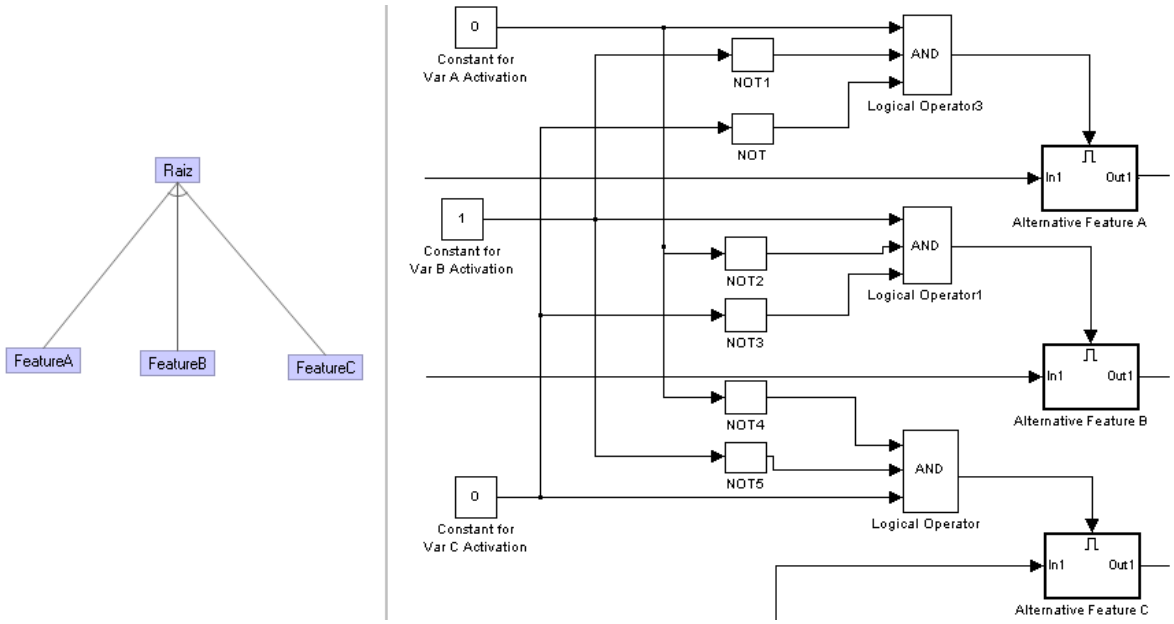


Figura 4.4: `Enabler subsystems` combinados com blocos do tipo `and` e `not` usados como mecanismo de variabilidade para modelar *features* com relação ou-exclusivo

Features com relação do tipo ou-inclusivo modeladas com blocos de integração

Quando os blocos referentes às *features* que possuam relacionamento ou-inclusivo estão espalhados pelo modelo, elas poderão ser modeladas com `enabler subsystems`. Caso os blocos referentes a cada uma das *features* não estejam espalhados (i.e. exista alguma espécie de dependência de sinal entre eles), um bloco de integração pode ser usado como mecanismo de variabilidade. Um bloco de integração pode ser implementado de diversas maneiras como, por exemplo, utilizando funções de máximo, mínimo (para o caso de mais de uma *feature* agrupada na relação ou-inclusivo ser selecionada), ou qualquer outro tipo de função que faça sentido no contexto das *features*.

A Figura 4.5 mostra um exemplo de um bloco de integração (função máximo) usado como mecanismo de variabilidade.

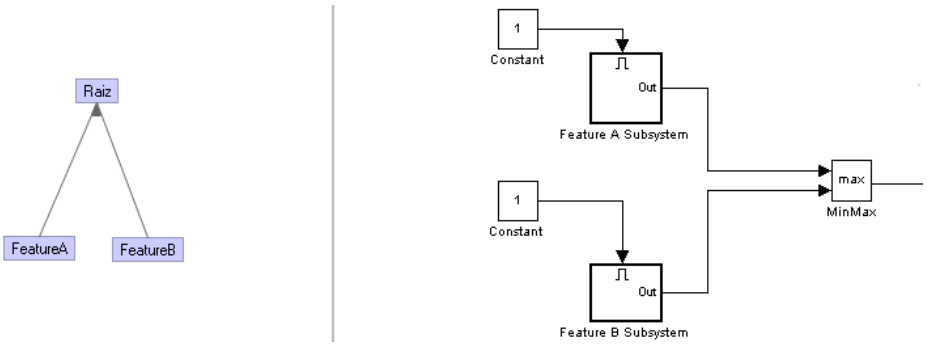


Figura 4.5: Bloco de integração usado como mecanismo de variabilidade para *features* com relação ou-inclusivo

Variabilidades de features com relações de dependências ou hierarquia modeladas com blocos do tipo switch

Para modelar variabilidade de *features* que possuam relação de hierarquia ou relação de dependência, pode ser usado um bloco do tipo `switch` como mecanismo de variabilidade. A Figura 4.6 mostra um exemplo que poderia tanto ser de um cenário de variabilidades de relação de dependência como de hierarquia; isso ocorre quando há uma *feature* com relação hierárquica e a *feature* filha só poderá ser selecionada caso a *feature* pai também o seja, assim como uma *feature* que dependa de outra só pode ser selecionada se a *feature* da qual ela dependa também for selecionada.

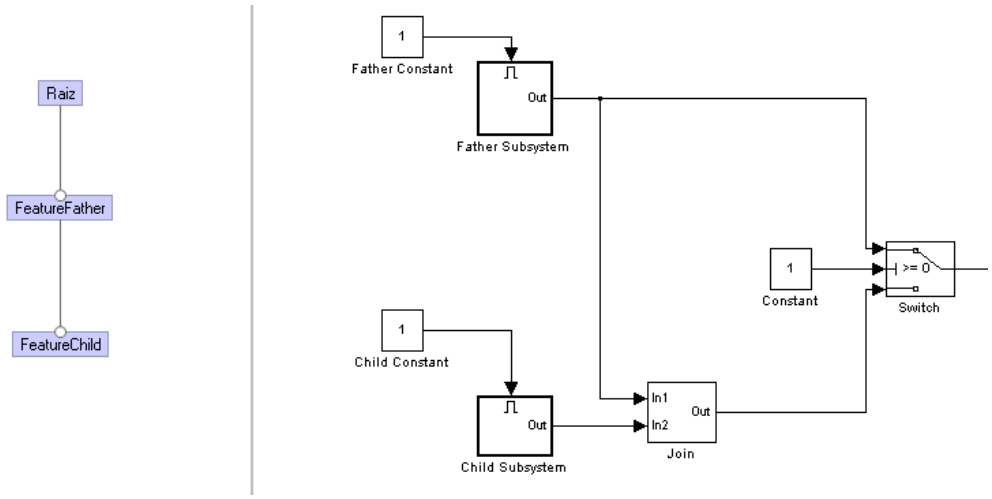


Figura 4.6: Uso de um bloco do tipo `switch` como mecanismo de variabilidade para *features* com relações de hierarquia

Variabilidade de features com relações de dependências ou hierárquicas modeladas com enabler subsystems

A variabilidade de *features* com relações de dependência ou de hierarquia e cujas variantes funcionais estejam encapsuladas em um ou mais blocos do tipo `enabler subsystem` pode ser modelada com um bloco do tipo `AND` (para comparações lógicas); esse bloco é usado para determinar se a entrada da porta de `enabler` da *feature* dependente ou filha será ativada ou não. A *feature* filha ou dependente será ativada se o seu valor de ativação e o da *feature* da qual ela depende ou é filha forem verdadeiros (única caso no qual a saída do bloco `AND` será verdadeira). Caso um dos dois valores de ativação seja falso, ela não será ativada. A Figura 4.7 mostra um exemplo desse padrão.

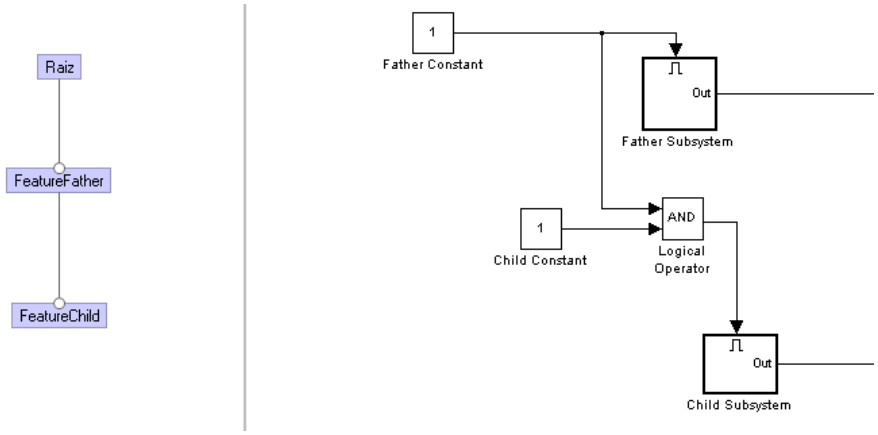


Figura 4.7: Bloco do tipo `enabler subsystem` usado como mecanismo de variabilidade para *features* com relações de hierarquia

4.2.2 Mecanismos de variabilidade para máquinas de estados do MATLAB/Simulink

Além do fluxo de dados do MATLAB/Simulink, as máquinas de estados também podem conter variabilidades. Aqui são apresentadas duas abordagens que podem ser usadas como mecanismos de variabilidades: substituições de máquinas de estados inteiras, cada qual com uma variabilidade específica, ou criação de transições condicionadas a variáveis que definem variabilidades.

Máquinas de estados únicas para cada variabilidade

Uma maneira de usar essa abordagem como mecanismo de variabilidade é criar uma máquina de estados diferente para cada possível variabilidade nas máquinas de estados do modelo e encapsular cada uma delas em um `enabler subsystem` de modo que o projetista da aplicação possa ativar apenas a máquina de estados desejada e desativar todas as outras. Todos esses `enabler subsystems` com as máquinas de estados devem então ser encapsulados em um subsistema que possua uma infraestrutura para gerenciar as entradas e saídas corretas da máquina de estado, que deverá ser selecionada para execução.

A Figura 4.8 mostra o subsistema (`FSM Selector`) usado como essa infraestrutura. Nesse exemplo existem duas entradas e duas saídas que são obrigatórias, ou seja, são sempre usadas independentemente da máquina de estados escolhida; há também uma entrada e uma saída opcional e cada uma delas poderá ou não ser usada, dependendo da máquina de estados selecionada. Para esse exemplo, caso a entrada opcional não seja usada, o `enabler subsystem opt subsys1` deverá ser desabilitado, apesar de isso não interferir muito no sistema, uma vez que a máquina que tem essa entrada não será executada; já no caso da saída, se a saída opcional não for usada, é importante que o

`enabler subsystem` `opt subsys2` seja desabilitado, pois se ele não for, o sistema simulará ou executará uma configuração inválida, pois esse `enabler subsystem` funcionará mesmo com a sua entrada sendo sempre 0.

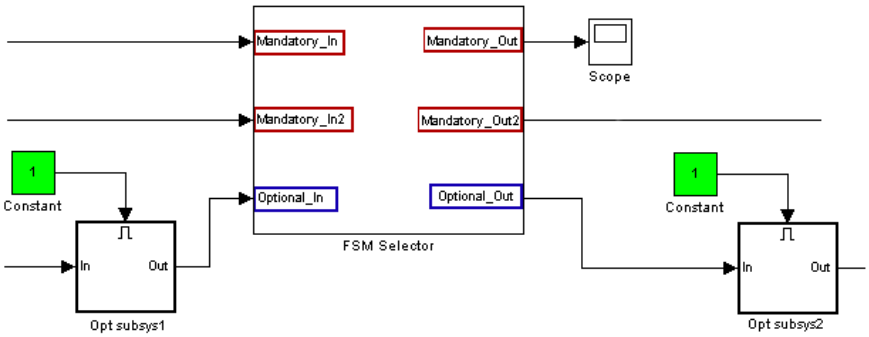


Figura 4.8: Subsistema usado como infraestrutura para organizar entradas e saídas das máquinas de estados que implementam variabilidades

A Figura 4.9 mostra o interior do subsistema `FSM Selector` da Figura 4.8, que possui duas máquinas de estados únicas, cada uma executando comportamentos de variabilidades diferentes. Além da variabilidade interna de cada uma das máquinas, elas também possuem variabilidade nas entradas e saídas: a máquina `FSM1` possui duas entradas e três saídas e a máquina `FSM2` possui três entradas e duas saídas. Como as duas primeiras saídas do `FSM selector` podem vir tanto de uma máquina como da outra, os sinais das portas de saídas 1 e 2 de ambas as máquinas foram organizados por meio de um bloco do tipo `switch`.

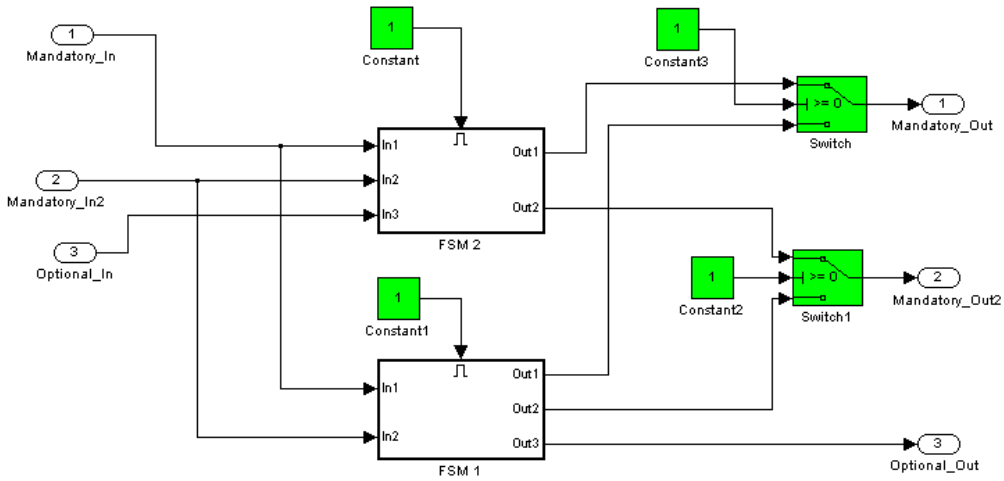


Figura 4.9: Mecanismo de variabilidade usado para organizar MEFs que possuem diferentes variabilidades

Criação de transições condicionadas a variáveis que definem variabilidades

Máquinas de estados do Matlab podem ter entradas vindas da parte de fluxo de dados do MATLAB/Simulink. Essas entradas podem ser usadas como variáveis de condição para executar uma transição de um estado para outro, sendo usadas como variáveis booleanas sozinhas ou com expressões envolvendo operadores lógicos. A Figura 4.10 mostra um exemplo de transição de um estado a outro de uma MEF; essa transição possui uma condição (entre colchetes), que serve para ativar a transição, e também possui uma ação (entre chaves) que ocorre quando a transição é realizada.

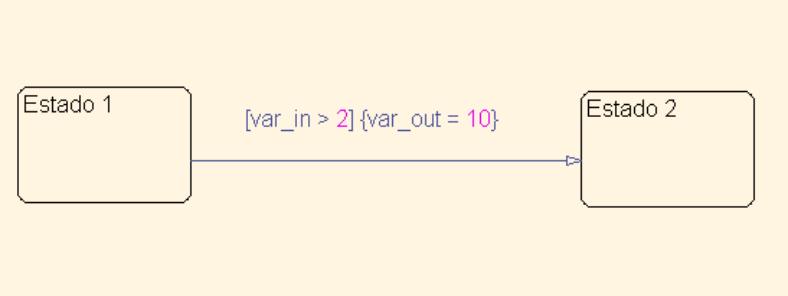


Figura 4.10: Exemplo de transições condicionadas e com ação

Se houver estados específicos na máquina que implementem comportamentos referentes à variabilidades, pode ser criada uma transição alternativa na MEF de modo que ela possa entrar ou não nesses estados dependendo da condição associada às transições. Se a condição das transições for uma variável cujo valor é atribuído a uma entrada da MEF vinda da parte de fluxo de dados, o controle da seleção de variabilidades irá requerer muito pouco esforço podendo ser feito, por exemplo, por meio de um bloco do tipo `constant`.

A Figura 4.11 mostra uma máquina de estados simples. Ela foi criada para discretizar valores de pico de uma onda senóide: sempre que o valor da onda for menor que 10, a saída `discrete_out` da máquina será 0 e sempre que o valor for igual a 10 (valor de pico da onda), a saída `discrete_out` da máquina também será 10.

Essa máquina possui três estados: o `Low_peak`, que é o estado inicial da máquina e que deixa a saída `discrete_out` em 0 e dois estados variáveis, o `High_preak` e o `High_preak_counter`. A máquina entra no estado `High_preak` se o valor da variável de entrada `counter` for falso e se a entrada `sine_wave_value` for maior que 10 (ou seja, o valor do pico definido). Esse estado deixará a saída `discrete_out` com valor 10. O estado `High_preak_counter` é exatamente igual ao `High_preak`, exceto pelo fato de que ele incrementa a variável de saída `peaks` em 1 toda vez que a máquina entrar nesse estado (essa variável é um contador do número de vezes que um pico foi detectado na máquina de estados com essa variabilidade) e que ele só entrará nesse estado se o valor da variável de

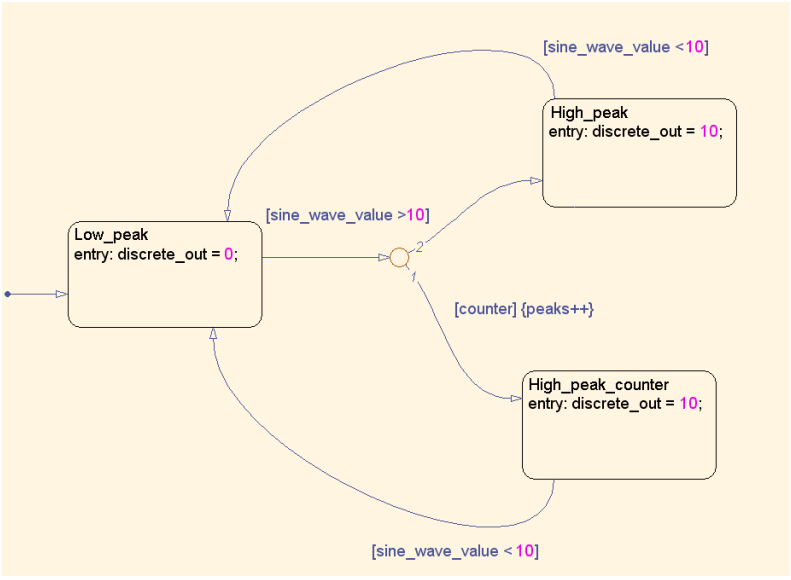


Figura 4.11: Exemplo de MEF com variabilidades implementadas com base em transições condicionadas

entrada `counter` for verdadeiro. Quando o valor da entrada `sine_wave_value` for menor que 10, a máquina voltará ao estado `Low_peak`.

Nessa máquina de estados existem duas configurações de variabilidade possíveis, uma que faz transição do estado `Low_peak` para o estado `High_preak` e outra que faz transições do estado `Low_peak` para o `High_preak_counter`. Se o seu valor for maior que 0, a variabilidade que conta o número de vezes que a máquina detectou um pico será ativada, caso contrário será desativada. A Figura 4.12 mostra o bloco de máquina de fluxo de estados no qual está a MEF da Figura 4.11. Na figura é possível ver as entradas da MEF da onda senóide (bloco com rótulo *Sine Wave*) e a entrada que ativa a variabilidade que conta o número de picos (bloco `constant`.

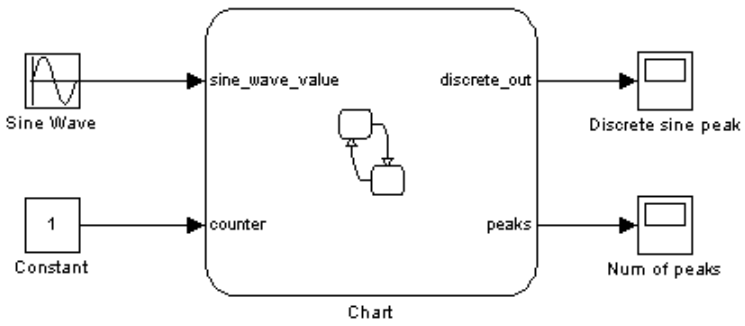


Figura 4.12: Bloco de máquina de fluxo de estados da MEF da Figura 4.11 e seus blocos e varíaveis de entrada e saída

A Figura 4.13 mostra os valores da onda senoide de entrada, a saída da MEF (onda de seno discretizada) e o valor da variável de saída peaks; as partes A) e B) mostram, respectivamente, os valores para a MEF com a variabilidade que conta os picos desativada e ativada.

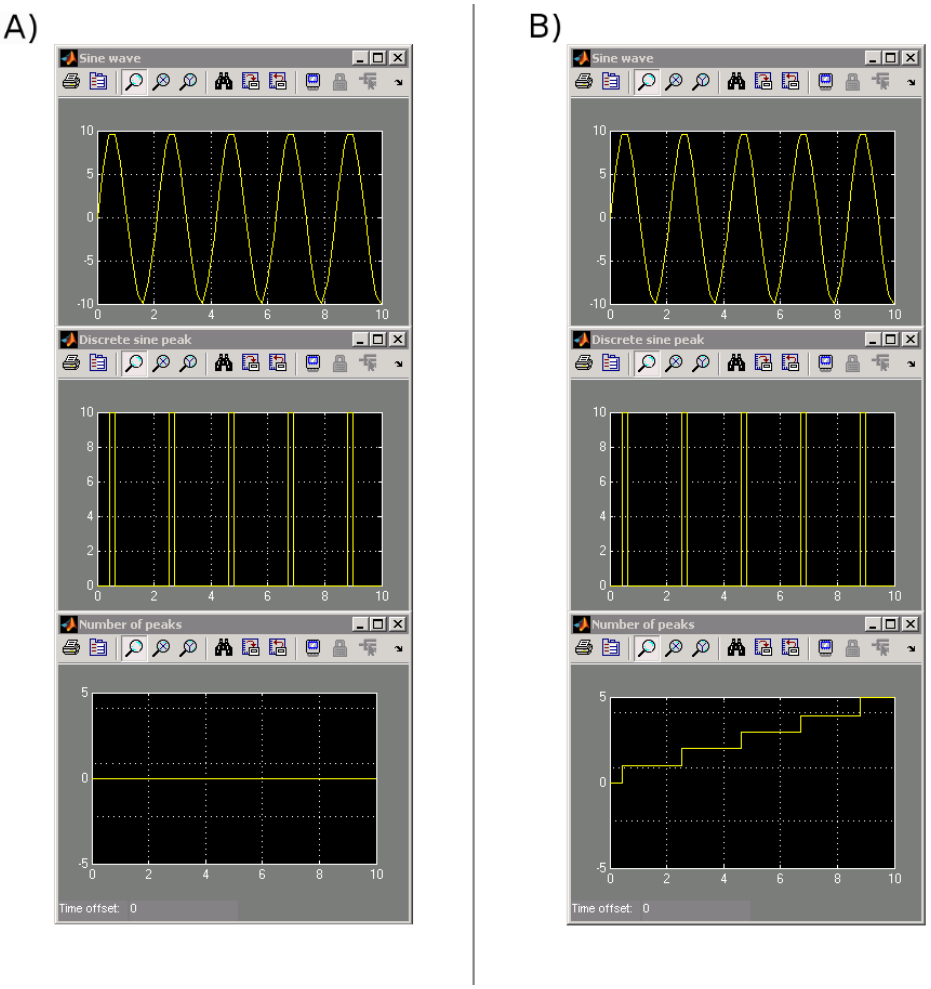


Figura 4.13: Sinais da onda senoide que alimenta a entrada `sine_wave_value` e sinais de saída `discrete_out` e `num_peaks` para a MEF sem e com a variabilidade que conta o número de picos (A e B respectivamente)

4.3 Projeto de Features Baseadas no Modelo Simulink do VANT Tiriba

Para projetar as variabilidades, além do estudo do domínio dos VANTs, houve contato com desenvolvedores e especialistas do domínio da empresa AGX, que cedeu o modelo Simulink utilizado no trabalho. Foram definidas seis *features* opcionais e duas *features* alternativas que são *sub-features* de uma obrigatória, conforme ilustra o modelo de *features*

da Figura 4.14. Das *features* opcionais, duas estão relacionadas ao *payload* da aeronave, uma ao paraquedas e três a características da missão (implementadas na máquina de estados que controla a missão). Esse diagrama foi criado apenas para fornecer uma prova de conceito, de modo que ele ficou muito simplificado, possuindo *features* opcionais como a maior parte das suas *features*.

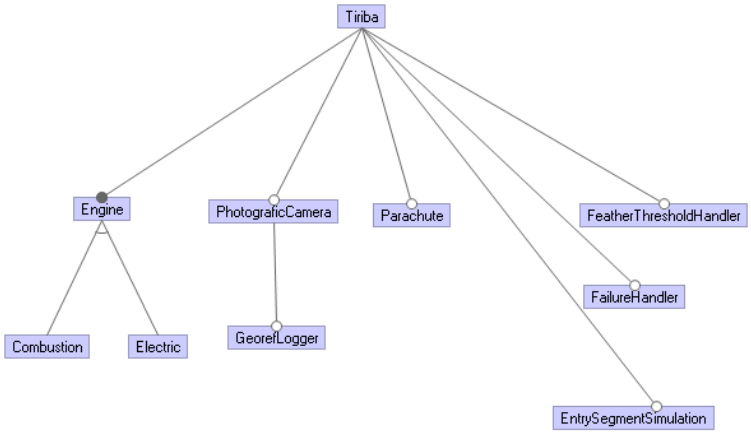


Figura 4.14: Modelo de *features* criado com base no modelo Simulink do VANT Tiriba

Para o *payload* da aeronave foram criadas duas *features* opcionais: *Photografic camera* e *Georef log*. A *feature Photografic camera* controla a máquina fotográfica do avião; se ela não for selecionada, significa que o avião não tirará fotos. A *feature Georef log* refere-se a um sistema que utiliza informações do GPS da aeronave para armazenar em um arquivo as coordenadas geográficas dos locais onde a aeronave tirou fotos; essa *feature* é uma sub-*feature* da *Photografic camera*, uma vez que, para sua existência fazer sentido, é necessário que o avião tire fotos.

A *feature parachute* estava planejada desde o começo do projeto do Tiriba, no entanto, quando esta dissertação foi escrita, ela ainda não havia sido implementada no modelo Simulink da aeronave. A implementação do paraquedas é simples: ele recebe um pulso constante das principais unidades da aeronave e, caso algum pulso fique um determinado tempo sem ser observado, o paraquedas é ativado, pois a não emissão do pulso por uma das unidades da aeronave significa necessariamente que a unidade travou e muito provavelmente a aeronave cairá.

As *features* relacionadas a características da missão são: *Entry segment simulation*, *Feather threshold handler* e *Failure Handler*; essas três *features* estão implementadas na máquina de estados que controla a missão. A *feature Entry segment simulation* refere-se a uma simulação executada pela unidade de navegação da aeronave ao entrar em um novo segmento na rota especificada para a missão, com o objetivo de encontrar a melhor manobra para passar de um segmento para outro. A *feature Feather threshold handler*

faz com que a aeronave execute manobras para corrigir sua rota sempre que se afastar mais que um determinado limiar da linha traçada na rota da missão. A *feature Failure Handler* faz com que a aeronave execute manobras para recuperar pontos de fotografia perdidos na missão qualquer que seja o motivo.

Duas *features* foram criadas para o motor da aeronave: *combustion* e *electric*. A *feature combustion* refere-se ao motor a combustão; esse tipo de motor garante uma autonomia maior para a aeronave, já que ele possui bem mais potência que o motor elétrico (representado pela *feature electric*). No entanto, o controle da aceleração do motor a combustão é mais complexo. Tanto a aceleração como a desaceleração do motor devem ser feitas de modo gradativo e suave, caso contrário o motor pode morrer, o que é arriscado, pois pode significar a perda da aeronave ou acidentes. Uma outra característica do motor a combustão é que ele possui um limite inferior de velocidade, e, caso a velocidade seja menor que esse limite, o motor também morrerá. O motor elétrico possui bem menos restrições de controle, a sua aceleração ou desaceleração pode ser feita de modo brusco e a velocidade do motor pode chegar a 0 que o motor não morrerá. Apesar de ele possuir uma autonomia menor que o motor a combustão, ele é uma boa alternativa, pois é mais seguro, o controle é fácil de ser desenvolvido e integrado à aeronave e sua manutenção é mais simples.

4.4 Reestruturação do Modelo Simulink do Tiriba para Comportar Variabilidades

O modelo Simulink do Tiriba foi reestruturado nos pontos referentes a cada uma das *features* criadas para permitir sua configuração em ferramentas de gerenciamento de configuração. Para fazer essa reestruturação, foram usados alguns dos mecanismos de variabilidade descritos na Seção 4.2. O modelo obtido ao final dessa reestruturação contém os blocos referentes às *features* criadas. É possível gerar manualmente uma configuração específica de um produto válido a partir da parametrização de blocos que controlam os mecanismos de variabilidade; no entanto, gerar produtos dessa maneira é muito mais trabalhoso e passível de erros e inconsistências do que com o uso de alguma ferramenta de gerenciamento de configuração.

Os blocos referentes às *features* do *payload* da aeronave foram removidos do seu local de origem e colocados em dois `enabler subsystems`, um representando a *feature Georef log* e outro a *feature Photografic camera*. Um bloco do tipo `multiport switch` de três portas foi usado como mecanismo de variabilidade. Com esse `multiport switch` há três configurações possíveis dessas *features*: a seleção da *feature Georef log* e *Photografic camera*, a seleção apenas da *feature Photografic camera* ou nenhuma das duas selecionadas.

A Figura 4.15 mostra a reorganização do modelo Simulink do Tiriba na parte referente a essas *features*; as partes A e B da figura são, respectivamente, o modelo antes e depois da reorganização.

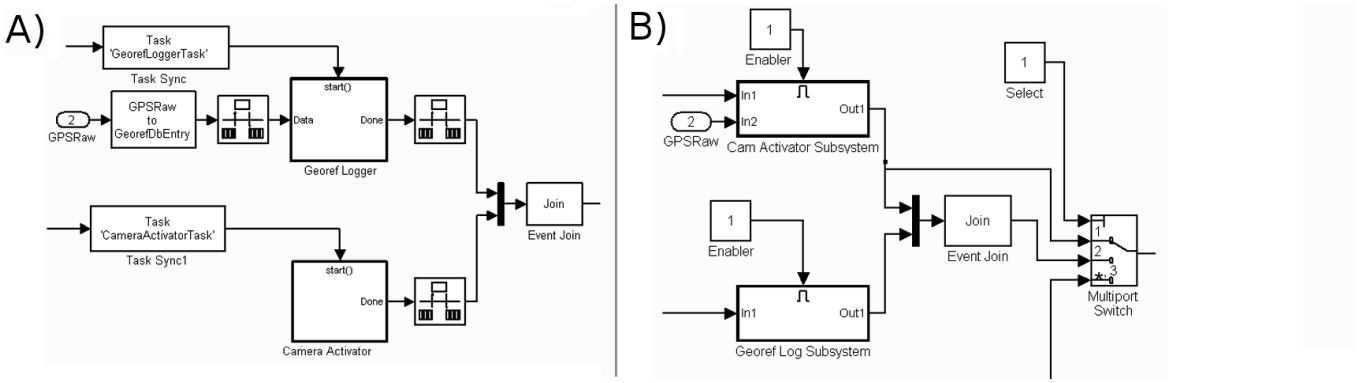


Figura 4.15: Parte do modelo Simulink do Tiriba referente às *features* da *payload* da aeronave antes e depois da reestruturação

As variabilidades das *features* do motor estão presentes em duas áreas distintas, uma referente a configurações do motor (para configurar a velocidade mínima e máxima) e outra referente a uma função que suaviza a aceleração ou desaceleração do motor (necessária apenas para o motor a combustão). A Figura 4.16 mostra como foi modelada a variabilidade referente à configuração de velocidade de motor. Essa configuração pode ser feita via parametrização dos blocos, porém nessa modelagem de variabilidades foi decidido deixar os blocos de configuração pré-configurados e deixar a seleção de cada uma delas no mesmo padrão que foi usado nas outras *features* (utilização de mecanismos de variabilidades, nesse caso, o bloco switch).

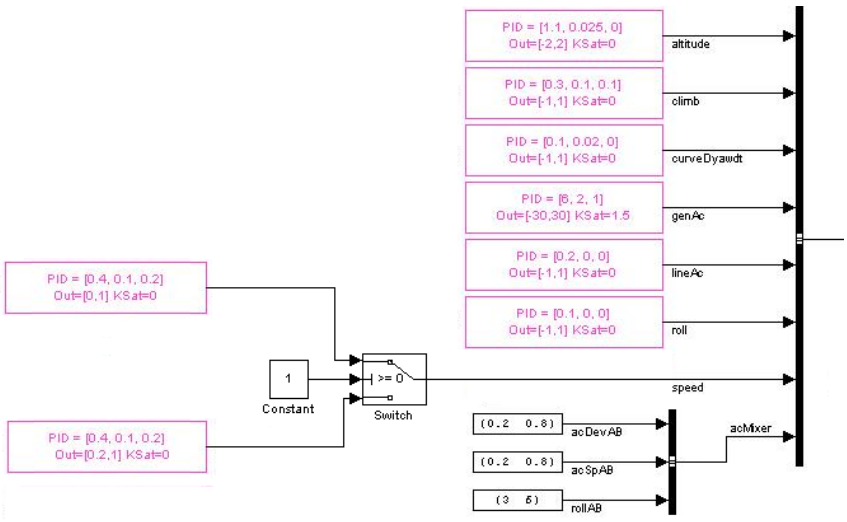


Figura 4.16: Modelagem da variabilidade dos motores do Tiriba: parâmetros para velocidade do motor elétrico e a combustão

A Figura 4.17 mostra alguns subsistemas que aplicam funções aos sinais do motor. O último deles é usado para determinar o nível da aceleração do motor: no caso do motor a combustão deve ser usado uma função que suaviza a transição de um nível de aceleração do motor (implementada pelo bloco `Lever`. O sinal transmitido pela linha conectado à segunda porta do subsistema é o nível de aceleração solicitado pelo sistema de controle. No caso do motor elétrico, o nível de aceleração do motor é esse próprio sinal (porta 3 do bloco switch) já que esse motor não precisa de qualquer tratamento de suavização para o nível de aceleração do motor.

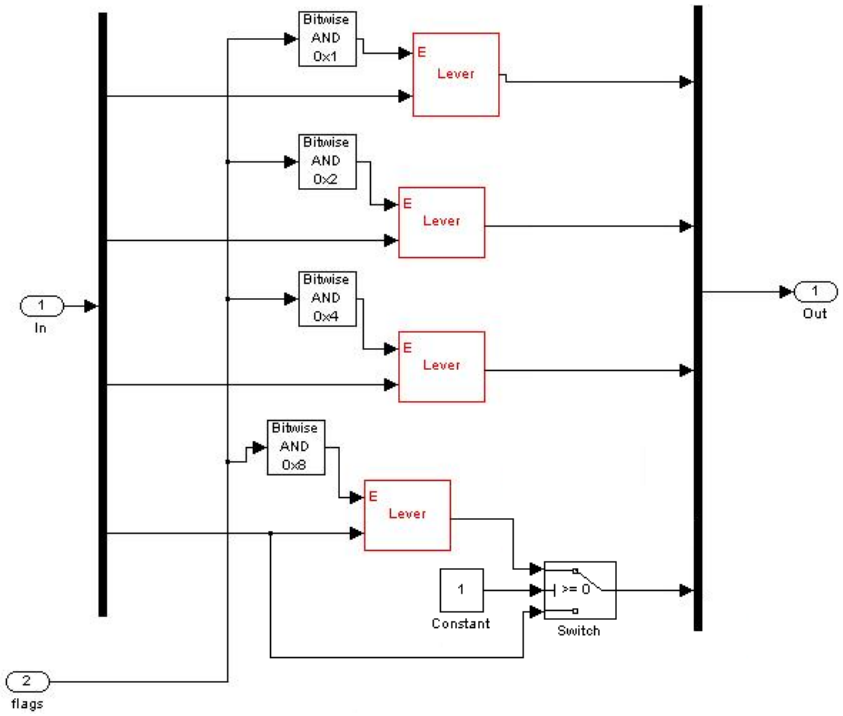


Figura 4.17: Variabilidades dos motores elétrico e a combustão no que diz respeito ao nível de aceleração dos motores

No Tiriba, o subsistema que controla a ativação do paraquedas funciona com base num solenoide. Ele recebe um sinal de pulso de cada um dos principais subsistemas (unidade de controle, pressão, inercial e navegação). Se algum desses pulsos não chegar corretamente é porque muito provavelmente uma das unidades travou e será melhor ativar o paraquedas. Como no modelo do Simulink os blocos referentes à *feature* do paraquedas já estavam isoladas em um subsistema, para estruturar a variabilidade dessa *feature* bastou adicionar ao subsistema um bloco de `enabler`, transformando-o em um `enabler subsystem` (tipo de subsistema que pode ser usado para implementar a variabilidade de uma *feature* opcional conforme descrito na Seção 4.2.1). A Figura 4.18 A mostra o modelo do paraquedas antes da adição do bloco`enabler` e a Figura 4.18 B mostra o modelo após a adição.

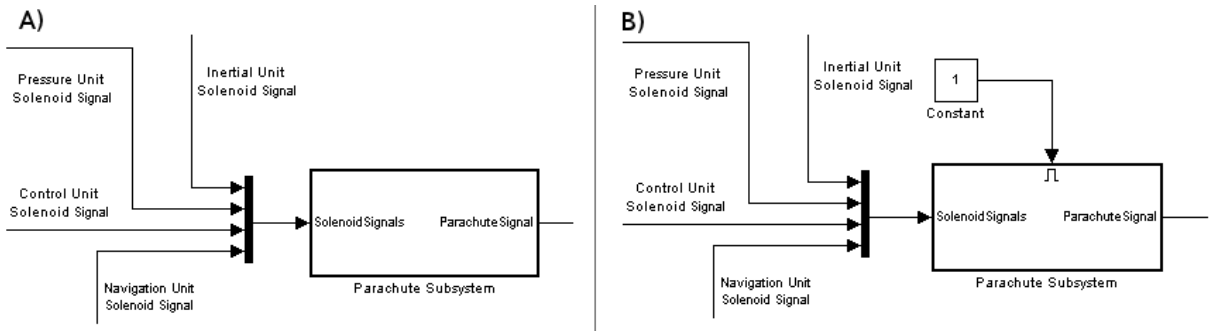


Figura 4.18: Modelo Simulink do Tiriba antes e depois da reestruturação para comportar as variabilidades da *feature* do paraquedas

Por fim, as variabilidades referentes às características da missão (todas opcionais) foram implementadas na máquina de estados que controla a missão do Tiriba seguindo o mecanismo de variabilidade de criação de transições condicionadas a variáveis que definem variabilidades. Como todas essas funções estão implementadas em uma MEF, uma transição alternativa foi criada na MEF para cada *feature*, cada qual condicionada a uma variável booleana de entrada diferente; o estado referente a cada variabilidade é ignorado caso a variável associada a essa variabilidade seja configurada como falsa e a transição é feita para ele caso a variável seja configurada como verdadeira.

A Figura 4.19 ilustra esse mecanismo de variabilidade criado para a *feature Entry segment simulation*. Caso o projetista da aplicação deseje selecionar essa *feature* no modelo, ele deverá definir o valor da porta de entrada da máquina de estados `SimEnable` como `true`. Desse modo, a transição para o estado `Simulating` será tomada, executando assim a simulação descrita na Seção 4.3

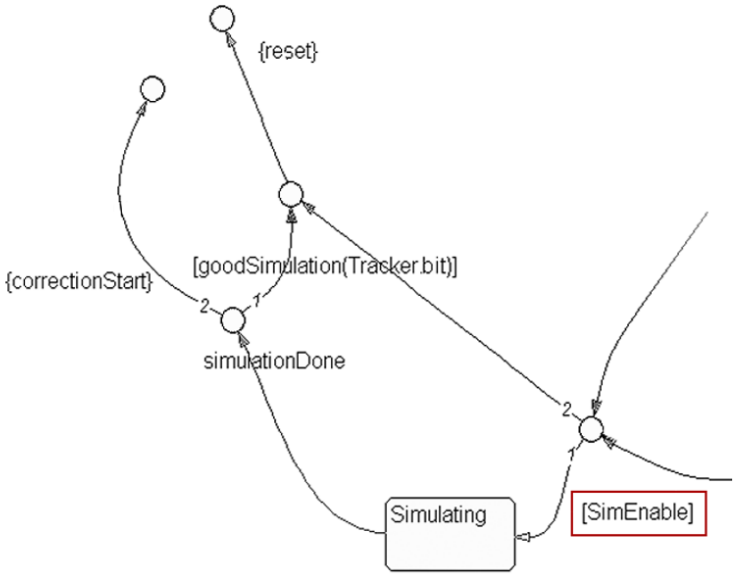


Figura 4.19: Adição de transição alternativa na parte da MEF referente à *feature Entry segment simulation* para comportar variabilidade da *feature*

A Figura 4.20 mostra a MEF que controla a missão do Tiriba inserida na parte de fluxo de dados do modelo do Tiriba. É possível ver as três portas de entrada criadas na MEF para o mecanismo de variabilidade: SimEnable, FailureHandler e FeatherThrHandler.

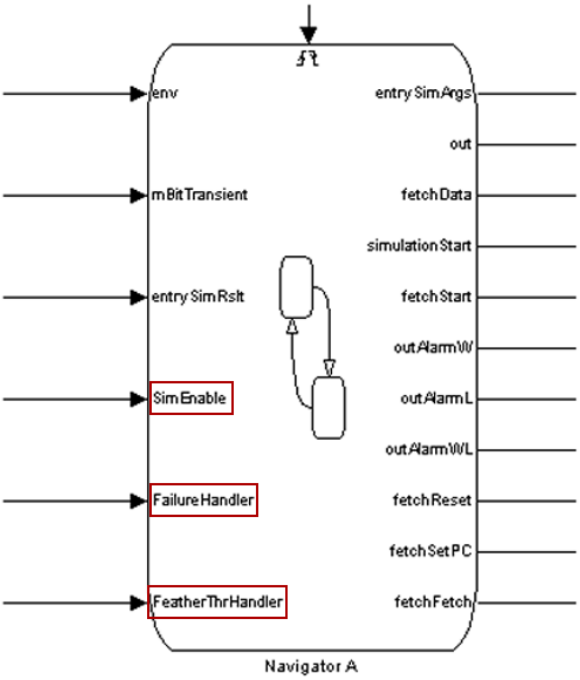


Figura 4.20: MEF que controla a missão aeronave: entradas relacionadas ao mecanismo de variabilidade estão destacadas

O mecanismo de variabilidade de criação de transições condicionadas a variáveis de entrada na MEF foi escolhido ao invés de se utilizarem MEFs únicas para cada variabilidade porque a utilização do segundo mecanismo iria exigir a criação de oito MEFs únicas, o que daria bastante trabalho tanto para criação das MEFs quanto para a organização da infraestrutura necessária para a seleção da MEF da variabilidade desejada. Para modelar a variabilidade do mecanismo escolhido foram criadas apenas três transições condicionadas às três entradas criadas na MEF.

4.5 Considerações finais

Neste capítulo foram apresentadas algumas possibilidades de modelagem de variabilidades em modelos Simulink por meio de mecanismos de variabilidade; essas possibilidades cobrem diferentes tipos de *features* e relações entre *features*. Depois, foi mostrado como esses mecanismos de variabilidade foram utilizados para modelar variabilidades de algumas *features* que foram criadas no modelo Simulink do VANT Tiriba.

No próximo capítulo é mostrado como as variabilidades de uma LPS modelada em Simulink podem ser gerenciadas por duas ferramentas que possuem abordagens diferentes.

Capítulo

5

Gerenciamento de Configuração de um VANT com as ferramentas Hephaestus e Pure::variants

5.1 Considerações Iniciais

Este capítulo descreve um plugin da ferramenta Pure::variants e uma extensão da ferramenta Hephaestus, as quais apoiam o gerenciamento de configuração de LPS em Simulink. Também é mostrado como as ferramentas podem ser usadas para realizar o gerenciamento de configuração em LPS; para isso as *features* que foram criadas no modelo Simulink do Tiriba são usadas como exemplo. Por fim, uma comparação entre as duas ferramentas é realizada.

Na Seção 5.2 o plugin Pure::variants conector é descrito e é explicado como ele pode ser usado para realizar o gerenciamento de configuração de LPS em Simulink e também é mostrado como as *features* criadas no modelo Simulink do VANT Tiriba foram configuradas com a ferramenta. Na Seção 5.3 a extensão feita na ferramenta Hephaestus para apoiar o gerenciamento de configuração e LPS em Simulink é descrita e sua implementação apresentada; depois, assim como na seção 5.2, é mostrado como as *features* criadas no modelo Simulink do VANT Tiriba foram configuradas com o Hephaestus. Na Seção

5.4 é apresentada uma comparação entre as duas ferramentas, ilustrando as principais características, vantagens e desvantagens de cada uma.

5.2 Pure::variants Connector para Simulink

Pure::variants é uma ferramenta comercial amplamente usada no gerenciamento de configuração em linhas de produtos. Entre os plugins da ferramenta, há o *Simulink connector*, que permite o gerenciamento de configuração em linhas de produtos em Simulink que utiliza uma abordagem para gerenciamento de configuração semelhante ao apresentado por Dziobek et al. (2008). O gerenciamento de configuração com o Simulink conector faz uso do conceito de "ponto de variação". Um ponto de variação encapsula a informação de variabilidade de uma determinada variante funcional e é definido pelos seguintes elementos:

- Um identificador único
- Parâmetro de variabilidade: descreve um parâmetro de configuração em um modelo Simulink, que pode ser configurado para selecionar uma variante funcional.
- Mecanismos de variabilidade: descrevem como variantes funcionais são selecionadas em diferentes lugares dos modelos Simulink. Eles garantem que depois de se configurarem parâmetros de variabilidade, apenas as variantes selecionadas serão executadas. Esses mecanismos de variabilidades são implementados por um conjunto de blocos específicos, exportados da Pure::variants para o MATLAB/Simulink mas que são usados de modo igual aos padrões introduzidos no capítulo 4. Exemplos de blocos usados como mecanismos de variabilidade são apresentados na Seção 4.2.

A maior parte dos blocos de mecanismos de variabilidade precisam de sinais de entrada que controlam a sua execução, servindo assim para a seleção da variabilidade. Os blocos responsáveis por esses sinais são denominados de blocos de controle e geralmente controlam a variabilidade no modelo por valores definidos pelo parâmetro de variabilidade de seu ponto de variação.

A Pure::variants permite a definição do valor que será atribuído ao parâmetro de variabilidade usado pelos blocos de controle, com base em três artefatos principais: modelo de *features*, modelo de instância (uma configuração específica do modelo de *features*) e modelo de variante, que mapeia expressões de *features* do modelo de *features* para valores nos parâmetros de variabilidade, conforme ilustra a Figura 5.1.

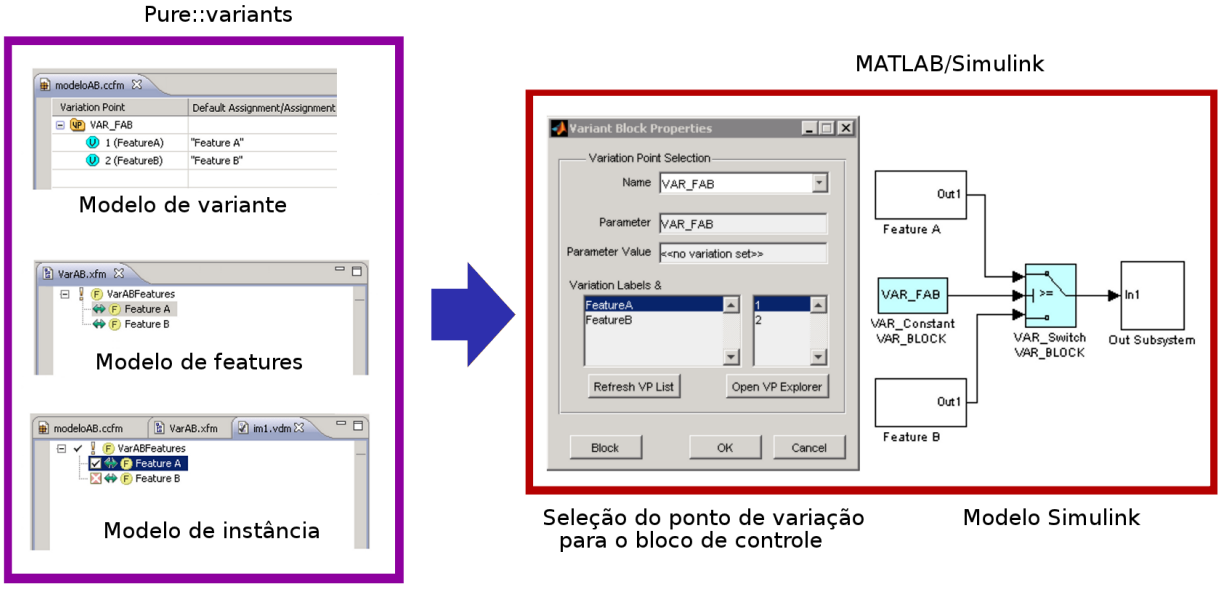


Figura 5.1: Visão alto nível do funcionamento da Pure::variants

5.2.1 Gerenciamento de configuração no modelo Simulink do Tiriba usando a Pure::variants

Para configurar o modelo do Tiriba de modo a ter suas variabilidades gerenciadas pela ferramenta Pure::variants, foram criados seis pontos de variação: um para a *feature* do paraquedas, um para as *features* relacionadas ao *payload* (*Photografic camera* e *Georef log*), um para as *features* do motor (*combustion* e *electric*) e um para cada uma das *features* relacionadas à MEF que controla a missão da aeronave (*Entry Segment Simulation*, *Failure Hander* e *Feather Threshold Handler*).

As *features Photografic camera* e *Georef log* foram configuradas com um mesmo ponto de variação, cujo parâmetro de variabilidade pode ter os valores 1, 2 ou 3 referindo-se, respectivamente, a nenhuma das duas *features* selecionadas, apenas à *Photografic camera* selecionada e às duas selecionadas. A Figura 5.2 ilustra esse ponto de variação adicionado a três blocos de controle: um diretamente ligado ao `switch` (usado como mecanismo de variabilidade) e dois ligados a um bloco do tipo `comparator`, que está ligado na porta `enabler` de cada um dos subsistemas que estão encapsulando os blocos das *features Photografic camera* e *Georef log*.

Os blocos referentes às *features* do motor estão em duas áreas espaciais distintas no modelo do Tiriba. As duas *features* do motor (*Electric* e *Combustion*) são alternativas e já haviam sido implementadas com o uso de um bloco `switch` como mecanismo de variabilidade, conforme ilustram as figuras 4.16 e 4.17. Para configurar essas *features* com o Pure::variants foi criado um ponto de variação com dois valores possíveis: 0 e 1. Um

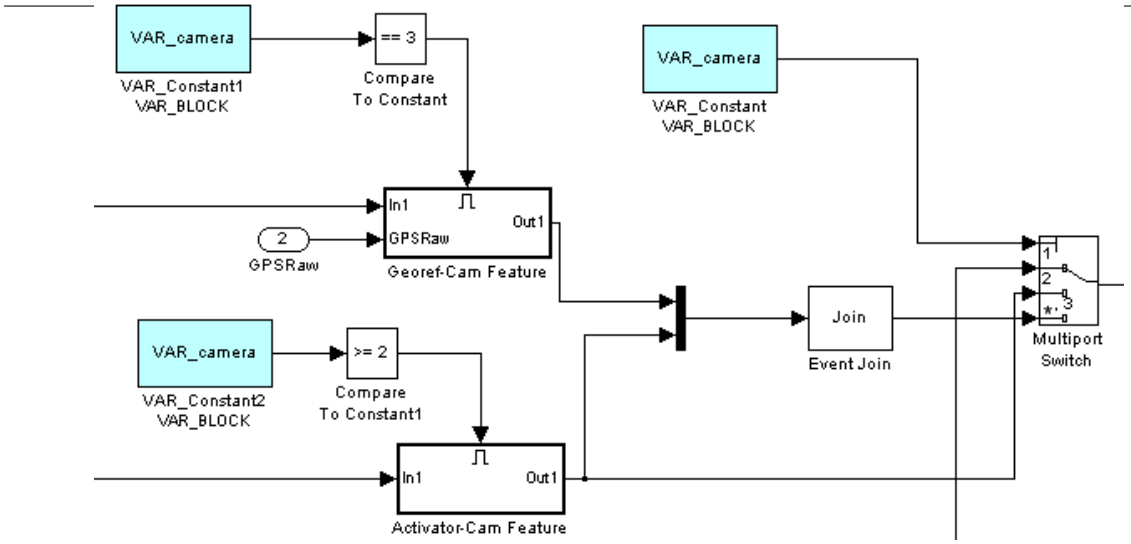


Figura 5.2: Configuração das *features* relacionadas à *payload* da aeronave do modelo Simulink do Tiriba seguindo a abordagem da Pure::variants

bloco de controle com esse ponto de variação foi criado para cada um dos blcos `switch` conforme ilustra a Figura 5.3.

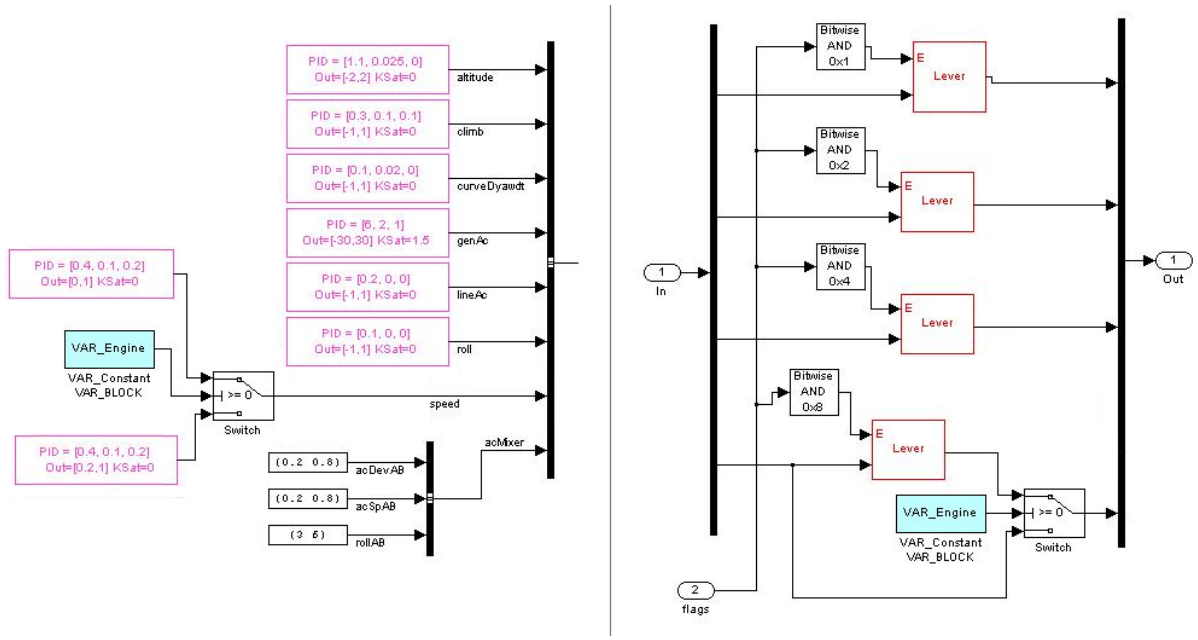


Figura 5.3: Configuração das *features* do motor do Tiriba com a Pure::variants

A *feature* do paraquedas foi a mais simples de ser configurada. O parâmetro de variabilidade do ponto de variação criado para ela pode possuir valor 0 ou 1, e um bloco de controle com esse ponto de variação substituiu o bloco `constant` da Figura 4.18 B conectado à porta `enabler`, conforme ilustra a Figura 5.4.

Os pontos de variação das *features* relacionadas ao comportamento da missão da aeronave (*Entry Segment Simulation*, *Failure Hander* e *Feather Threshold Handler*), por

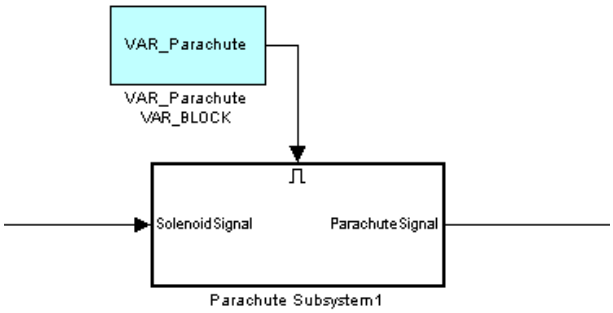


Figura 5.4: Subsistema referente à *feature Parachute* configurada com a Pure::variants

serem *features* opcionais simples, possuem valores 0 ou 1. As variabilidades dessas *features* foram implementadas na MEF com transições condicionadas a variáveis que podem levar ou não a estados que executem as variabilidades. O valor dessas variáveis vem de portas de entrada da MEF (para cada *feature* foi criada uma porta e uma variável de entrada). O bloco conectado a essas portas de entrada é um bloco de controle associado ao ponto de variação que define as variabilidades, conforme ilustra a Figura 5.5.

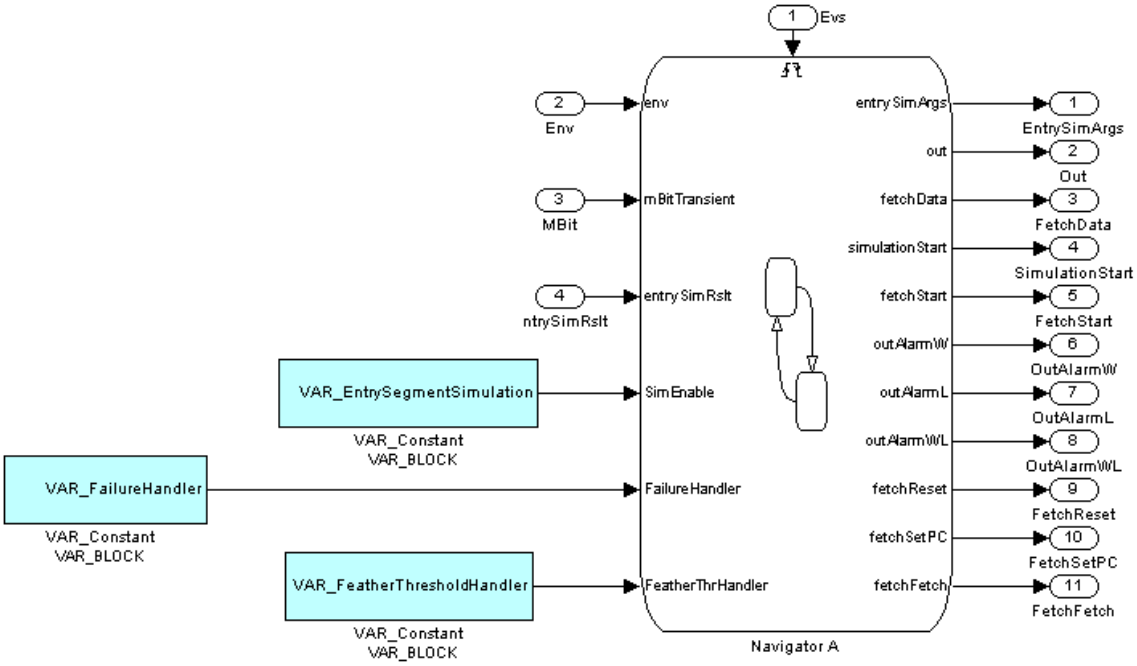


Figura 5.5: Variabilidade das *features Entry Segment Simulation*, *Failure Hander* e *Feather Threshold Handler* configuradas com a Pure::variants

Conforme dito na Seção 5.2, a configuração de uma LPS com o Pure::variants connector para Simulink deve ser feita em duas partes, uma no modelo Simulink da LPS (por meio da estruturação de variabilidades do modelo com mecanismos de variabilidade e definição de pontos de variações em blocos de controle que controlam esses mecanismos) e outra na

própria Pure::variants (definição de um modelo de *features*, um modelo de instância e um modelo de variabilidade).

No início da Seção 5.2.1 foram descritos os mecanismos de variabilidade usados para estruturar as *features* do Tiriba, bem como os pontos de variação usados em seus blocos de controle. A Figura 5.6 mostra o modelo de *features*, instância e variabilidade do Tiriba criados na Pure::variants. No modelo de variabilidade há todas as expressões de *features* que, juntamente com o modelo de instâncias, servem para atribuir valores aos pontos de variação. É possível ver que a maior parte das expressões é bastante simples (com apenas o nome de uma *feature* ou o nome com um operador do tipo `not`).

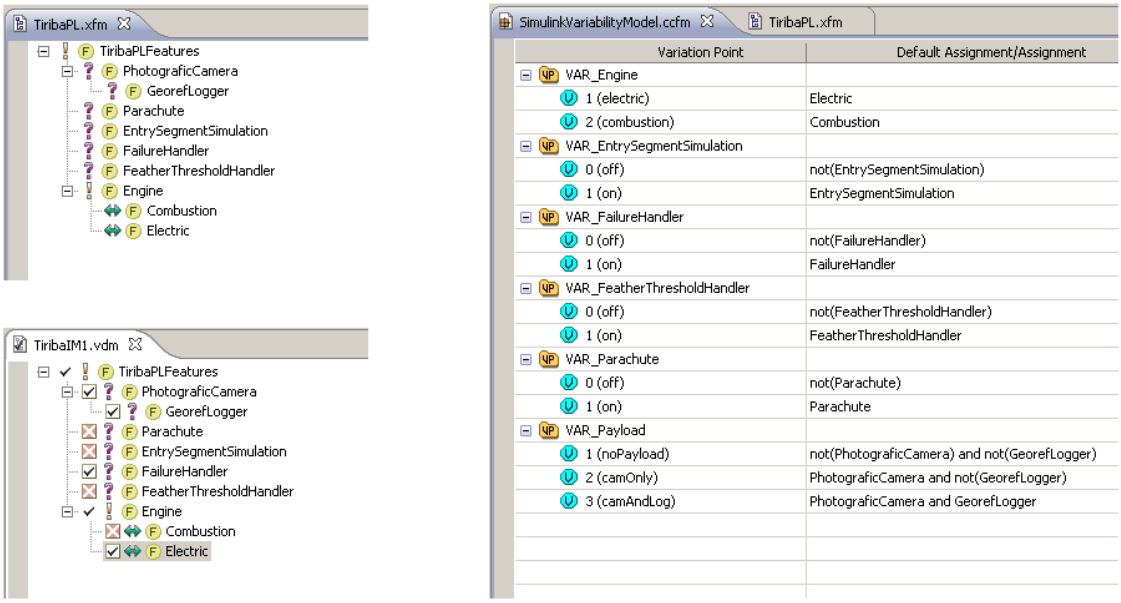


Figura 5.6: Modelo de *features* e instância (lado esquerdo) e modelo de variabilidade (lado direito) criados no Pure::variants para realizar o gerenciamento de configuração na LPS criada a partir do modelo Simulink do Tiriba

A Figura 5.7 mostra a simulação de um modelo do Tiriba gerado pela Pure::variants com o modelo de instância da Figura 5.6. É possível notar que essa execução é menos eficiente em termos de pontos fotografados e em manobras realizadas do que a da Figura 2.11; naquela execução, a MEF da missão havia sido configurada com a seleção de todas as variabilidades das *features* do Tiriba, diferentemente da execução da Figura 5.7, que possui apenas a *feature Failure Handler* da missão selecionada. O fato de a *feature Entry Segment Simulation* não ter sido selecionada para essa simulação foi o principal motivo de as manobras terem sido realizadas de maneira menos eficiente.

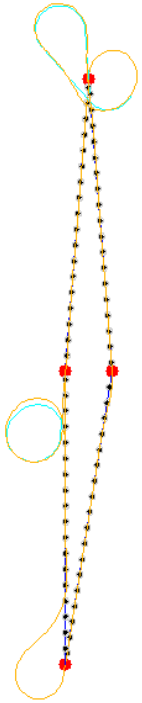


Figura 5.7: Simulação do modelo do Tiriba gerado pela Pure::variants com o modelo de instância da Figura 5.6

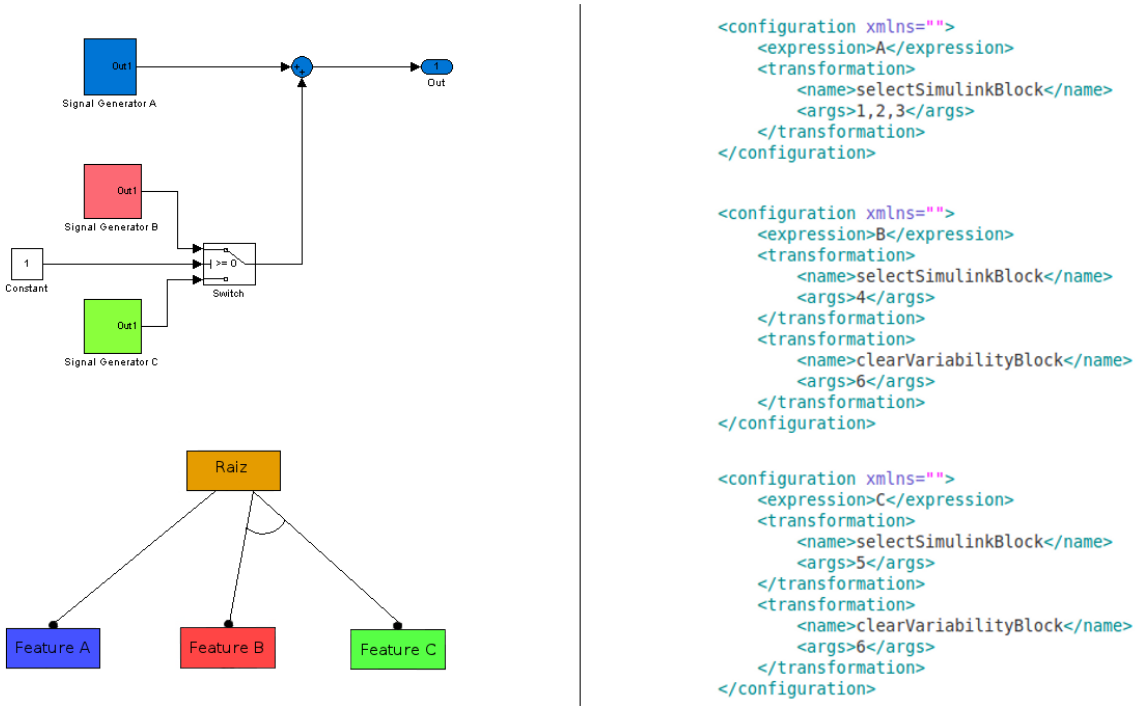
5.3 Extensão do Hephaestus

Hephaestus é uma ferramenta utilizada para apoiar a engenharia de linha de produtos gerando artefatos de instâncias específicas de LPS. A geração de instâncias específicas de uma LPS pela ferramenta se dá por transformações nos ativos da LPS e é baseada em um processo que possui quatro artefatos de entrada: modelo de *features*, modelos de instância que representam a configuração das *features* dos produtos da LPS, ativos da linha de produto e, por último, o conhecimento de configuração, que é responsável por associar expressões de *features* às transformações que devem ser executadas sobre os ativos da LPS. O Hephaestus avalia cada expressão de *features* do conhecimento de configuração a partir das *features* selecionadas do modelo de instância. Todas as transformações associadas às expressões de *features* avaliadas como verdadeiras são aplicadas aos ativos da LPS selecionado para o processo; essas transformações selecionam ou modificam partes dos ativos para o produto que está sendo gerado. Mais detalhes sobre a ferramenta são apresentados na Seção 3.6.1.

A ferramenta Hephaestus foi recentemente estendida como parte deste trabalho para permitir a geração de instâncias de produtos de LPS em Simulink. O conhecimento de configuração definido nessa extensão associa a cada expressão de *feature* um conjunto de duplas, cada qual composta por um identificador de bloco Simulink e uma transformação a ser aplicada a esse bloco. Duas transformações foram implementadas no Hephaesuts:

`selectSimulinkBlock`, que deve ser associada aos identificadores dos blocos que devem ser selecionados para o modelo a ser gerado e `clearVariabilityMechanism`, que deve ser associada a blocos que implementam mecanismos de variabilidades (como os descritos na Seção 4.2).

A Figura 5.8 mostra um exemplo simples de um conhecimento de configuração, modelo de *features* e modelo Simulink de uma pequena LPS (ativo da LPS). Os blocos de id 1, 2 e 3 são os blocos de cor azul, o bloco de id 4 é o de cor magenta, o bloco de id 5 é o de cor verde e o bloco de id 6 é o bloco do tipo `switch` que está sendo usado como mecanismo de variabilidade. É possível notar que apenas um conjunto de transformações das que possuem a expressão de *feature* `B` e `C` será executado, uma vez que essas duas *features* possuem uma relação `ou-exclusiva`. Desse modo, é garantido que a transformação `clearVariabilityMechanism` será executada apenas uma vez sobre o bloco `switch`.




```
<configuration xmlns="">
    <expression>A</expression>
    <transformation>
        <name>selectSimulinkBlock</name>
        <args>1,2,3</args>
    </transformation>
</configuration>

<configuration xmlns="">
    <expression>B</expression>
    <transformation>
        <name>selectSimulinkBlock</name>
        <args>4</args>
    </transformation>
    <transformation>
        <name>clearVariabilityBlock</name>
        <args>6</args>
    </transformation>
</configuration>

<configuration xmlns="">
    <expression>C</expression>
    <transformation>
        <name>selectSimulinkBlock</name>
        <args>5</args>
    </transformation>
    <transformation>
        <name>clearVariabilityBlock</name>
        <args>6</args>
    </transformation>
</configuration>
```

Figura 5.8: Exemplo de um modelo de *features*, conhecimento de configuração e ativo de uma LPS em Simulink

O Hephaestus segue a técnica de variabilidade positiva na geração de seus produtos. A ferramenta começa com um modelo vazio, e por meio de transformações os ativos vão sendo adicionados ou modificados no modelo que está sendo gerado. Para gerar produtos com essa extensão, primeiramente são executadas as transformações `selectSimulinkBlock`, que adicionam à instância do modelo que está sendo gerada todos os blocos que compõem as *features* selecionadas, bem como as linhas que conectam cada um desses blocos. Após executar essas transformações, são executadas as transformações `clearVariabilityMe-`

`chanism`; No estado atual da implementação do Hephaestus, essa transformação obtém do modelo Simulink de entrada as linhas de entrada e saída dos blocos associados a essas transformações (blocos de mecanismo de variabilidade) que são conectadas a blocos que foram selecionados; a partir dessas linhas, a ferramenta cria uma única linha que liga o seu bloco de entrada ao seu bloco de saída. A Figura 5.9 mostra um exemplo de um modelo Simulink antes e depois da aplicação dessa transformação. Nele, o bloco `switch` está associado à transformação `clearVariabilityMechanism` e os blocos `Input Kind A` e `Out1` estão associados à transformação `selectSimulinkBlock` (já tendo sido executada).

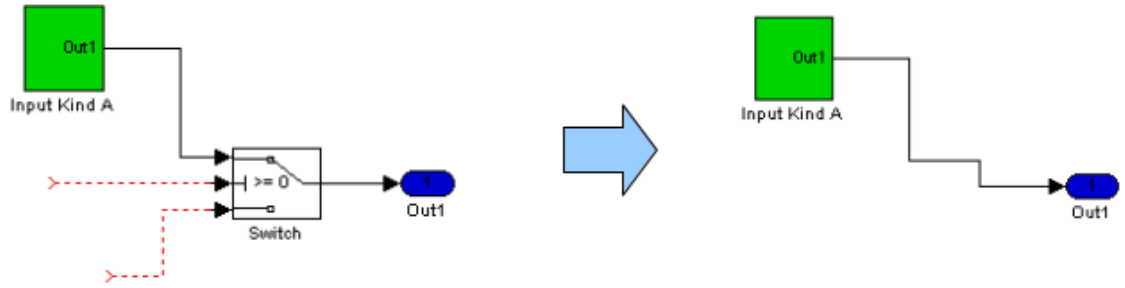


Figura 5.9: Aplicação da transformação `clearVariabilityMechanism` em um modelo Simulink

5.3.1 Implementação da extensão do Hephaestus

Hephaestus foi totalmente desenvolvido com o uso da linguagem de programação funcional Haskell. Essa decisão foi tomada porque a semântica do processo de derivação de produto foi formalizada usando um framework de modelagem transversal cujos interpretadores foram implementados em uma outra linguagem de programação funcional: Schema. Depois de formalizar essas semânticas, foi percebido que Haskell era adequado para o desenvolvimento do Hephaestus porque as funções de ordem superior e aplicação parcial de funções (dois mecanismos que estão na base do projeto de linguagens funcionais) levam a uma implementação simples e elegante dos tipos de conhecimento de configuração e processo de derivação de produto (Bonifácio et al., 2009).

Para implementar essa extensão do Hephaestus foi criada uma biblioteca `Simulink`, um módulo para as transformações dessa extensão na biblioteca de transformações e alguns códigos de adaptação.

A biblioteca `Simulink` contém o módulo do *parser* de modelos Simulink, o módulo para impressão de modelos Simulink e um módulo com os tipos de dados criados para essa implementação. O *parser* da biblioteca foi desenvolvido com auxílio da BNFC (BNF, 2012), uma ferramenta para construção de compiladores que gera o *front-end* de um compilador a partir de uma gramática BNF. Além de Haskell, essa ferramenta auxilia a geração de *parsers* em outras linguagens de programação como Java, C e C++. O módulo para

a impressão de modelos Simulink foi implementado com o auxílio da biblioteca Haskell *Prettyprint*, o que deixou o desenvolvimento mais organizado e fácil de testar. Para essa implementação foram criados alguns tipos de dados principalmente para representarem abstratamente a estrutura de modelos Simulink, tanto para o modelo de entrada (ativo da LPS) como para o modelo de saída (instância de produto da LPS).

A Figura 5.10 mostra alguns dos tipos de dados criados nessa extensão: `Block`, `Line` e `SimulinkModel`, que correspondem, respectivamente à representação abstrata de bloco, linha e modelo Simulink. O `SimulinkModel` possui quatro atributos principais: `modelAtt` (atributos do modelo), `sysAtt` (atributos do sistema), `blocks` (lista de dados do tipo `block`) e `sLines` (lista de dados do tipo `Line`).

```
data SimulinkModel = SimulinkModel {
 modelAtt :: SimulinkAttributes,
 systemAtt :: SimulinkAttributes,
 blocks :: [Block],
 sLines  :: [Line]
} deriving(Show, Data, Typeable)


data Block = Block {
 tpe :: Type,
 bId :: SID,
 name :: Name,
 isVariabilityBlock :: Bool,
 otherAttribs :: SimulinkAttributes
} deriving(Data, Typeable)


data Line =
   SLine Src Dst SimulinkAttributes
 | BLine Src BDst SimulinkAttributes
  deriving(Data, Eq, Typeable)
```

Figura 5.10: Implementação de alguns dos tipos de dados usados na extensão do Hephaestus: bloco, linha e modelo Simulink

Conforme dito no início da Seção 5.3, foram implementadas duas transformações nessa extensão (`selectSimulinkBlock` e `clearVariabilityBlock`) que, juntas, geram instâncias de produtos em LPS escritas em Simulink.

A Figura 5.11 mostra uma visão geral do processo de geração de produtos da abordagem proposta nesta implementação, com os modelos de entrada e saída de cada atividade e transformação. Os modelos de entradas e saídas das transformações são todos dados do tipo `SimulinkModel` (em vermelho).

Ao final da execução das transformações `selectSimulinkBlock`, todos os blocos que deverão ser selecionados para o produto final e suas respectivas linhas serão adicionados ao modelo que está sendo gerado. Para isso, essa transformação usa duas funções principais: `addBlock` e `addLines`. A função `addBlock` percorrerá a lista de blocos da estrutura de

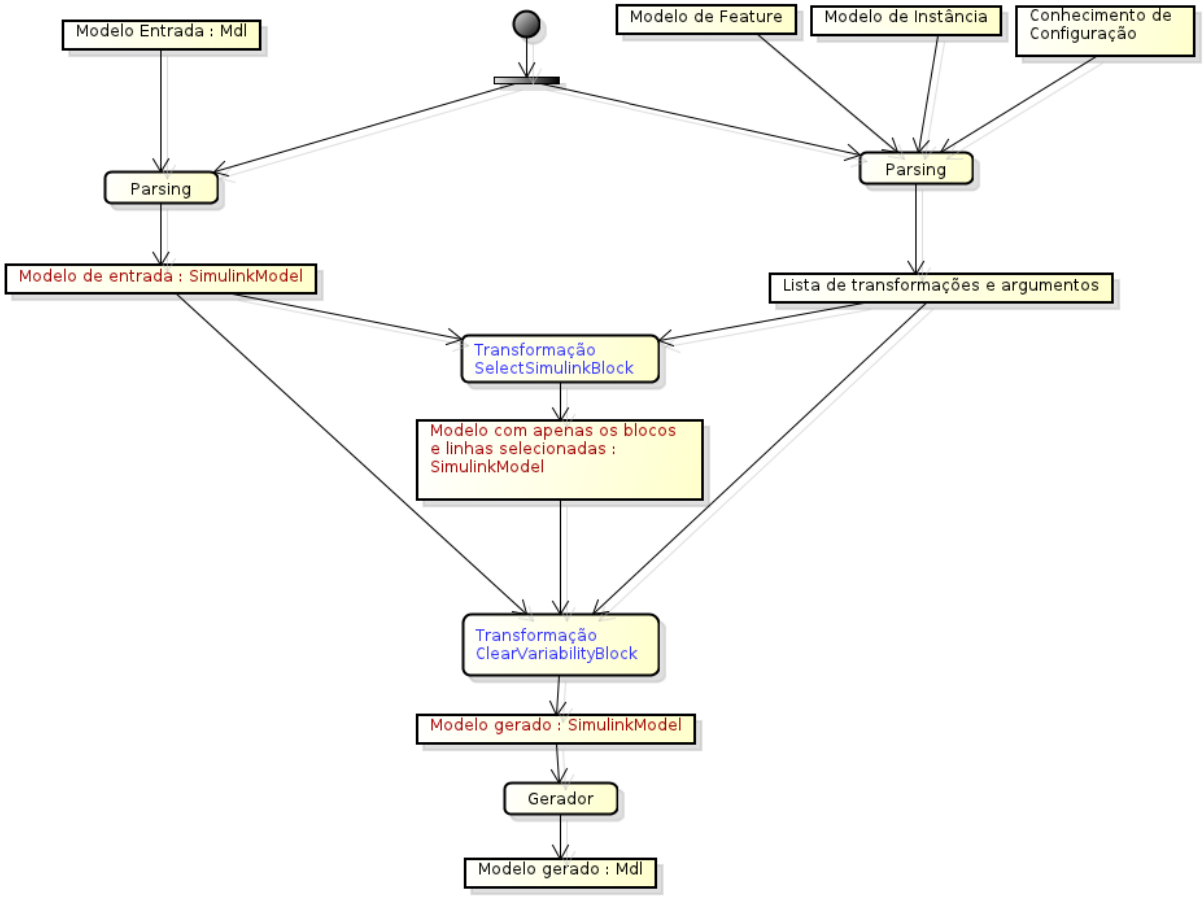


Figura 5.11: Visão geral do processo de geração de produtos da extensão do Hephaestus

dados `SimulinkModel` de entrada do Hephaestus (obtido depois de se executar um *parser* no modelo Simulink de entrada) e comparar o id de cada um de seus blocos com os ids dos blocos associados às transformações `selectSimulinkBlock` do conhecimento de configuração que tiverem sua expressão de *features* validada como verdadeira. A função `addBlock` retornará então um dado do tipo `SimulinkModel` que contém todos os blocos selecionados.

Após a execução da função `addBlock` será executada a função `addLines`. Como pode ser visto na Figura 5.12, essa função recebe dois dados do tipo `SimulinkModel` como parâmetro: o `SimulinkModel`, referente ao modelo Simulink de entrada e o `SimulinkModel`, obtido depois de executar a função `addBlock` (que não possui nenhuma linha). Para cada linha presente no `SimulinkModel` do modelo Simulink de entrada será verificado se tanto a origem como o destino da linha são blocos que foram selecionados (i.e, pertencem ao conjunto de blocos do `SimulinkModel` que foi obtido pela função `addBlock`) e, se o forem, a linha será adicionada ao conjunto de linhas que será retornado pelo função `addLines` e que, posteriormente, será adicionado ao modelo que está sendo gerado.

```
addLines :: SimulinkModel  -- Asset-base model
         -> SimulinkModel  -- Instance model only with selected blocks
         -> [Lines]        -- Selected lines set
addLines (SimulinkModel _ _ _ assetBaseLines) (SimulinkModel _ _ instBlocks _) =
    linesToBeAdded [] assetBaseLines (blockNamesList instBlocks)
```

Figura 5.12: Função `addLines` da extensão do Hephaestus

A transformação `clearVariabilityBlock` é executada por meio da função `clearVariabilityMechanismBlocks`. Essa função recebe como parâmetro uma lista de linhas (do tipo de dado `Line`) do modelo Simulink de entrada e duas listas do tipo de dados `Block`, uma contendo os blocos selecionados para o modelo (os mesmos que fazem parte do modelo retornado pela função `addBlock`) e uma contendo os blocos de variabilidades, que são os blocos cujos ids foram associados às transformações `clearVariabilityMechanismBlocks` do conhecimento de configuração que tiveram sua expressão de *features* validada como verdadeira. Para cada bloco de variabilidade, será procurado na lista de linhas duas linhas: uma que conecte um bloco selecionado à sua porta de entrada e uma que conecte um bloco selecionado à sua porta de saída. Depois, será executada a função `mergeLines`, que realiza uma fusão dessas duas linhas, criando uma linha que terá como origem o bloco ligado à entrada do bloco de variabilidade e como destino o bloco ligado à saída do bloco de variabilidade, resolvendo dessa maneira as variabilidades e deixando no modelo apenas os blocos relacionados às *features* que foram selecionadas. Por fim, será retornada uma lista de linhas contendo apenas as novas linhas criadas pelas funções `mergeLines`

```
clearVariabilityMechanismBlocks :: [Line] -> [Line] -> [Block] -> [Block] -> [Line]
clearVariabilityMechanismBlocks lines noVarLines [] selectedBlocks = lines
clearVariabilityMechanismBlocks lines noVarLines (currentVB:nextVB) selectedBlocks =
    let inLine = getVarBlockInputLine currentVB lines selectedBlocks
        outLine = getVarBlockOutputLine currentVB lines
    in clearVariabilityMechanismBlocks lines (mergeLines noVarLines inLine outLine)
nextVB selectedBlocks
```

Figura 5.13: Função `clearVariabilityMechanismBlocks` da extensão do Hephaestus

A transformação `clearVariabilityBlock` foi implementada com foco nos padrões de mecanismos de variabilidade introduzidos na Seção 4.2; com essa transformação e com a `selectSimulinkBlock` é possível resolver todos esses tipos de variabilidade e gerar instâncias de produtos que contenham apenas blocos relacionados às *features* selecionadas. A única restrição que existe para um mecanismo de variabilidade configurado com a transformação `clearVariabilityBlock` é que deve ser garantido que o Hephaestus aplique a transformação `selectSimulinkBlock` para um bloco conectado à entrada e para um bloco conectado à saída do bloco associado à transformação `clearVariabilityBlock`.

Uma configuração que satisfaça essa restrição não é difícil, pois os padrões de variabilidade já inferem que ao menos um bloco seja selecionado na entrada e um na saída do bloco de variabilidade, caso ela deva ser resolvida.

A Figura 5.14 mostra um exemplo de geração de um produto com o Hephaestus. Os modelos de entrada mostrados no exemplo são modelo Simulink, modelo de *features* e modelo de instância. O conhecimento de configuração é exatamente igual ao mostrado na Figura 5.8. Os modelos intermediários e de saída utilizados nas transformações e em suas funções são dados do tipo `SimulinkModel` e o modelo obtido ao final da aplicação da transformação `clearVariabilityBlock` é transformado em um modelo Simulink pelo gerador implementado (conforme ilustra a Figura 5.11).

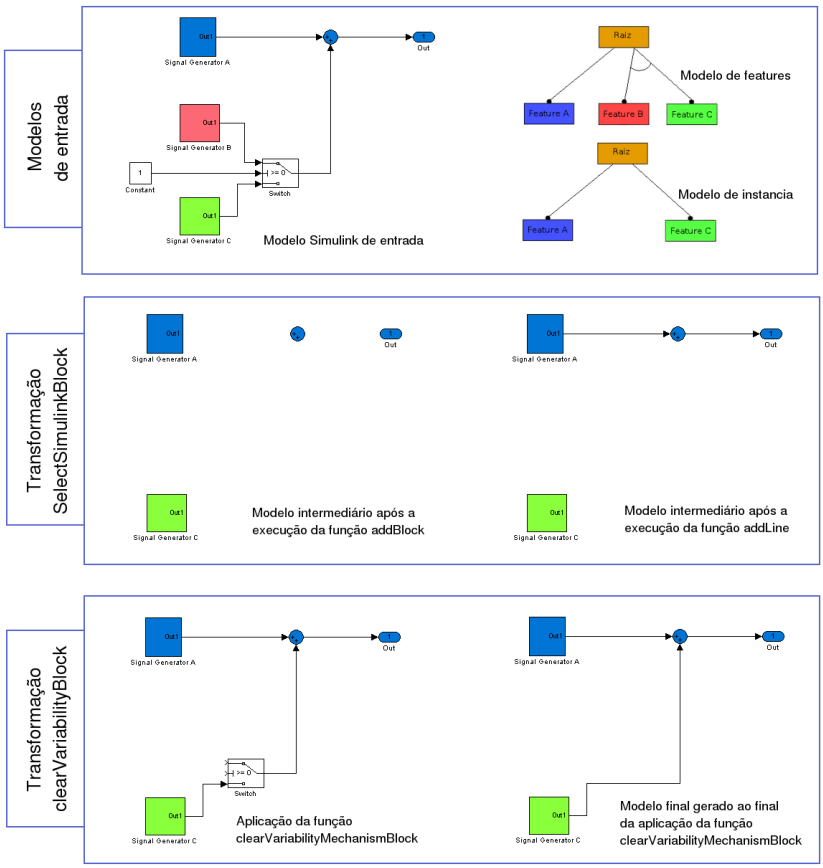


Figura 5.14: Exemplo de geração de um produto com a extensão do Hephaestus: modelos de entrada e dados do tipo `SimulinkModel` modificados passo a passo nas transformações e em suas principais funções

5.3.2 Gerenciamento de configuração no modelo Simulink do Tiriba usando o Hephaestus

Na Seção 3.6.1 foi introduzido em detalhes o funcionamento do Hephaestus. Conforme já foi mencionado, para realizar o gerenciamento de configuração em LPS a ferramenta precisa de quatro modelos como entrada: modelo de *features*, modelo de instância, conhecimento de configuração e ativos da LPS.

Para configurar as nove *features* criadas no Tiriba, foram criadas treze expressões de *features* no conhecimento de configuração. Os ativos do núcleo do Tiriba foram organizados em uma única *feature* obrigatória: *Core*, referente aos blocos obrigatórios do Tiriba. A grande maioria dos blocos do modelo está associada a essa expressão e nela há apenas transformações do tipo `selectSimulinkBlock`.

A *feature Parachute* precisou de apenas uma expressão de *feature* para ser configurada já que ela é uma *feature* opcional e os blocos referentes a essa *feature* estão estruturados de uma maneira que, se ela não for selecionada, isso não torna o modelo inválido (seu conhecimento de configuração pode ser implementado apenas com transformações do tipo `selectSimulinkBlock`). A Figura 5.15 mostra o conhecimento de configuração da *feature Parachute*: Caso a *feature* seja selecionada, 10 blocos deverão ser adicionados ao modelo que será gerado (o id desses 10 blocos estão especificado entre as *tags* `<args>` na figura). Esses blocos são referentes a um subsistema de solenoide e um bloco do tipo `out` (esse bloco cria uma porta de saída no subsistema em que ele estiver) em cada uma das principais unidades do Tiriba (controle, missão, pressão e inercial), um para um multiplexador dos sinais de cada um desses solenoides e um para o subsistema do paraquedas.

```
<configuration xmlns="">
    <expression> Parachute </expression>
    <transformation>
        <name> selectSimulinkBlock </name>
        <args> 13019, 13046, 13052, 13061, 13067,
               13076, 13082, 13091, 13097, 13105 </args>
    </transformation>
</configuration>
```

Figura 5.15: Conhecimento de configuração da *feature Parachute* do Tiriba

Para o conhecimento de configuração das *features Photografic camera* e *Georef log*, foram necessárias configurações com três expressões de *features* diferentes. Isso aconteceu porque existem três possibilidades de combinações de expressões dessas *features* (seleção das duas *features*, apenas da *Photografic camera* e de nenhuma das duas) e em cada caso a variabilidade deve ser resolvida de uma maneira diferente. A Figura 5.16 mostra o conhecimento de configuração dessas *features*.

```
<configuration xmlns="">
    <expression> PhotograficCam && GeorefLog </expression>
    <transformation>
        <name> selectSimulinkBlock </name>
        <args> 12785, 12813, 12817 </args>
    </transformation>
    <transformation>
        <name>clearVariabilityBlock</name>
        <args>12822</args>
    </transformation>
</configuration>

<configuration xmlns="">
    <expression> PhotograficCam && not(GeorefLog) </expression>
    <transformation>
        <name> selectSimulinkBlock </name>
        <args> 12785, 12817 </args>
    </transformation>
    <transformation>
        <name>clearVariabilityBlock</name>
        <args>12822</args>
    </transformation>
</configuration>

<configuration xmlns="">
    <expression> not(PhotograficCam) && not(GeorefLog) </expression>
    <transformation>
        <name>clearVariabilityBlock</name>
        <args>12822</args>
    </transformation>
</configuration>
```

Figura 5.16: Conhecimento de configuração das *features Photografic camera* e *Georef log* do Tiriba

Para cada uma das *features* relacionadas à MEF da missão do Tiriba (*Entry Segment Simulation*, *Failure Hander* e *Feather Threshold Handler*) foram criadas duas configurações de expressões de *features*, uma, caso a *feature* seja selecionada e outra, se não o for. Isso foi feito porque transformações deverão ser aplicadas no modelo tanto se ela for selecionada como se o não for. Se a *feature* for selecionada, um bloco do tipo `constant` com valor verdadeiro (ligado à porta do bloco de máquina de fluxo de estados que atribui o valor à variável da MEF que define a variabilidade) deverá ser selecionado para o modelo, caso contrário deverá ser selecionado um bloco do tipo `constant` com valor falso (que deverá ser ligado a essa mesma porta)

A Figura 5.17 A mostra o bloco de máquina de fluxo de estados (no qual está a MEF da missão do Tiriba). Nela é possível ver as entradas usadas para controlar as variabilidades das *features* implementadas na MEF, os blocos do tipo `constant` usados para atribuir valor a essas entradas e blocos do tipo `switch` usados como mecanismos de variabilidade. Na Figura 5.17 B está o conhecimento de configuração da *feature EntrySegmentSimulation*

(que possui duas configurações de expressões de *feature* para o caso de ser selecionada ou não); as demais *features* relacionadas à MEF (*Failure Hander* e *Feather Threshold Handler*) possuem o mesmo formato no conhecimento de configuração.

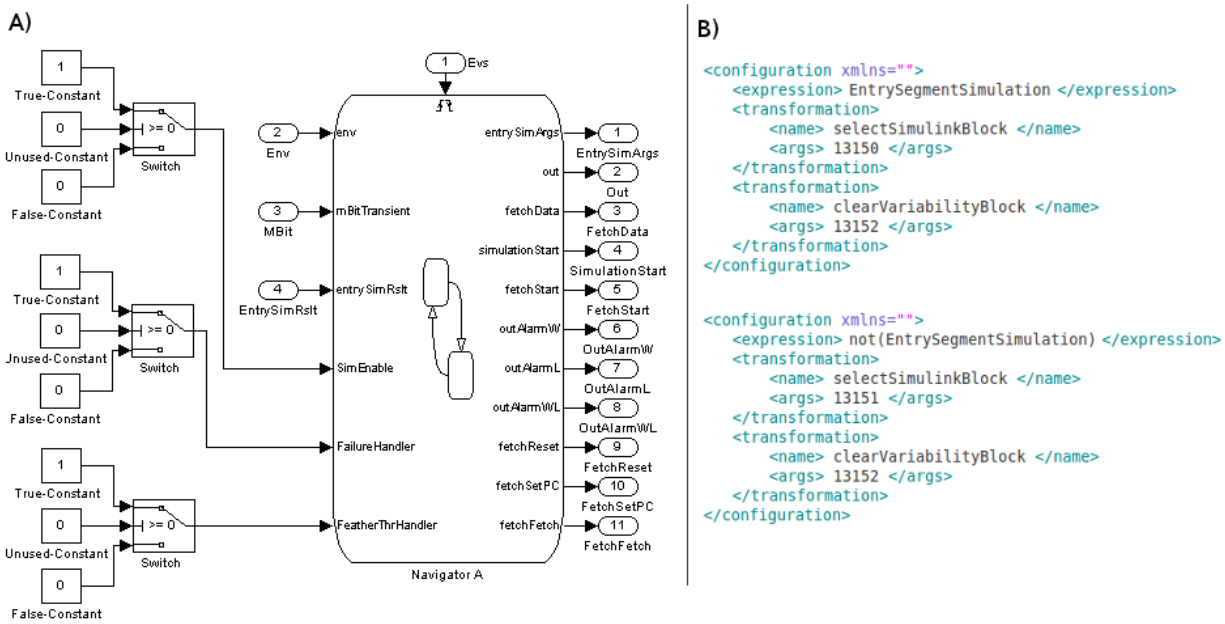


Figura 5.17: Blocos `switch` usados como mecanismos de variabilidade para configurar corretamente as *features* da MEF (A) e conhecimento de configuração da *feature* *EntrySegmentSimulation* do Tiriba (B)

As *features* do motor (*Engine* e *Combustion*) foram configuradas cada uma com uma expressão de *feature* no conhecimento de configuração. O fato de elas serem *features* com relação `ou-exclusivo` não interferiu no conhecimento de configuração, uma vez que a seleção de apenas uma delas deverá ser garantida pelo modelo de *features* na entrada do Hephaestus e haverá um erro caso nenhuma delas ou as duas sejam selecionadas. A Figura 5.18 mostra o conhecimento de configuração das *features* do motor. Essas *features* foram implementadas com dois mecanismos de variabilidade, pois os blocos referentes ao motor estão em duas áreas espaciais distintas no modelo do Tiriba conforme ilustram as figuras 4.16 (parâmetros para velocidade dos motores) e 4.17 (níveis de suavidade para aceleração e desaceleração dos motores). É possível ver que a *feature* *Electric* possui menos blocos do que a *Combustion*; isso ocorre porque o seu comportamento é mais simples.

A Figura 5.19 mostra o modelo de *features* do Tiriba feito no padrão do Hephaestus. Esse padrão possui dois tipos de *tags*: `feature` e `featureGroup`. A *tag* `feature` declara uma *feature* obrigatória ou opcional e com ela também é possível compor algumas relações como de hierarquia e dependência. Essa *tag* possui dois atributos que são usados para definir o tipo de uma *feature*: `min` e `max`; Se o atributo `min` de uma determinada *feature* for 1, significa que a *feature* deverá ser necessariamente selecionada (caso contrário a

```
<configuration xmlns="">
    <expression> Combustion </expression>
    <transformation>
        <name> selectSimulinkBlock </name>
        <args> 13458, 13477, 9161 </args>
    </transformation>
    <transformation>
        <name> clearVariabilityBlock </name>
        <args> 13455, 13492 </args>
    </transformation>
</configuration>


<configuration xmlns="">
    <expression> Electric </expression>
    <transformation>
        <name> selectSimulinkBlock </name>
        <args> 13412 </args>
    </transformation>
    <transformation>
        <name> clearVariabilityBlock </name>
        <args> 13455, 13492 </args>
    </transformation>
</configuration>
```

Figura 5.18: Conhecimento de configuração das *feature* do motor (*Engine* e *Combustion*) do Tiriba

configuração será inválida) de modo que isso a torna uma *feature* obrigatória. Se o `min` de uma *feature* for 0, significa que ela pode não ser selecionada para um produto, sendo, desse modo, uma *feature* opcional. No modelo de *features* da Figura 5.19 é possível ver uma feature obrigatória (`Core`) e algumas opcionais (como, por exemplo, `Parachute`). É também possível ver dois exemplos de relação hierárquica: todas as *features* que estão entre a *tag* de início e fim da feature `Core` e as *features* `Photografic Camera` e `Georef Logger`.

A *tag* `featureGroup` é usada para criar agrupamentos de *features* e explicitar relações entre elas. No modelo de *features* do Tiriba ela foi usada para modelar as *features* do motor (que possuem uma relação do tipo `ou-exclusivo`) pelo `featureGroup Engine`. É possível ver que os atributos `min` e `max` do agrupamento possuem ambos valor 1; isso significa que uma, e apenas uma das *features* do grupo (`Combustion` ou `Electric`) deverá ser selecionada.

O modelo de instância do Hephaestus possui um padrão parecido com o seu modelo de *features*, exceto que não existem *tags* do tipo `featureGroup` e nem atributos do tipo `min` e `max`. Isso acontece porque essa *tag* e esses atributos são usados para definir restrições dos modelos de *features* de modo que seria desnecessário duplicar essas informações nas *features* selecionadas para o modelo de instância. A Figura 5.20 mostra um exemplo de um modelo de instância do Tiriba no padrão do Hephaestus.

```
<feature min="1" max="1" name="Core" type="NONE" id="Core">

   <featureGroup min="1" max="1" id="Engine">
       <feature min="0" max="1" name="Combustion" type="NONE" id="Combustion">
       </feature>
       <feature min="0" max="1" name="Electric" type="NONE" id="Electric">
       </feature>
   </featureGroup>

   <feature min="0" max="1" name="Photografic Camera" type="NONE" id="PhotograficCam">
       <feature min="0" max="1" name="Georef Logger" type="NONE" id="GeorefLog">
       </feature>
   </feature>

   <feature min="0" max="1" name="Parachute" type="NONE" id="Parachute">
   </feature>

   <feature min="0" max="1" name="Entry Segment Simulation" type="NONE" id="EntrySegmentSimulation">
   </feature>

   <feature min="0" max="1" name="Failure Hander" type="NONE" id="FailureHander">
   </feature>

   <feature min="0" max="1" name="Feather Threshold Handler" type="NONE" id="FeatherThrHandler">
   </feature>

</feature>
```

Figura 5.19: Modelo de features do Tiriba no padrão do Hephaestus

A Figura 5.21 mostra a imagem resultante da execução da simulação de um modelo do Tiriba gerado pelo Hephaestus com as *features* selecionadas pelo modelo de instância da Figura 5.20. Nessa execução a aeronave realiza menos manobras do que na execução da Figura 5.7 e equivalentes às da Figura 2.11, mas em termos de pontos fotografados é menos eficiente do que as das figuras 2.11 e 5.7. Isso ocorre porque a *feature Failure Handler* da missão não foi selecionada, então a aeronave fica com um pouco mais de flexibilidade em relação a perdas de pontos de fotografia.

5.4 Comparação entre as ferramentas Pure::variants e Hephaestus

As duas ferramentas são diferentes no que diz respeito à geração de uma instância de produto de uma LPS. A Pure::variants utiliza uma abordagem de variabilidade negativa; uma instância de produto de uma LPS cuja variabilidade é gerenciada por essa ferramenta sempre conterá os blocos relacionados a todas as *features* da LPS (mesmo as não selecionadas) e todos os blocos usados como mecanismos de variabilidade. A geração de produtos se dá por meio de modificação no valor de variáveis do *workspace* do MATLAB presentes em alguns blocos Simulink que controlam mecanismos de variabilidade dentro do modelo; esses blocos de controle desativam os blocos relacionados às *features* que não

```
<feature name="Core" type="NONE" id="Core">

    <feature name="Electric" type="NONE" id="Electric">
    </feature>

    <feature name="Photografic Camera" type="NONE" id="PhotograficCam">
    </feature>

    <feature name="Parachute" type="NONE" id="Parachute">
    </feature>

    <feature name="Feather Threshold Handler" type="NONE" id="FeatherThrHandler">
    </feature>

</feature>
```

Figura 5.20: Exemplo de um modelo de *features* do Tiriba no padrão do Hephaestus

foram selecionadas, fazendo com que somente sejam executados os blocos relacionados às *features* selecionadas. O valor que pode ser atribuído a cada uma dessas variáveis pertence a um conjunto de possibilidades pré-definido nos pontos de variação criados no próprio modelo Simulink. A Pure::variants realiza, então, um mapeamento de expressões de *features* (do modelo de *features* da LPS) para cada um desses valores. Com esse mapeamento, a Pure::variants pode propagar a configuração de qualquer modelo de instância válido para os valores definidos para cada um dos pontos de variação.

A abordagem do Hephaestus para gerar produtos de uma LPS utiliza a técnica de variabilidade positiva. Ao gerar um produto, o Hephaestus começa com um modelo vazio. Inicialmente ele adiciona os atributos Simulink a esse modelo e depois adiciona, por meio de transformações, os blocos associados às *features* selecionadas para o produto que está sendo gerado. Cada transformação a ser executada é definida no conhecimento de configuração da LPS, que relaciona *features* e expressões de *features* da LPS a um conjunto de transformações que traduzem ativos da LPS em artefatos específicos de produtos. Depois disso, a ferramenta adiciona todas as linhas do modelo que possuam um bloco de origem e ao menos um bloco de destino selecionado (removendo as ligações com os blocos de origem ou destino não selecionados). Por último, a ferramenta limpa os mecanismos de variabilidade definidos no modelo, para que o modelo gerado possua apenas os blocos de variantes funcionais relacionadas às *features* selecionadas.

A tabela 5.1 sintetiza as principais diferenças entre as duas ferramentas.

A geração de código em LPS cujas variabilidades são gerenciadas pela Pure::variants é menos eficiente do que as gerenciadas pelo Hephaestus. Conforme dito anteriormente, modelos de produtos gerados pelo Hephaestus contêm apenas os blocos associados às *features* selecionadas; sendo assim, o código gerado a partir deles conterá apenas partes referentes aos blocos selecionados. No caso da Pure::variants, como os modelos de produtos de LPS contêm sempre todos os blocos, o código gerado para todos os produtos será sempre praticamente o mesmo, mudando apenas o valor de algumas variáveis que fazem com que

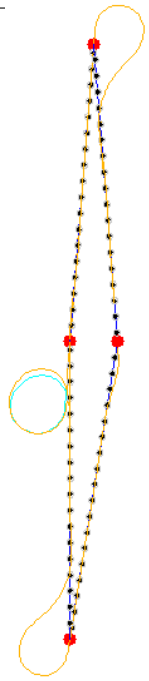


Figura 5.21: Simulação de uma instância de produto da LPS do Tiriba gerada pelo Hephaestus

Características	**Pure::variants**	**Hephaestus**
Depende do MATLAB/-Simulink para funcionar	Sim	Não
Tipo de variabilidade na geração dos produtos	Negativa	Positiva
Processo de geração de produtos	Baseada em blocos de controle	Baseado em transformações para geração de modelos
Blocos do modelo gerado	Contém todos os blocos da LPS, porém só executa os blocos associados às *features* que forem selecionadas ao modelo	Contém apenas os blocos que fazem parte das *features* selecionadas para o modelo
Requer estruturação no modelo Simulink da LPS	Necessário para todos os casos	Nem sempre é necessário
Conhecimento de configuração	Parte está no modelo de variabilidade e parte está no modelo Simulink	Fica em um único arquivo XML contém todas as informações necessárias
Geração de código	Gera código referente a todos os blocos do modelo (mesmo dos não selecionados para o produto)	Gera código referente apenas aos blocos selecionados para o produto

Tabela 5.1: Principais características das ferramentas Hephaestus e Pure::variants

apenas as variabilidades selecionadas no código sejam executadas (o que faz com que boa parte do código gerado seja código morto). Essas variáveis são geradas a partir dos blocos de controle do modelo cujos valores foram definidos pela Pure::variants.

Essa é a principal vantagem do Hephaestus sobre a Pure::variants. No domínio de sistemas embarcados, normalmente a memória dos programas é pequena e em muitos casos proibitiva e, portanto, a adição de código morto nesses sistemas é uma falha de projeto grave. Para contornar esse problema, os engenheiros de aplicação que queiram gerar código de algum produto cuja LPS está tendo sua variabilidade gerenciada pela Pure::variants têm duas opções: manualmente verificar no código gerado os pontos de variação e fazer as alterações adequadas no código para manter apenas as partes referentes às variabilidades selecionadas ou editar o modelo Simulink, removendo manualmente todos os blocos que não fazem parte do modelo (inclusive os blocos de variabilidade) que se está querendo gerar e gerar código a partir desse modelo editado; nesse caso, esse modelo editado deixaria de ter a sua variabilidade gerenciado pela Pure::variants. Essas duas opções exigem um grande esforço e são propensas a gerar erros, pois são atividades manuais e os modelos podem ser bastante complexos.

A construção e evolução do conhecimento de configuração das duas ferramentas é um processo simples. No entanto, Hephaestus exige mais esforço, porque o identificador de cada um dos blocos deverá ser encontrado (na representação textual do modelo) e associado à expressão de *features* responsável por executar uma transformação sobre ele. Essa é a principal vantagem da Pure::variants sobre o Hephaestus. Evoluir uma LPS com a Pure::variants é muito mais fácil e intuitivo, uma vez que a variabilidade é denotada no próprio modelo.

Depois de se construir o conhecimento de configuração, a geração de instâncias da linha de produto é um processo rápido, tanto para a Pure::variants como para o Hephaestus. No caso da Pure::variants, depois de se criar o modelo de instância basta "propagar" a sua configuração para o modelo com apenas um comando. No Hephaestus, depois de se criar um modelo de *feature*, um modelo de instância e o conhecimento de configuração, basta rodar o processo de geração de produto, que tem esses três artefatos e mais os ativos da LPS como entrada.

A tabela 5.2 mostra alguns dados coletados na configuração da LPS criada a partir do modelo Simulink do Tiriba para as duas ferramentas. É possível ver que a ferramenta Pure::variants demandou um número de tokens muito menor para organizar o seu conhecimento de configuração. Isso ocorre porque a ferramenta automatiza a criação e edição do conhecimento de configuração por meio de uma interface gráfica que faz modificações no modelo de variantes. Até o atual estado da implementação do Hephaestus ainda não houve nenhum esforço nesse sentido, de modo que o engenheiro de aplicação que usar a ferramenta deverá fazer manualmente o conhecimento de configuração seguindo o formato XML da ferramenta. No entanto, a ideia de automatizar a criação e edição do seu conhecimento de configuração é perfeitamente possível.

Características	**Pure::variants**	**Hephaestus**
Número de mecanismos de variabilidade usados	Foram usados seis mecanismos de variabilidade (três para as *features* do *payload*, um para o paraquedas, nenhum para as *features* da missão e dois para as *features* do motor)	Foram usados seis mecanismos de variabilidade (um para as *features* do *payload*, nenhum para o paraquedas, três para as *features* da missão e dois para as *features* do motor)
Número de tokens usados para especificar o conhecimento de configuração	40	718
Configuração do conhecimento de configuração	Treze expressões de *feature* para atribuir valores em seis pontos de variação	Treze expressões de *feature* associadas a vinte e três transformações

Tabela 5.2: Algumas características das ferramentas Hephaestus e Pure::variants observadas durante a configuração do modelo do Tiriba

5.5 Considerações finais

Neste capítulo foram apresentadas as extensões das ferramentas Pure::variants e Hephaestus para apoiar o gerenciamento de configuração em LPS modeladas em Simulink. Foi também mostrado como a LPS criada a partir do modelo Simulink do Tiriba foi configurada em cada uma das ferramentas e, por fim, foi apresentada uma comparação entre as duas ferramentas.

Capítulo

6

Conclusão

6.1 Considerações Finais

Nesta dissertação foi realizada uma prova de conceito sobre como um modelo de sistema embarcado em Simulink pode ser estendido e configurado como uma LPS. O modelo Simulink utilizado neste trabalho é uma versão de simulação do VANT Tiriba, desenvolvido em parceria entre a empresa AGX (AGX, 2012) e o INCT-SEC (INCT-SEC, 2012).

A partir do estudo do modelo do Tiriba e de conversas com especialistas e desenvolvedores do domínio, uma linha de produtos com oito *features* variáveis pôde ser desenvolvida. Apesar de o número de *features* ser pequeno, várias características interessantes puderam ser agregadas a essa LPS, como *features* opcionais, obrigatórias alternativas, *features* com relação de hierarquia e *features* cujas variantes funcionais se encontram em máquinas de estados.

Neste trabalho foi realizada também uma implementação seguindo abordagem nova para a geração de produtos em LPS em Simulink. Essa abordagem foi implementada na forma de uma extensão da ferramenta Hephaestus, que já possui uma estrutura completa de avaliação de modelos de instância e modelos de *features*, bem como uma estrutura de avaliação de expressões de *features* relacionadas a transformações que devem ser aplicadas para gerar instâncias de modelos. Com essa implementação, o Hephaestus passou a gerar modelos Simulink com apenas blocos e linhas referentes a variantes funcionais selecionadas pelos modelos de instância de *features* na entrada do seu processo de geração de produtos.

Por fim, a LPS criada a partir do modelo Simulink do Tiriba foi configurada com a extensão feita no Hephaestus e com a ferramenta comercial Pure::variants (com o conector para Simulink). Foi realizada uma comparação das duas ferramentas considerando as principais características gerais e as características observadas na configuração da LPS criada neste trabalho.

6.2 Contribuições

Neste trabalho foi mostrado como linhas de produtos em Simulink podem ser configuradas. Um conjunto de padrões para implementar variabilidades de *features* de diferentes tipos e com diferentes relações foi introduzido. Além disso, foi mostrado como as ferramentas Pure::variants e Hephaestus (com a extensão para Simulink) podem ser usadas para realizar o gerenciamento de configuração em LPS modeladas em Simulink.

A ferramenta Hephaestus foi estendida neste trabalho. Com esssa implementação, a ferramenta passou a apoiar a fase de engenharia de produto no desenvolvimento de LPS em Simulink.

Também foi mostrado como uma LPS em Simulink pode ser criada e configurada com *features* de diferentes tipos e diferentes relações. Para isso, foi criado de fato uma pequena LPS, com base no modelo Simulink do VANT Tiriba. Os padrões de implementação de variabilidade propostos neste trabalho foram usados incialmente para estruturar as variabilidades das *features* no modelo Simulink da LPS e posteriormente essa LPS foi configurada com as ferramentas Pure::variants (com o conector pra Simulink) e Hephaestus (com a extensão implementada).

Uma comparação entre a Pure::variants e a Hephaestus foi realizada após configurar a LPS criada neste trabalho com ambas as ferramentas. As principais características gerais, vantagens e desvantagens das ferramentas foram avaliadas bem como as características específicas que puderam ser observadas na configuração da LPS criada.

6.3 Trabalhos Futuros

Como trabalhos futuros, pode-se aplicar os padrões propostos para implementação de variabilidades e as ferramentas usadas na prova de conceito em uma LPS real. Essa LPS real pode ser feita com base no modelo Simulink de outros VANTs, além do Tiriba, usando a abordagem extrativa de adoção de LPS. Da parceria entre o INCT-SEC e a empresa AGX existem outros projetos de VANTs, alguns que já foram concluídos (por exemplo, o VANT ARARA) e outros que estão em curso (como SarVANT e o SGA3). O desenvolvimento do software de todos esses VANTs foi ou está sendo feito no ambiente

MATLAB/Simulink e boa parte da arquitetura deles é semelhante, de modo que o esforço empregado para extrair uma LPS a partir desses modelos não será muito grande.

Um segundo trabalho futuro é referente à automatização da construção do conhecimento de configuração na extensão da ferramenta Hephaestus para Simulink. Na extensão do Hephaestus, a construção do conhecimento de configuração exige um esforço muito grande tanto na configuração inicial da LPS quanto na sua evolução; isso ocorre porque o engenheiro de aplicação deve buscar na representação textual do modelo Simulink de entrada o id de cada um dos blocos Simulink que devem fazer parte do conhecimento de configuração. Uma maneira de automatizar isso seria gerar uma representação gráfica do modelo Simulink (exatamente igual à gerada pelo ambiente MATLAB/Simulink) e criar uma interface gráfica que possibilite a seleção de expressões de *features* e associá-las a transformações (também selecionadas) sobre os blocos; ao construir o conhecimento de configuração, bastaria o engenheiro de aplicação clicar nos blocos para a ferramenta capturar automaticamente seus ids.

Referências

Agostino, S.; Mammone, M.; Nelson, M.; Zhou, T. *Classification of Unmanned Air Vehicles.* Relatório Técnico, Aeronautical Engineering Department, University of Adelaide, Australia, 2006.

AGX Tecnologia Ltda. (Acessado em 10/02/2012), 2012. Disponível em `http://www.agx.com.br`

Airforce -technology.com. (Acessado em 10/02/2012), 2010. Disponível em `http://www.airforce-technology.com/projects/predator/predator6.html`

Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E.; Laitenberger, O.; Laqua, R.; Muthig, D.; Paech, B.; Wust, J.; J., Z. *Component-based product line engineering with UML.* Addison-Wesley Professional, 2002.

Balasubramanian, K.; Gokhale, A.; Karsai, G.; Sztipanovits, J.; Neema, S. Developing applications using model-driven design environments. *Computer*, v. 39, n. 2, p. 33–40, 2006.

Bayer, J.; Flege, O.; Knauber, P.; Laqua, R.; Muthig, D.; Schmid, K.; Widen, T.; DeBaud, J. PuLSE: A methodology to develop software product lines. In: *Proceedings of the 1999 symposium on Software reusability*, ACM, 1999, p. 122–131.

Beuche, D. *Variant Management with pure:: variants.* Relatório Técnico, Pure-systems GmbH, 2003.

Beuche, D. Modeling and building software product lines with pure:: variants. In: *Software Product Line Conference, 2008. SPLC'08. 12th International*, IEEE, 2008, p. 358.

Bhat, S.; Malagi, V.; Rangarajan, K.; Babu, R. Computer vision based guidance in UAVs: software engineering challenges. *ACM SIGSOFT Software Engineering Notes*, v. 35, n. 6, p. 1–6, 2010.

BNF Converter. (Acessado em 10/02/2012), 2012. Disponível em `http://www.cse.chalmers.se/research/group/Language-technology/BNFC/`

Bonifácio, R.; Borba, P. Modeling Scenario Variability as Crosscutting Mechanisms. In: *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, ACM, 2009, p. 125–136.

Bonifácio, R.; Teixeira, L.; Borba, P. Hephaestus A Tool for Managing SPL Variabilities. *III Brazilian Symposium on Components, Architectures and Reuse (SBCARS)*, p. 26–34, 2009.

Botterweck, G.; Polzer, A.; Kowalewski, S. Using Higher-order Transformations to Derive Variability Mechanism for Embedded Systems. *Models in Software Engineering*, p. 68–82, 2010.

Bouhraoua, A.; Merah, N.; AlDajani, M.; ElShafei, M. Design and Implementation of an Unmanned Ground Vehicle for Security Applications. In: *2010 7th International Symposium on Mechatronics and its Applications (ISMA)*, 2010, p. 1–6.

Braga, R.; Branco, K.; Junior, O.; Gimenes, I. Evolving Tiriba Design towards a Product line of Small Electric-Powered UAVs. In: *1st Brazilian Conference on Critical Embedded Systems*, 2011, p. 63–72.

Branco, K.; Pelizzoni, J.; Neris, L.; Trindade, O.; Osorio, F.; Wolf, D. Tiriba-a new approach of UAV based on model driven development and multiprocessors. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, IEEE, 2011, p. 1–4.

Breivold, H.; Larsson, S.; Land, R. Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies. In: *Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference*, IEEE, 2008, p. 232–239.

Buhrdorf, R.; Churchett, D.; Krueger, C. Salion's Experience with a Reactive Software Product Line Approach. *Software Product-Family Engineering*, p. 317–322, 2004.

Bunse, C.; Gross, H.; Peper, C. Applying a Model-based Approach for Embedded System Development. In: *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, IEEE, 2007, p. 121–128.

Catal, C. Barriers to the Adoption of Software Product line Engineering. *ACM SIGSOFT Software Engineering Notes*, v. 34, n. 6, p. 1–4, 2009.

Cetina, C.; Giner, P.; Fons, J.; Pelechano, V. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes (HTML). *Computer*, v. 42, n. 10, p. 46–52, 2009.

Chong, S. *An Evaluation Report for Three Product-Line Tools (Form, Pure:: Variants and Gear)*. Relatório Técnico, Jet Propulsion Laboratory, California Institute of Technology, and NASA Ames Research Center, 2008.

Cirilo, E.; Kulesza, U.; Lucena, C. Genarch: a model-based product derivation tool. In: *Proceedings of the First Brazilian Symposium on Components, Architecture and Reuse*, 2007, p. 31–44.

Clements, P.; Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley Reading MA, 2001.

Dabney, J.; Harman, T. *Mastering simulink*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

Decker, S.; Dager, J. Software Product Lines Beyond Software Development. In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*, IEEE, 2007, p. 275–280.

Dziobek, C.; Loew, J.; Przystas, W.; Weiland, J. Functional variants handling in simulink models. In: *MathWorks Virtual Automotive Conference, Stuttgart*, 2008.

Ebert, C.; Jones, C. Embedded software: Facts, figures, and future. *Computer*, v. 42, n. 4, p. 42–52, 2009.

Ebert, C.; Salecker, C. Guest Editors Introduction: Embedded Software Technologies and Trends. *IEEE Software*, v. 26, n. 3, p. 14–18, 2009.

El Kaim, W. System Family Software Architecture Glossary. *Consortium-Wide Deliverable, ESAPS Project*, p. 1–19, 2000.

Faust, D.; Verhoef, C. Software Product Line Migration and Deployment. *Software: Practice and Experience*, v. 33, n. 10, p. 933–955, 2003.

Fragal, V.; Junior, E.; Gimenes, I. Mapping Software Product Line Features to Unmanned Aerial Vehicle Models. In: *1st Brazilian Conference on Critical Embedded Systems*, 2011, p. 49–54.

Gomaa, H.; Shin, M. Tool Support for Software Variability Management and Product Derivation in software Product Lines. In: *Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC), Boston, USA*, 2004, p. 73–84.

Groher, I.; Voelter, M. Expressing feature-based variability in structural models. In: *Workshop on Managing Variability for Software Product Lines*, 2007.

Harsu, M. *FAST Product-line Architecture Process.* Relatório Técnico, Software Systems Laboratory, Tampere University of Technology, 2002.

INCT-SEC Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos. (Acessado em 10/02/2012), 2012.
Disponível em `http://www.inct-sec.org`

Júnior, C.; Masiero, P.; Braga, R. Captor-AO: Gerador de Aplicações apoiado pela Programação Orientada a Aspectos. *II Brazilian Symposium on Components, Architectures and Reuse (SBCARS)*, p. 1–8, 2008.

Kang, K.; Cohen, S.; Hess, J.; Novak, W.; Peterson, A. *Feature-oriented domain analysis (FODA) feasibility study.* Relatório Técnico, Carnegie-Mellon University – Pittsburgh, PA, 1990.

Kang, K.; Lee, J.; Donohoe, P. Feature-oriented Product Line Engineering. *Software, IEEE*, v. 19, n. 4, p. 58–65, 2002.

Karsai, G.; Massacci, F.; Osterweil, L.; Schieferdecker, I. Evolving Embedded Systems. *Computer*, p. 34–40, 2010.

Kästner, C.; Apel, S.; Kuhlemann, M. Granularity in Software Product Lines. In: *Proceedings of the 30th International Conference on Software Engineering*, ACM, 2008, p. 311–320.

Kopetz, H. *Real-time systems: design principles for distributed embedded applications.* Springer, 1997.

Krueger, C. Easing the Transition to Software Mass Customization. *Software Product-Family Engineering*, p. 178–184, 2002a.

Krueger, C. Eliminating the Adoption Barrier. *IEEE Software*, v. 19, n. 4, p. 29–31, 2002b.

Krueger, C. Variation Management for Software Production Lines. *Software Product Lines*, p. 107–108, 2002c.

Lee, E. The Future of Embedded Software. (Acessado em 10/01/2011), 2006. Disponível em `http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt`

Li, Q.; Yao, C. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

Liggesmeyer, P.; Trapp, M. Trends in Embedded Software Engineering. *IEEE Software*, p. 19–25, 2009.

Lutz, R. Enabling Verifiable Conformance for Product Lines. In: *12th International Software Product Line Conference*, IEEE, 2008, p. 35–44.

Marwedel, P. *Embedded System Design*. Kluwer Academic Pub, 2003.

MathWorks Real-Time Workshop Data Sheet. (Acessado em 02/02/2012), 2007. Disponível em `http://www.mathworks.com/products/rtw/`

MathWorks MATLAB/Simulink. (Acessado em 02/02/2012), 2012. Disponível em `www.mathworks.com/products/simulink`

Muller, G. *Opportunities and Challenges in Embedded Systems*. Relatório Técnico, Embedded Systems Institute, TU/e Eindhoven University of Technology, 2003.

Muthig, D.; Patzke, T. Generic implementation of product line components. *Objects, Components, Architectures, Services, and Applications for a Networked World*, p. 313–329, 2009.

Myllymäki, T. *Variability management in software product lines*. Tampere University of Technology, 2002.

Neris, L. *Um piloto automático para as aeronaves do projeto ARARA*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 2001.

Northrop, L.; Clements, P.; Bachmann, F.; Bergey, J.; Chastek, G.; Cohen, S.; Donohoe, P.; Jones, L.; Krut, R.; Little, R.; et al. A Framework for Software Product Line Practice, version 5.0. (Acessado em 10/01/2011), 2007. Disponível em `http://www.sei.cmu.edu/productlines/framework.html`

Obbink, H.; Müller, J.; America, P.; van Ommering, R.; Muller, G.; van der Sterren, W.; Wijnstra, J. COPA; a Component-oriented Platform Architecting Method for Families

of Software-intensive Electronic Products. In: *Tutorial for the First Software Product Line Conference, Denver, Colorado*, 2000.

OMG Model Driven Architecture. (Acessado em 02/02/2012), 2005. Disponível em `http://www.omg.org/mda`

Pastor, E.; López; Royo, P. A hardware/software architecture for UAV payload and mission control. *25th Digital Avionics Systems Conference IEEE/AIAA*, p. 5B41–5B48, 2006.

Pohl, K.; Böckle, G.; Van Der Linden, F. *Software product line engineering: foundations, principles, and techniques.* Springer-Verlag New York Inc, 2005.

Pohl, K.; Metzger, A. Variability management in software product line engineering. In: *Proceedings of the 28th International Conference on Software Engineering*, ACM, 2006, p. 1049–1050.

Polzer, A.; Kowalewski, S.; Botterweck, G. Applying Software Product Line Techniques in Model-based Embedded Systems Engineering. In: *6th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, Vancouver, CA, 2009, p. 2–10.

Sanz, R.; López, I.; Hernández, C. Self-awareness in real-time cognitive control architectures. *AI and Consciousness: Theoretical Foundations and Current Approaches*, 2007.

Schmid, K. A quantitative model of the value of architecture in product line adoption. *Software Product-Family Engineering*, p. 32–43, 2004.

Spencer, H.; Collyer, G. # ifdef considered harmful, or Portability Experience with C News. *In USENIX Conf*, p. 185–198, 1992.

Sugumaran, V.; Park, S.; Kang, K. Software product line engineering. *Communications of the ACM*, v. 49, n. 12, p. 28–32, 2006.

Svahnberg, M.; Mattsson, M. Conditions and restrictions for product line generation migration. *Software Product-Family Engineering*, p. 211–235, 2002.

Torres, M.; Kulesza, U.; Braga, R. T. V.; Masiero, P. C.; Pires, P. F.; Cirilo, F. D. E.; Batista, T. V.; Teixeira, L.; Borba, P.; Lucena, C. J. P. Estudo Comparativo de Ferramentas de Derivação Dirigidas por Modelos: Resultados Preliminares. *I Brazilian Workshop on Model-Driven Development*, 2010.

Van Der Linden, F.; Schmid, K.; Rommes, E. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering.* Springer-Verlag New York Inc, 2007.

Yoshimura, K.; Ganesan, D.; Muthig, D. Defining a Strategy to Introduce a Software Product Line Using Existing Embedded Systems. In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, ACM, 2006, p. 72.