RAFAEL GARLET DE OLIVEIRA

Contribuições para Melhoria do Processo de Verificação Formal de Propriedades em Programas AADL

FLORIANÓPOLIS
2011

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Contribuições para Melhoria do Processo de Verificação Formal de Propriedades em Programas AADL

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a obtenção
do grau de Mestre em Engenharia de Automação e Sistemas.

RAFAEL GARLET DE OLIVEIRA

Florianópolis, Julho, 2011.

Contribuições para Melhoria do Processo de Verificação Formal de Propriedades em Programas AADL

Rafael Garlet de Oliveira

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia de Automação e Sistemas, Área de Concentração em *Controle, Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina.'

Leandro Buss Becker, Dr.
Orientador

Jean-Marie Farines, Dr.
Orientador

José Eduardo Ribeiro Cury, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia de Automação e Sistemas

Banca Examinadora:

Leandro Buss Becker, Dr.

Jean-Marie Farines, Dr.

Leticia Mara Peres, Dra.

Max Hering de Queiroz, Dr.

José Eduardo Ribeiro Cury, Dr.

À Carol.

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer à vida por ter me permitido essa oportunidade de crescimento, onde aprendi tantas coisas e pude olhar mais de perto a essência de mim mesmo.

Gostaria de agradecer aos meus orientadores Leandro Buss Becker e Jean-Marie Farines por acreditarem que eu fosse capaz e me estimularem ao máximo desenvolvimento em todos os sentidos.

Ao colega Gabriel H. R. Santos, pelo apoio no decorrer do trabalho, auxiliando no desenvolvimento das ferramentas e modelagem dos sistemas.

Aos pesquisadores dos laboratórios IRIT e LAAS, por todo suporte com as ferramentas utilizadas.

Aos meus colegas de mestrado agradeço por todo o apoio, amizade e excelente companhia.

Ao pessoal aqui de casa, pelos almoços, café, risadas, reflexões e paciência.

Aos meus pais, por nunca faltarem com o amor e compreensão e aos meus avós, por servirem como exemplo de pessoas com uma vida maravilhosa.

Agradeço ainda a todos que, de forma direta ou indireta, visível ou invisível, ajudaram a seguir os meus passos e abrir os meus olhos.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

Contribuições para Melhoria do Processo de Verificação Formal de Propriedades em Programas AADL

Rafael Garlet de Oliveira

Julho/2011

Orientador: Leandro Buss Becker, Dr. e Jean-Marie Farines, Dr.
Área de Concentração: Controle, Automação e Sistemas
Linha de Pesquisa: Sistemas Computacionais
Palavras-chave: verificação formal, sistemas críticos, propriedades de verificação, AADL
Número de Páginas: 1 + 90

O projeto de sistemas embarcados críticos exige o uso de metodologias adequadas, visto que falhas no sistema podem causar danos catastróficos. Neste contexto se enquadra o projeto Topcased, o qual propõe uma série de ferramentas capazes de suportar a verificação formal de propriedades. A linguagem AADL tem um papel fundamental neste processo, pois sua utilização permite o emprego da transformação de modelos e a aplicação da verificação formal de propriedades. Entretanto, a especificação das propriedades de verificação e a análise dos seus resultados são ainda tópicos em aberto. Esta dissertação visa suprir esta carência propondo um assistente para especificação de propriedades de verificação na linguagem AADL e uma interface para a visualização dos resultados de verificação, juntamente com um simulador de contraexemplos. O assistente construído classifica as propriedades em padrões pré-definidos, utilizando uma linguagem natural ao usuário. Para validar as ferramentas desenvolvidas realizou-se um estudo de caso, o qual consistiu da especificação e verificação de propriedades de um sistema de marcapasso. Desta forma, este trabalho contribui com a melhoria da cadeia de verificação da linguagem AADL no escopo do projeto Topcased.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Automation and Systems Engineering.

Contributions to Improvement of the Formal Properties Verification Process in AADL Programs

Rafael Garlet de Oliveira

July/2011

Advisor: Leandro Buss Becker, Dr. and Jean-Marie Farines, Dr.
Area of Concentration: Control, Automation and Systems
Research Area: Computation Systems
Key words: formal verification, critical systems, verification properties, AADL
Number of Pages: 1 + 90

The design of critical embedded systems requires appropriate methodologies, because a failure can cause catastrophic damage. In this context arises the Topcased project, which proposes a toolset to support model-verification. The AADL language plays a key role in this process, since it allows to perform model transformations and the application of formal properties verification. However, the specification of properties verification and analysis of its results are still not implemented in this verification process. Thus, this work aims to fill this gap by proposing a assistant for specifying properties. The assistant classifies the properties into predefined patterns, using a natural language to the user. These patterns are then processed in logical formulas, which are the inputs to the model-checker. Moreover, this work builds an interface for viewing the verification results and trace simulation, allowing to observe behavior characteristics of the model to fix errors. Finally, the tools were validated with the specification and verification of properties in a pacemaker system. This case study enabled to made comparisons with other proposals that have the same focus. Thus, this work contributes to improving the AADL verification chain of the Topcased project.

Sumário

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivos 2

1.3 Organização do Documento 3

2 Verificação Formal de Propriedades **5**

2.1 Verificação Lógica 6

2.1.1 Lógica Temporal 6

2.1.2 Operadores Lógicos 6

2.1.3 Operadores Temporais 7

2.1.4 Quantificadores de Caminho 8

2.1.5 A Semântica da Lógica Temporal 8

2.1.6 As Lógicas Temporais LTL, CTL e CTL* 8

2.1.7 Especificação de Propriedades em Lógica Temporal 9

2.2 Ferramenta de *model-checking* SELT 10

2.2.1 Exemplo de Funcionamento da Ferramenta 11

2.3 Conclusão 13

3 A Linguagem AADL e a Verificação de Propriedades **15**

3.1 Detalhamento da Linguagem AADL 16

3.1.1 Componente de Sistema 18

3.1.2 Componentes de *Software* 18

3.1.3 Componentes de *Hardware* 20

3.1.4 Conexões 21

3.1.5 Fluxos 23

3.1.6 Modos de Operação 24

3.1.7 Anexo Comportamental 25

3.1.8 Exemplo de Programa AADL: Modelo do *Token Ring* 26

3.2 Ferramentas Para Modelagem em AADL 28

3.3 Verificação de Modelos AADL no Projeto Topcased 30

3.3.1 Linguagem Fiacre 31

3.3.2 Funcionamento da Cadeia de Verificação 32

3.4 A Linguagem BLESS – Uma Alternativa para a Verificação de Modelos . . . 33

3.5 O Projeto COMPASS 34

3.6 Conclusão 36

4 Contribuições para Especificação de Propriedades e Análise de Resultados de Verificação em AADL **37**

4.1 Sistema de Padrões de Propriedades 39

4.1.1 Padrões na Especificação de Propriedades 39

4.1.2 Proposta para Sistema de Padrões de Propriedades 40

4.1.2.1 Propriedade de Ausência 41

4.1.2.2 Propriedade de Existência 42

4.1.2.3 Propriedade de Justiça 44

4.1.2.4 Propriedade de Universalidade 44

4.1.2.5 Propriedade de Precedência 45

4.1.2.6 Propriedade de Resposta 46

4.2 Assistente para Especificação das Propriedades 47

4.2.1 Tradutor de Propriedades 49

4.3 Visualização dos Resultados de Verificação e Contraexemplo 51

4.3.1 Janela das Propriedades Verificadas 52

4.3.2 Janela dos Componentes do Modelo AADL 54

4.3.3 Janela dos Contraexemplos 54

4.3.4 Simulador de Contraexemplos 56

4.4 Conclusão 56

5 Estudo de Caso - Modelagem e Verificação de um Marcapasso Usando AADL 59

5.1 Modelagem em AADL 60

5.2 Verificação Formal do Modelo 63

5.2.1 Análise do Contraexemplo 64

5.3 Comparação com Outras Propostas 66

5.4 Conclusão 69

6 Conclusões 71

A Apêndice – Código AADL do Modelo do Marcapasso 73

A Anexo – Linguagem BLESS 77

A.1 *Assertion* 77

A.2 *SubBLESS* 78

A.3 *BLESS* 79

A.4 Mecanismo de Provas 81

B Anexo – Modelo do Marcapasso em AADL com o anexo de Linguagem BLESS 83

Lista de Siglas e Abreviaturas

AADL	Architecture Analysis and Design Language
ADL	Linguagem de Descrição de Arquitetura
BLESS	Behavior Language for Embedded Systems with Software
CADP	Construction and Analysis of Distributed Processes
COMPASS	Correctness, Modeling, and Performance of Aerospace Systems
CTL	Computation Tree Logic
FDIR	Detecção de Falhas, Isolamento e Reparo
FIACRE	Formato Intermediário para Arquiteturas de Componentes Distribuídos Embarcados
IEEE	Institute of Electrical and Electronics Engineers
LTL	Linear Temporal Logic
MDE	Model Driven Engineering
OSATE	Open Source AADL Tool Environment
SAE	Society of Automotive Engineers
SELT	State/Event LTL model-checker
TINA	TIme petri Net Analyzer
TOPCASED	Toolkit in OPen-source for Critical Application & SystEms Development
TTS	Time Transition Systems
UML	Unified Modeling Language
XML	Extensible Markup Language

Lista de Figuras

1.1 Contribuições nesta dissertação. 3

2.1 Protocolo *token ring* modelado em Redes de Petri e analisado com o SELT. . 12

3.1 Definição da linguagem AADL. 17

3.2 Componente de Sistema. 18

3.3 Componentes de *software*. 19

3.4 Componentes de *hardware*. 20

3.5 Declaração de portas. 21

3.6 Declaração de um grupo de portas. 22

3.7 Declaração de acesso à dados. 22

3.8 Declaração de chamada a subprogramas. 22

3.9 Declaração de fluxos. 24

3.10 Sistema do *token-ring*. 26

3.11 Cadeia de verificação no projeto Topcased. 30

3.12 Comunicação entre os processos Fiacre transformados de um modelo AADL. 32

4.1 Cadeia de verificação do Topcased integrada com as ferramentas desenvolvidas. 38

4.2 Padrões na Especificação de Propriedades. 39

4.3 Assistente para especificação das propriedades de verificação. 48

4.4 Visualização dos resultados de verificação. 53

5.1 Modelagem do sistema do marcapasso em AADL. 61

5.2 Especificação das propriedades de verificação. 64

5.3 Resultados da verificação. 64

5.4 Simulador utilizado para detectar erro de modelagem. 66

Capítulo 1

Introdução

Os sistemas embarcados críticos são sistemas onde uma falha excede muito os seus benefícios, podendo causar danos ao sistema ou aos usuários. Estas possíveis falhas são consideradas catastróficas [18]. São exemplos dispositivos de marcapasso, sistemas aviônicos, sistemas de controle de plantas nucleares.

Alguns destes sistemas, principalmente os de grande porte, apresentam uma elevada complexidade de entendimento e análise. Logo, para estes projetos, são desejáveis metodologias e ferramentas que permitam a representação clara e precisa de seus elementos, como também a garantia de segurança, no sentido de *safety* [22], em seu desenvolvimento.

Neste contexto, surge a linguagem AADL [19] (*Architecture Analysis and Design Language*), que possibilita representar, além da arquitetura, o comportamento de um sistema, permitindo que seja feita uma série de análises, como requisitos de segurança, escalonabilidade e alocação de recursos por exemplo. É uma linguagem de alto nível que visa estabelecer padrões para descrever a arquitetura de *hardware* e *software* de um sistema.

Esta linguagem é utilizada por empresas no setor automobilístico e, principalmente, aeronáutico, assim como por empresas e universidades que participam do projeto Topcased [43] (*Toolkit in OPen-source for Critical Application & SystEms Development*). Topcased consiste em um projeto internacional formado pela união de empresas e universidades pertencentes ao polo francês *Aerospace Valley*, contando também com a participação do Departamento de Automação e Sistemas da UFSC. O objetivo do Topcased é a criação de métodos e ferramentas de código aberto (*open-source*) para o desenvolvimento de sistemas críticos embarcados [6]. Sua proposta é acompanhar todo o ciclo de projeto destes sistemas, promovendo a diminuição de custos e economia de tempo de projeto.

Um dos focos do projeto Topcased é a verificação formal de propriedades, especialmente para os sistemas críticos, que assegura que o sistema está sendo construído corretamente, antes da fase de implementação e testes. A verificação procura varrer todas as possibilidades

de execução do modelo de um sistema, buscando os estados em que uma dada propriedade é falsa. Estas propriedades, por sua vez, representam os requisitos do sistema, como segurança ou exclusão mútua de um recurso, por exemplo, e devem ser satisfeitas para o seu correto funcionamento.

Com este objetivo, o Topcased desenvolveu um conjunto de ferramentas para compor uma cadeia de verificação para algumas linguagens de alto nível [5], dentre as quais pode ser citada a AADL. Aplicando técnicas de *Model Driven Engineering* (MDE) [40], o modelo de um sistema, que é especificado primeiramente em uma destas linguagens de alto nível, como AADL, é transformado em uma representação matemática formal para ser confrontado com as suas propriedades no processo de verificação.

1.1 Motivação

A exemplo do que ocorre com o emprego da verificação formal de uma forma geral, o mapeamento dos requisitos do sistema em propriedades de verificação, usando AADL, ainda é um tópico em aberto [10]. Isto porque a especificação destas propriedades geralmente é feita utilizando-se fórmulas lógicas, ou grafos de comportamento. Em linhas gerais, a especificação de propriedades exige do projetista o conhecimento de conceitos que excedem o domínio de uma linguagem de modelagem como AADL.

Outro ponto que carece maior atenção na cadeia de verificação do projeto Topcased é a análise dos resultados de verificação. A interpretação destes resultados é essencial para constatar quais são as propriedades satisfeitas em um sistema e, para aquelas não satisfeitas, identificar no sistema as suas falhas.

Esta dissertação, que se encontra no escopo do projeto Topcased, desenvolve uma proposta para a especificação de propriedades de verificação na linguagem AADL e também uma alternativa para a visualização dos resultados da verificação e dos contraexemplos.

No desenvolvimento deste trabalho, é construída uma ferramenta, tendo por inspiração alguns trabalhos [17] [15], que classifica as fórmulas lógicas em padrões de propriedades de verificação, facilitando a especificação destas propriedades do ponto de vista do usuário.

1.2 Objetivos

O objetivo principal desta dissertação é contribuir com o processo de verificação de modelos AADL, facilitando a especificação das propriedades desejadas em um sistema, além de proporcionar a apresentação e simulação dos resultados de verificação.

Este objetivo pode ser desmembrado em duas partes. Em primeiro lugar, deve-se desenvolver uma alternativa para a especificação de propriedades em alto nível, permitindo que sejam especificadas propriedades utilizando-se uma linguagem mais natural ao usuário, não exigindo, desta forma, o conhecimento prévio de formalismos lógicos.

Em um segundo momento, deve-se aplicar esta alternativa na verificação de modelos AADL, integrando-o à cadeia de verificação do projeto Topcased, criando ainda ferramentas para apresentar os resultados do processo de verificação, possibilitando sua análise e simulação.

A figura 1.1 ilustra as interfaces construídas, destacando a especificação de propriedades e a visualização e análise dos resultados de verificação, que permitem a simulação de contraexemplos em modelos AADL.

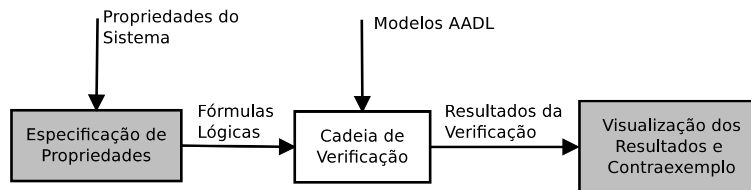

Figura 1.1: Contribuições nesta dissertação.

1.3 Organização do Documento

Esta dissertação está dividida em outros cinco capítulos, estruturados como explicado a seguir. O capítulo 2 trata da especificação e verificação formal de propriedades; o capítulo 3 descreve a linguagem AADL; o capítulo 4 apresenta a proposta desta dissertação para a especificação de propriedades e a análise dos resultados de verificação em modelos AADL, detalhando as ferramentas desenvolvidas; o capítulo 5 detalha o estudo de caso realizado com as ferramentas, além de compará-las a outras propostas similares; e, para finalizar, o capítulo 6 apresenta as conclusões e perspectivas futuras deste trabalho.

O capítulo 2 apresenta, inicialmente, os princípios teóricos que alicerçam o desenvolvimento desta dissertação, detalhando a verificação lógica de propriedades de modelos. É apresentado também, neste capítulo, o ambiente de verificação do projeto Topcased ao qual este trabalho se integra.

O capítulo 3 detalha as construções da linguagem AADL e a forma de especificar um sistema com essa linguagem. Alguns trabalhos relacionados referentes à verificação formal de propriedades de modelos em AADL são também apresentados neste capítulo, bem como a maneira com que AADL se integra ao ambiente de verificação ao qual este trabalho se integra.

No capítulo 4 são detalhadas todas as ferramentas desenvolvidas nesta dissertação, citando um auxiliar para especificação de propriedades de verificação, o sistema de padrões de propriedades utilizado e as ferramentas para análise e simulação dos resultados.

Um estudo de caso é apresentado no capítulo 5, validando e demonstrando a utilização de todos os recursos e facilidades disponibilizados pelas ferramentas desenvolvidas, concernentes tanto à especificação e verificação de propriedades, como também à análise e simulação dos resultados desta. Este capítulo também se dispõe a apresentar uma comparação entre as funcionalidades dispostas nas ferramentas desenvolvidas e nas ferramentas apresentadas como trabalhos relacionados.

Concluindo esta dissertação, o capítulo 6 expõe quais são os resultados e contribuições alcançados por este trabalho, apresentando as possibilidades para trabalhos futuros que podem enriquecer ainda mais a modelagem e verificação de sistemas em AADL.

Capítulo 2

Verificação Formal de Propriedades

Neste capítulo é tratada a verificação formal de propriedades e as diferentes técnicas existentes. Primeiramente é necessário fazer a distinção entre a verificação e a validação de sistemas. Segundo Roussel [34], a validação determina se o modelo concorda com o propósito do projetista e a verificação determina se a semântica de um sistema é correta, ou seja, se ele atende a certas propriedades. De uma maneira mais simples, de acordo com Boehm [7], pode-se afirmar que a validação assegura que o sistema correto está sendo construído, enquanto que a verificação assegura que o sistema está sendo construído corretamente. Como técnicas para validação podem ser citadas a simulação e testes, de acordo com Clarke [13], que permitem que se observe se o sistema atende a determinada funcionalidade. Entretanto, em modelos com elevado número de elementos, se torna impraticável atingir todos os estados de execução em uma simulação ou teste. Assim surge a verificação de modelos, que procura varrer todas as possibilidades de execução do modelo de um sistema.

A verificação de modelos, de agora em diante referenciada apenas como verificação, é um conjunto de técnicas que permitem confrontar e comparar a descrição operacional de um sistema com suas especificações (ou propriedades esperadas). Fazem parte da verificação a descrição comportamental do sistema, a descrição das propriedades esperadas e o procedimento de decisão, que verifica se o sistema satisfaz as propriedades esperadas. São duas as categorias de verificação: a verificação comportamental e a verificação lógica.

O procedimento da verificação comportamental [29] faz uso de dois elementos básicos, cada um na forma de um grafo representando um sistema de transição: a descrição operacional do sistema e a descrição das propriedades esperadas. O procedimento de decisão se baseia em conceitos de relações de equivalência e consiste em buscar a equivalência entre os dois grafos de comportamento e, em caso positivo, a propriedade esperada é considerada como verdadeira.

No caso da verificação lógica as propriedades esperadas são representadas na forma de

fórmulas lógicas utilizando a lógica temporal e a descrição operacional do sistema é representada na forma de um grafo. O procedimento de decisão utilizado pela verificação lógica, conhecido como *model checking* [13], consiste em uma busca a todos os estados de execução do modelo, para determinar se uma dada especificação é verdadeira ou não.

Existem várias ferramentas que desempenham a verificação formal de modelos. Um exemplo conhecido de ferramenta que emprega a verificação comportamental é o CADP [21]. Algumas ferramentas conhecidas que utilizam a verificação lógica são Uppaal [2], NuSMV [12], MRMC [25] e SELT [3], a utilizada nesta dissertação. Na próxima seção, a verificação lógica será detalhada, por ser o foco desta dissertação.

2.1 Verificação Lógica

O processo de decisão utilizado na verificação lógica é o *model checking*. Segundo Clarke [13], este procedimento consiste em varrer todo o espaço de estados de um modelo para buscar se uma dada propriedade é verdadeira ou não. No caso de ser encontrado um estado que torne a propriedade falsa, é gerado um contraexemplo. Este procedimento pode ser realizado automaticamente por meio de um algoritmo. As fórmulas lógicas referentes às propriedades de verificação utilizadas são especificadas de acordo com a lógica temporal.

2.1.1 Lógica Temporal

A motivação para desenvolver a lógica temporal é que a lógica de primeira ordem, utilizada pela lógica de predicados [41], não permite que seja utilizada a noção de precedência de estados no tempo. A lógica temporal foi desenvolvida especificamente para representações e raciocínios que envolvem a noção de ordem no tempo. Na verificação, a lógica temporal serve para especificar formalmente propriedades relacionadas com a execução de um sistema. A lógica temporal faz uso de proposições atômicas para declarar uma afirmação sobre um estado. Uma proposição pode ter o seu valor verdadeiro, ou falso em determinado estado. Na lógica temporal existem dois tipos de operadores: lógicos e temporais. Além destes operadores existem os quantificadores de caminho que permitem quantificar sobre o conjunto de execuções.

2.1.2 Operadores Lógicos

A lógica temporal se utiliza da lógica proposicional, que é a composição de proposições e operadores lógicos. Os operadores lógicos são:

- Constantes: *true* e *false*

- Negação: $\neg$

  A fórmula $\neg P$ é verdadeira quando a proposição P for falsa.

- Conjunção: $\bigwedge$

  A fórmula $P \bigwedge Q$ é verdadeira quando as proposições P **e** Q forem verdadeiras.

- Disjunção: $\bigvee$

  A fórmula $P \bigvee Q$ é verdadeira quando as proposições P **ou** Q forem verdadeiras.

- Implicação: $\Rightarrow$

  A fórmula $P \Rightarrow Q$ é verdadeira quando se a proposição P for verdadeira a proposição Q deve ser verdadeira.

- Dupla Implicação: $\iff$

  A fórmula $P \iff Q$ é verdadeira quando as proposições P e Q forem ambas verdadeiras ou ambas falsas.

2.1.3 Operadores Temporais

Permitem construir expressões relacionadas ao sequenciamento dos estados ao longo de uma execução e não apenas aos estados individualmente. Os operadores temporais são:

- **X** (*Next*), também denotado como: $\bigcirc$

  A fórmula **X P** é verdadeira quando a proposição P é verdadeira no próximo estado de execução.

- **F** (*Future*), também denotado como: $\Diamond$

  A fórmula **F P** é verdadeira quando a proposição P é verdadeira em algum estado no futuro.

- **G** (*Globally*), também denotado como: $\Box$

  A fórmula **G P** é verdadeira quando a proposição P é verdadeira em todos os estados de execução.

- **U** (*Until*)

  A fórmula **P U Q** é verdadeira quando a proposição P é verdadeira até o momento em que a proposição Q se torne verdadeira.

- **W** (*Weak Until*)

  A fórmula **P W Q** é verdadeira quando a proposição P é verdadeira enquanto a proposição Q não for verdadeira.

- **R** (*Release*)

  A fórmula **P R Q** é verdadeira se sempre que a proposição Q for falsa a proposição P é verdadeira no estado imediatamente anterior.

2.1.4 Quantificadores de Caminho

Os quantificadores de caminho são:

- **A** (*All*)

  Percorre todos os caminhos de execução do modelo.

- **E** (*Exist*)

  Procura por, ao menos, um caminho de execução do modelo.

2.1.5 A Semântica da Lógica Temporal

A Lógica Temporal se utiliza de modelos chamados Estruturas de Kripke. Estes modelos podem ser entendidos como autômatos que contêm todos os estados alcançáveis do sistema. Um autômato pode ser visto como:

$$A = \{Q, T, q_0, l\} \quad \texttt{com} \quad T \subseteq Q \times Q \tag{2.1}$$

Onde Q são os estados do sistema, T são as transições entre os estados, q_0 é o conjunto de estados iniciais e l é a função *label*, que associa a cada estado q $\in$ Q o conjunto l(q) de proposições atômicas que são válidas em q. As propriedades do sistema são representadas por meio de fórmulas lógicas Φ. Para uma dada fórmula Φ ser verdadeira tem-se a seguinte equação:

$$A, \sigma, i \models \Phi \tag{2.2}$$

Esta equação expressa que em um dado tempo i da execução σ no autômato A, a fórmula Φ é verdadeira.

2.1.6 As Lógicas Temporais LTL, CTL e CTL*

A *Linear Temporal Logic* (LTL) [33] e a *Computation Tree Logic* (CTL) [14] são as duas lógicas temporais mais usadas em *model checking*. Podem ser vistas como um fragmento da

lógica temporal CTL*. A lógica CTL* permite combinar proposições, operadores lógicos, operadores temporais e quantificadores de caminho. A lógica LTL é obtida de CTL* ao desconsiderar os quantificadores de caminho **A** e **E**. LTL lida apenas com o conjunto de execuções e não com a maneira como elas estão organizadas em uma árvore. No caso da lógica CTL, os quantificadores de caminho são permitidos, mas os operadores temporais (**X**, **F**, **G**) devem estar imediatamente dentro do escopo destes quantificadores. A expressividade de CTL é limitada pela necessidade de sempre quantificar sobre as possibilidades futuras.

2.1.7 Especificação de Propriedades em Lógica Temporal

De acordo com a literatura [9][26], podem ser destacados alguns tipos de propriedades como os mais utilizados na verificação de sistemas.

- Propriedades de Alcançabilidade

  Expressa que alguma situação particular pode ser atingida, como por exemplo, atingir um certo estado a partir de algum ponto, ou então atingir novamente o estado inicial. A lógica CTL permite expressar a alcançabilidade com a fórmula **EFP**, expressando que existe uma execução tal que no futuro a proposição **P** será verdadeira. A lógica LTL permite apenas expressar a não alcançabilidade.

CTL: EFP (Existe um caminho em que no futuro a proposição P é verdadeira.) LTL: $G\neg P$ (A proposição P nunca é verdadeira.)

- Propriedades de Segurança

  Expressa que, sob certas condições, algo nunca ocorrerá.

CTL: $AG\neg unsafe$ (A proposição *unsafe* nunca poderá ser verdadeira.) LTL: $G\neg(P1 \bigwedge P2)$ (As proposições P1 e P2 nunca são verdadeiras simultaneamente.)

- Propriedades de Vivacidade

  Expressa que, sob certas condições, algo deverá ocorrer.

CTL: $AG(request \Rightarrow Freceive)$ (Uma solicitação é sempre atendida.) LTL: $G(I \Rightarrow FP)$ (Uma vez que a proposição I é verdadeira sempre implica que a proposição P se torne verdadeira em algum momento.)

- Propriedade Livre de Bloqueio

  Expressa que o sistema nunca pode chegar em uma situação em que nenhum progresso é possível.

CTL: $AGEXtrue$ (Para todo estado alcançável, existirá um estado imediatamente sucessor.)
LTL: $G(Xfalse \Rightarrow terminal)$ (Quando não existir um próximo estado, significa que foi atingido um estado terminal do modelo.)

- Propriedade de Justiça

  Expressa que, sob certas condições, algo ocorrerá, ou não ocorrerá, infinitamente muitas vezes.

CTL: (Para poder representar com essa lógica se faz necessário impor restrições ao modelo – *fairness constraints* [22].)
LTL: GFP (A proposição P pode ser atingida infinitas vezes.)

2.2 Ferramenta de *model-checking* SELT

SELT (*State/Event LTL model-checker*) é uma poderosa ferramenta para verificação formal de propriedades que faz parte do *toolbox* TINA [1] [3] e é empregada no projeto Topcased [43]. Este conjunto de ferramentas foi desenvolvido pelo laboratório francês LAAS com o propósito de criação, edição e análise de Redes de Petri, Redes de Petri Temporais e Sistemas de Transição Temporizados (TTS), sendo que seu processo de *model-checking* é baseado na verificação lógica.

As entradas para o *model-checker* SELT são as estruturas de Kripke, que representam o sistema com um formalismo matemático, e as fórmulas lógicas no formato LTL, representando as propriedades que se desejam verificar. As estruturas de Kripke são o resultado da compilação do modelo do sistema representado como um sistema TTS.

Estruturas de Kripke normalmente são conhecidas como sistemas de transição não rotulados; são grafos direcionados que representam todos os estados alcançáveis do modelo. De acordo com as definições em [11], os sistemas TTS podem ser considerados como Redes de Petri Temporais estendidas com dados, ou seja, sistemas de transição de estados com acesso a variáveis.

As fórmulas aceitas pelo SELT são escritas com uma linguagem LTL estendida com algumas constantes. Suas primitivas são constituídas pelos seguintes elementos:

- Constantes:

  T (*true*), **F** (*false*)

  dead (propriedade de *deadlock*)

  div (propriedade de divergência temporal)

[1] www.laas.fr/tina

- Prefixos:

  G – *always* (representado como [])

  F – *eventually* (representado como <>)

  X – *next* (representado como ())

  $\neg$ *negation* (representado como **-**)

- Infixos:

  U – *until*

  R – *release* (representado como V)

  => (implica)

  <=> (equivalente)

  $\bigwedge$ – conjunção (representado como /\)

  $\bigvee$ – disjunção (representado como \/)

  <=, >=, = (menor ou igual, maior ou igual, igual)

  + (soma)

  * (multiplicação)

Alguns exemplos de fórmulas lógicas aceitas pelo SELT juntamente com sua semântica:

- $[](A => <> B)$

  Vivacidade: uma certa propriedade A de um dado modelo sempre leva a propriedade B deste modelo.

- $[] - dead$

  Ausência de *deadlock*.

Os resultados do processo de verificação que são gerados pelo SELT podem ser guardados em um arquivo de texto. Neste arquivo consta o resultado de cada propriedade e, no caso de alguma fórmula ser verificada como falsa, consta também o contraexemplo que a torna falsa. Este contraexemplo, por sua vez, pode ser carregado e analisado no simulador do TINA para que se observe um o caminho de execução que torna a propriedade falsa.

2.2.1 Exemplo de Funcionamento da Ferramenta

Exemplificando o funcionamento da ferramenta, na figura 2.1 apresenta-se o modelo do protocolo *token-ring*, um exemplo clássico da literatura [42] onde é feito o controle de

uma seção crítica por meio de um *token*. Neste exemplo, o sistema é modelado com três estações; cada estação somente pode acessar a seção crítica depois de fazer a requisição pelo *token* e depois que este for devidamente recebido. O modelo é feito em Redes de Petri com a ferramenta TINA e verificado com o SELT. Neste exemplo são utilizadas três fórmulas lógicas como propriedades de verificação:

- Exclusão mútua: $[] - ((cs_1 /\backslash cs_2) \backslash/ (cs_2 /\backslash cs_3) \backslash/ (cs_1 /\backslash cs_3));$

  VERDADEIRO (É sempre falso que a seção crítica seja acessada por duas ou mais estações simultaneamente.)

- Garantia no atendimento das requisições: $[]((wait_1 => <> cs_1) \backslash/ (wait_2 => <> cs_2) \backslash/ (wait_3 => <> cs_3));$

  VERDADEIRO (Toda estação que faz o pedido do *token* entra em seção crítica no futuro.)

- Estação 3 sempre entra em seção crítica: $[](token_3 => <> cs_3);$

  FALSO (Qualquer estação somente pode entrar em seção crítica se fizer o pedido.)

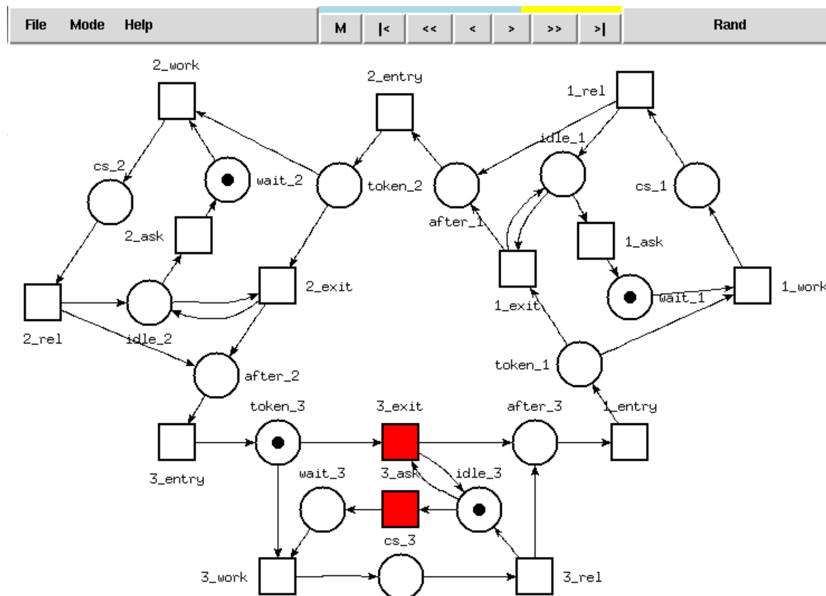

Figura 2.1: Protocolo *token ring* modelado em Redes de Petri e analisado com o SELT.

O resultado da terceira propriedade é gerado juntamente com o seu contraexemplo que é carregado no simulador apresentando qual é estado de execução que torna a propriedade falsa, como mostra a figura 2.1. Por meio deste contraexemplo pode-se observar uma situação em que a estação 3 se encontra no estado *idle* (significa que não fez o pedido) e acaba por não receber o *token* apesar deste já se encontrar disponível, não entrando, assim, em seção crítica.

2.3 Conclusão

Este capítulo apresentou algumas técnicas de verificação, com uma ênfase maior na abordagem lógica que é o foco deste trabalho. Nesta abordagem é empregada a lógica temporal, cuja definição é propriamente apresentada, como também alguns exemplos de definição de propriedades.

Por fim, é apresentada a ferramenta de verificação SELT desenvolvida no projeto Topcased e que será empregada neste trabalho.

Capítulo 3

A Linguagem AADL e a Verificação de Propriedades

Este capítulo apresenta a linguagem AADL, empregada no projeto Topcased para a modelagem de sistemas embarcados críticos, e as ferramentas utilizadas no Topcased para verificação de modelos AADL, além de apresentar a linguagem BLESS, uma alternativa para assegurar o correto funcionamento de modelos AADL, e o projeto COMPASS para modelagem, validação e verificação de modelos AADL.

Segundo Luckham [28], a arquitetura de um sistema pode ser entendida como sua especificação, onde se identificam seus componentes e suas interações, por meio de interfaces, conexões e restrições. Há aproximadamente duas décadas começaram-se a pensar formas para representação formal de arquitetura de sistemas por meio de linguagens computacionais. Nesse contexto inicia-se o desenvolvimento das linguagens de descrição de arquitetura (ADL). São linguagens de alto nível que visam estabelecer padrões para descrever a arquitetura de hardware e software de um sistema.

Por meio das ADLs é possível reduzir tempo e custo de projeto, pois modelar a arquitetura de um sistema com uma linguagem de alto nível significa representar linhas de código por componentes gráficos. Este tipo de linguagem é focado na estrutura do sistema, sem se importar propriamente com detalhes de implementação do código.

Conforme Medvidovic [30], uma ADL deve ser capaz de modelar os componentes do sistema, suas conexões e configurações:

- Componentes

  Podem ser entendidos como unidades do sistema (de hardware ou de software). Cada componente possui a sua própria interface para comunicação com os outros componentes do sistema. Os componentes podem conter outros componentes, formando

assim a hierarquia do sistema. Conforme a ADL escolhida, é possível que se descreva o comportamento interno dos componentes.

- Conectores

  São elementos característicos das ADLs, que providenciam as ligações entre as interfaces dos componentes.

- Configurações

  Consistem em interfaces e conexões entre os componentes de um sistema, representando a estrutura da arquitetura.

Outra vantagem de se descrever um sistema utilizando uma ADL é que estas linguagens permitem a análise de algumas características do sistema, como sua escalonabilidade por exemplo, ainda em fase de projeto. Além disso, o desenvolvimento baseado em arquitetura permite que seja empregada a Engenharia Dirigida a Modelos (MDE) [40], que desempenha a transformação de modelos de alto nível em modelos com outros níveis de abstração, permitindo aplicar técnicas de validação e verificação formal de propriedades, além de geração do código para posterior implementação.

AADL [19] [35] (*Architecture Analysis and Design Language*) é uma linguagem de descrição de arquitetura que tem como foco a descrição formal de sistemas embarcados e de tempo real. Sua criação é inspirada em uma linguagem de descrição de arquitetura chamada MetaH, que foi desenvolvida pela *Honeywell*, de onde foram derivados alguns conceitos como os componentes de *software* e de *hardware* e que tem sido aplicada em sistemas da aviação, robótica, automobilísticos. Inicialmente foi denominada *Avionics Architecture Description Language* e aprovada por companhias aeroespaciais. Posteriormente, após o interesse de outros grupos de pesquisa e empresas europeias, seu nome foi alterado para *Architecture Analysis and Design Language*, pois a linguagem permite o desenvolvimento de sistemas em geral, e não somente os aeroespaciais. No ano de 2004 o comitê formado pela SAE (*Society of Automotive Engineers*) publicou o documento de definição da linguagem, que foi aprovado por 23 organizações dos EUA e Europa.

3.1 Detalhamento da Linguagem AADL

A linguagem de modelagem AADL tem como objetivo descrever a arquitetura de *software* e a arquitetura da plataforma de execução (*hardware*) de sistemas críticos de tempo real. Um modelo AADL pode ser destinado à análise de suas propriedades e a posterior geração de código para a devida implementação em *hardware*.

A definição da linguagem contém um formato padronizado XML, baseado no seu metamodelo, que objetiva facilitar a transformação de modelos e a integração das ferramentas de

suporte, contém também a linguagem gráfica e a textual, como apresentado na figura 3.1. Dessa forma, AADL tem se tornado uma linguagem importante para estudo e aplicação devido às facilidades proporcionadas pela transformação de modelos que se utiliza da Engenharia Dirigida a Modelos (MDE).

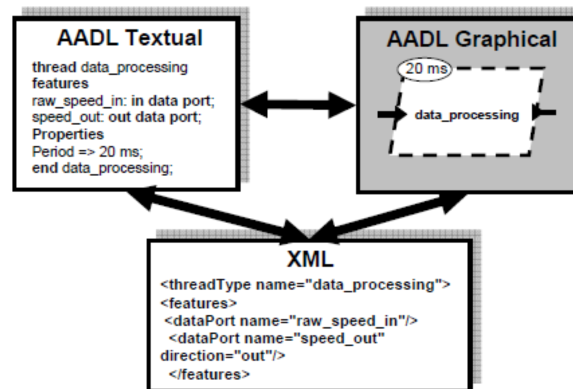

Figura 3.1: Definição da linguagem AADL.

Um modelo AADL pode ser entendido como um conjunto de componentes em uma organização hierárquica. Os componentes AADL são caracterizados por sua identidade (nome), interface com outros componentes, propriedades, subcomponentes e interações entre estes. Os componentes podem ainda conter a declaração de fluxos de informações, modos de operação e podem ser estendidos por meio dos anexos de linguagem. Um componente AADL é descrito em duas partes: tipo e implementação. A primeira descreve a interface deste elemento e atributos (como portas de comunicação) e a segunda, a sua estrutura interna (os seus subcomponentes e conexões entre eles), chamadas de subprogramas, modos de operação, implementação de fluxos e propriedades.

Os componentes AADL são divididos em três categorias: os componentes de *software* (*process*, *thread*, *thread group*, *data* e *subprogram*); os componentes de *hardware* (*processor*, *memory*, *device* e *bus*); e os componentes que dão a noção de sistema (*system* e *package*).

A interação entre os componentes é feita por meio de elementos de interface, que são declarados como *features* de um componente. Estes elementos, pontos de comunicação na interface de um componente, são os seguintes: portas, acesso a componentes (acesso a dados, por exemplo) e chamadas de subprogramas. As portas são responsáveis pela transmissão de dados e eventos, enquanto que os grupos de portas servem para organizar as portas em agrupamentos. A declaração de portas e acesso a componentes deve ser direcional, explicitando se é de saída ou de entrada no caso das portas e, no caso de acesso a componentes, se é fornecido ou requerido. A transmissão de dados através destes canais de comunicação e através dos componentes pode ser explicitamente declarado por meio de fluxos de informação.

O conceito de herança de componentes pode ser também aplicado, de forma semelhante

às linguagens de programação com orientação a objetos, onde um componente pode herdar a especificação de um outro componente, juntamente com suas propriedades e também estender novas especificações e propriedades.

A linguagem AADL se utiliza do conceito de modos de operação. Por meio dos modos, é possível determinar quais são os subcomponentes ou elementos de *software* de um determinado componente que estarão habilitados em um determinado instante de execução. Os modos podem ser entendidos como uma abstração da configuração dos componentes e suas conexões.

Algumas propriedades podem ser atribuídas aos componentes, para acrescentar restrições sobre as suas características e conexões. Estas propriedades dizem respeito a modo de acesso a dados, características de conexão, período de execução, *deadline*. Podem estar relacionadas aos modos de operação, sendo que em cada modo uma propriedade pode assumir um valor diferente.

A declaração dos elementos AADL define a arquitetura de um sistema, enquanto que o seu comportamento deve ser definido por meio de linguagens anexas aos componentes, como o Anexo Comportamental por exemplo. As próximas seções detalham os elementos da linguagem AADL.

3.1.1 Componente de Sistema

Representa um componente que possibilita a integração entre os demais componentes. Nesta categoria se enquadra os componente sistema (*system*). Pode conter a declaração de outros sistemas, assim como a declaração de componentes de *software* e de *hardware*. Sua representação gráfica pode ser vista na figura 3.2.

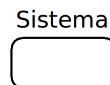

Figura 3.2: Componente de Sistema.

3.1.2 Componentes de *Software*

Os componentes de software podem ser divididos em cinco categorias: processo, *thread*, *thread group*, *data* e subprograma. Sua representação gráfica pode ser vista na figura 3.3. Nas próximas subseções segue a descrição mais detalhada de cada um deles.

Processo (*process*) Processos podem ser entendidos como um espaço de memória virtual, onde estão contidas estruturas de dados referenciadas por componentes externos e códigos

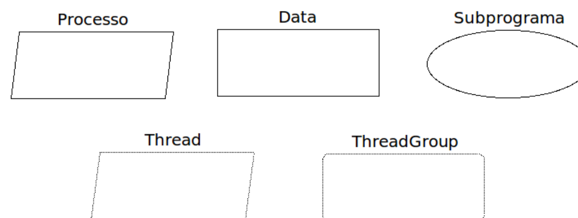

Figura 3.3: Componentes de *software*.

executáveis. Um processo não contém explicitamente uma *thread*, mas esta deve ser declarada para torná-lo um componente ativo.

Na declaração de um processo são permitidos três tipos de subcomponentes: *thread*, *thread group* e *data*. Para os processos e seus subcomponentes, é permitida a declaração de algumas propriedades, como proteção de memória, informação sobre código executável e protocolo de escalonabilidade.

Thread Uma *thread* é uma unidade escalonável de execução sequencial de código fonte. É declarada juntamente com suas propriedades de execução, explicitando, por exemplo, se é periódica, aperiódica, seu tempo de execução, ocupação de memória, entre outros.

O único subcomponente permitido é do tipo *data*, e são permitidas chamadas a subprogramas. Um componente do tipo *process* pode conter uma ou mais *threads*, e estas podem ser organizadas em componentes do tipo *thread group*.

Thread Group Um *thread group* é uma abstração para organização de *thread*, *thread group* e *data* dentro de um processo. Não representam um espaço de memória virtual ou uma unidade de execução, mas definem uma referência para múltiplas *threads* e dados associados. As propriedades relacionadas a este componente dizem respeito a características de tempo, memória e processador. Não possui um papel ativo de execução como uma *thread* e não permite a chamada de subprogramas, mas permite a declaração de fluxo de dados através de *flows* e o controle dos subcomponentes através dos modos de operação.

Data Um componente deste tipo serve para declarar os tipos de dados que serão utilizados no modelo de um sistema. A única *feature* permitida para este elemento são os subprogramas, que neste caso podem ser entendidos como um método de acesso aos dados. Seus subcomponentes são somente *data*. Como propriedades pode ser declarado o tamanho dos dados ou o protocolo de acesso.

Subprograma (*subprogram*) Este componente representa trechos de código fonte de execução sequencial, que podem ser invocados com ou sem parâmetros. As chamadas de subprogramas podem ser feitas de uma *thread* ou de um outro subprograma. Um subprograma pode representar uma chamada de método para operação com dados, ou chamadas de procedimentos remotos. Este componente não permite a declaração de subcomponentes. A declaração de propriedades está relacionada com requisitos e mapeamento de memória e com chamadas de subprogramas. A interação entre estes componentes é realizada por meio de portas de eventos.

3.1.3 Componentes de *Hardware*

A linguagem AADL fornece quatro categorias de componentes de plataforma de execução: processador (*processor*), memória (*memory*), dispositivo externo (*device*) e *bus*. Em uma especificação AADL, os componentes de *software* devem ser mapeados nos componentes de *hardware*, definindo onde o código é executado e onde é armazenado. Sua representação gráfica pode ser vista na figura 3.4. Nas próximas subseções segue a descrição mais detalhada de cada uma destas categorias.

Figura 3.4: Componentes de *hardware*.

Processador (*Processor*) O processador é responsável pelo escalonamento e execução de *threads*. Seu único subcomponente permitido é o *memory*. Suas propriedades relacionadas dizem respeito a descrição de *hardware* e protocolos de escalonamento e de disparo.

Memória (*Memory*) Representam componentes de armazenamento para dados e código executável. Este componente pode representar a memória interna de um processador ou uma unidade separada de memória que se comunica através de barramento (*bus*). Seu único subcomponente permitido é o próprio *memory*, sendo possível a modelagem de bancos de memória. As propriedades estão relacionadas a protocolo de acesso, tamanho de palavra e tipo de acesso (leitura-escrita).

Dispositivo Externo (*Device*) Os dispositivos responsáveis pela comunicação com o ambiente externo são representados por este componente. Pode ser visto de três formas diferentes: um componente físico, uma parte do sistema de aplicação ou uma unidade controlável do ambiente. Suas propriedades relacionadas dizem respeito a restrições de *software*, como

linguagem do código fonte e tamanho de código, e a restrições da plataforma de execução, como descrição do *hardware*.

Bus Este componente representa o barramento de comunicação entre memória, processador e dispositivos externos e os protocolos de comunicação associados. Os barramentos podem ser conectados também a outros barramentos, permitindo a flexibilidade na comunicação. Sua única *feature* permitida é o requerimento de acesso a barramento. As propriedades relacionadas são referentes a características de transmissão, como protocolos de acesso, tamanho de mensagem, tempo de transmissão, atraso de propagação, entre outras. Não é permitida a declaração de subcomponentes e o único componente que pode conter sua declaração é o *system*.

3.1.4 Conexões

A interação entre os componentes AADL é feita por meio de pontos de comunicação que são os seguintes: portas, grupos de portas, acesso a subcomponentes, chamadas de subprogramas e conexões de parâmetros. Os elementos de interface são declarados na seção de *features* de um componente.

As portas (*ports*) representam a interface de comunicação para transmissão direcional de dados, eventos ou ambos, sendo, então, classificadas como portas de dados (*data port*), portas de eventos (*event port*), ou ainda porta de dados e eventos (*event data port*). As portas podem ser agrupadas em grupos de portas. O tempo da comunicação e tempo de amostragem podem ser também modelados. A forma gráfica da declaração dos diferentes tipos de porta pode ser vista na figura 3.5. Nesta figura estão representadas somente portas de entrada, entretanto as portas podem ser declaradas como de saída, ou em ambas as direções.

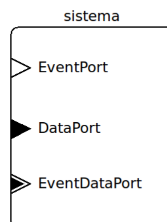

Figura 3.5: Declaração de portas.

Os grupos de portas (*port groups*) representam um conjunto de portas ou de outros grupos de portas. O conteúdo e a estrutura de um grupo de portas é completamente declarado no componente *type*, não existindo a declaração de implementação para grupos de portas. São utilizados para reduzir o número de conexões, simplificar a representação gráfica, agrupar

portas com as mesmas propriedades, entre outras funções. Sua representação gráfica pode ser vista na figura 3.6.

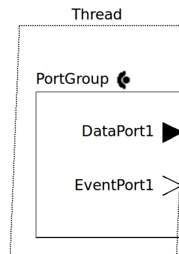

Figura 3.6: Declaração de um grupo de portas.

O acesso a subcomponentes de dados (*data*) e barramento (*bus*) é possível graças ao elemento de interação *data access*. Sua declaração deve ser direcional, explicitando se o acesso é fornecido ou requerido. As suas propriedades estão relacionadas a controle de concorrência e modo de leitura. Sua representação gráfica pode ser vista na figura 3.7.

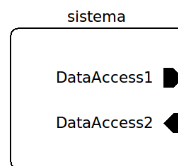

Figura 3.7: Declaração de acesso à dados.

As chamadas a subprogramas (*subprogram calls*) ocorrem em declarações de *threads* ou de implementações de subprogramas. As chamadas remotas a subprogramas contidos em outros processos também é permitida. Suas propriedades relacionadas se referem a condições para chamadas remotas e ao mapeamento de memória. Sua representação gráfica pode ser vista na figura 3.8.

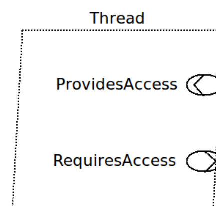

Figura 3.8: Declaração de chamada a subprogramas.

A última forma de comunicação entre componentes AADL é feita por meio de parâmetros. São relações de transferência de dados associadas a chamadas de subprogramas. Representa o envio e retorno de valores através de subprogramas. Estas conexões podem ser úteis para análise do fluxo de informações, quando utilizada a declaração de fluxos. Sua representação gráfica é a mesma de chamadas a subprogramas.

3.1.5 Fluxos

O fluxo de informações através dos componentes e conexões pode ser declarado explicitamente por meio do elemento *flow*. Este elemento representa uma abstração do caminho que será percorrido pelos dados no modelo. Sua completa declaração é feita em duas partes, uma contida na declaração do tipo de componente e outra na implementação deste. A parte contida no tipo do componente declara os elementos de um fluxo, ou seja, sua origem (*source*), destino (*sink*) e caminho (*path*). A parte contida na implementação descreve todo o caminho de dados nos subcomponentes contidos em um componente. Este elemento permite que sejam feitas algumas análises no modelo, como a propagação de erros e qualidade de serviço.

```
SYSTEM ControleTemperatura
END ControleTemperatura;

SYSTEM IMPLEMENTATION ControleTemperatura.others
SUBCOMPONENTS
   Controlador : SYSTEM Controlador;
   Sensor : DEVICE Sensor;
   Atuador : DEVICE Atuador;
END ControleTemperatura.others;

SYSTEM Controlador
FEATURES
   Temperatura : IN DATA PORT Temperatura;
   SinalDeControle : OUT EVENT DATA PORT SinalDeControle;
 flows
          fluxoControle : flow path Temperatura -> SinalDeControle;
END Controlador;

DEVICE Sensor
FEATURES
   Temperatura : OUT DATA PORT Temperatura;
 flows
          fluxoTemperatura : flow source Temperatura;
END Sensor;

DEVICE Atuador
FEATURES
   SinalDeControle : IN EVENT DATA PORT SinalDeControle;
 flows
          fluxoSinal : flow sink SinalDeControle;
END Atuador;
```

Listagem 3.1: Declaração de fluxos de informação

Na figura 3.9 é mostrado um exemplo de declaração de um sistema de controle simplificado. Existe a declaração de dois *devices*, um sensor que envia os dados para um sistema controlador e um atuador que recebe os dados de atuação deste sistema. A declaração do fluxo de informações pode ser observada na listagem 3.1, que é a forma textual desta representação gráfica. Após a geração do modelo textual, são feitas as declarações dos fluxos. Aqui não são apresentadas as declarações nos componentes de implementação.

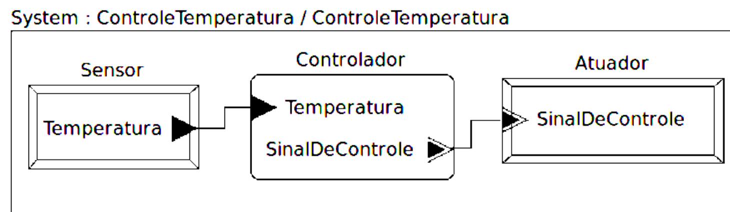

Figura 3.9: Declaração de fluxos.

3.1.6 Modos de Operação

Os modos de operação podem ser entendidos como configurações de componentes, conexões e propriedades. Representam estados diferentes de operação de cada componente. Ou seja, quando um determinado componente se encontra em um certo modo de operação, somente algumas das suas configurações (como portas e subcomponentes, por exemplo) estão ativas e quando este componente passa para outro modo de operação, são outras as configurações que serão ativadas. Um possível exemplo simples de aplicação, para melhor entendimento, seria deixar de habilitar um certo botão do sistema em um modo de operação, para que não ocorra uma dada interrupção durante uma parte crítica da execução.

```
system implementation aircraft.others
   subcomponents
      takingOff : process takingOff.others in modes (takeOff, navigate);
      navigating : process navigating.others in modes (navigate, land);
      landing : process landing.others in modes (land);
   connections
      event port takingOff.o -> navigating.i in modes (takeOff);
      event port navigating.o -> landing.i in modes (navigate, land);
   modes
      takeOff : initial mode;
      navigate : mode;
      land : mode;
      takeOff -[takingOff.o]-> navigate;
      navigate -[navigating.o]-> land;
end aircraft.others;
```

Listagem 3.2: Declaração de modos de operação

Os modos podem ser usados para estabelecer três diferentes tipos de configuração: configurações de componentes e conexões ativas, sequencias de chamada variáveis e por fim propriedades específicas de componentes de *hardware* e de *software*.

No exemplo da listagem 3.2 pode-se observar a declaração de uma instância do sistema *aircraft* com seus modos de operação: *takeOff*, *navigate* e *land*. Neste exemplo, os modos de operação definem diferentes configurações do sistema. Quando *aircraft* se encontra no modo *takeOff* o processo *landing* não está ativo, e quando se encontra no modo *land* o processo *takingOff* não está ativo. Os disparos das transições de modos se dão por meio do final de operação dos processos *takingOff* e *navigating*.

3.1.7 Anexo Comportamental

Os componentes apresentados nas seções anteriores servem apenas para descrever a arquitetura do sistema, no entanto não conseguem definir propriamente seu comportamento. Para suprir esta necessidade, foi desenvolvido o anexo comportamental que possibilita estender alguns componentes e descrever o seu comportamento [37]. Consiste de um anexo para a linguagem AADL que pode ser declarado na implementação de componentes. Este anexo pode ser entendido como um sistema de transição de estados, semelhante aos conhecidos autômatos, declarando estados do componente e transições com guardas e ações.

O anexo comportamental interage com os elementos do componente em que está inserido, relacionando-se com suas portas, dados e modos. As transições e ações podem acessar as variáveis do componente, com a possibilidade de efetuar a atribuição e teste de variáveis. Dessa forma é possível que haja a comunicação entre componentes em um sistema, por meio da troca de mensagens e de dados. Este anexo possibilita que haja a chamada de subprogramas no comportamento de um componente, assim como possibilita que se defina o comportamento de subprogramas.

Os modos de operação de um modelo AADL são considerados como estados dentro do anexo. Assim o anexo estende o sistema de modos associando guardas e ações às transições dos modos. Além disso, existe a possibilidade de definir um autômato relativo a um modo específico.

Características de tempo real podem ser definidas através de três primitivas: *delay* promove a suspensão da execução do componente durante um dado período de tempo, *computation* especifica um dado tempo de computação e por fim *timeout*, que pode ser utilizado com guardas.

Um exemplo de declaração do anexo comportamental pode ser visto na próxima seção, onde é declarado um modelo completo em AADL do protocolo *token-ring*.

3.1.8 Exemplo de Programa AADL: Modelo do *Token Ring*

A fim de exemplificar os componentes da linguagem AADL, é utilizado o sistema clássico do protocolo *token-ring* [42][23]. Foi escolhido um sistema com três estações que acessam uma seção critica, sendo que este acesso somente é permitido pela estação que possuir o *token*. A rede, neste caso, é tida como um componente à parte que promove a circulação do *token* entre as estações e, conforme a requisição, entrega o *token* para uma delas.

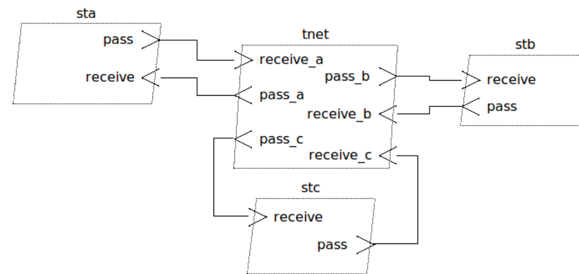

Figura 3.10: Sistema do *token-ring*.

O modelo é formado por um processo que contém quatro *threads* periódicas, uma delas representa a rede, e as outras três representam as estações. A representação gráfica destas quatro *threads* pode ser vista na figura 3.10.

```
system TokenRing
end TokenRing;

system implementation TokenRing.i
subcomponents
        p1 : process Process1.i;
end TokenRing.i;

process Process1
end Process1;

process implementation Process1.i
subcomponents
        sta : thread station.i;
        stb : thread station.i;
        stc : thread station.i;
        tnet : thread net.i;
connections
        event port tnet.pass_a -> sta.receive;
        event port tnet.pass_b -> stb.receive;
        event port tnet.pass_c -> stc.receive;
        event port sta.pass -> tnet.receive_a;
        event port stb.pass -> tnet.receive_b;
        event port stc.pass -> tnet.receive_c;
end Process1.i;
```

Listagem 3.3: Declaração do sistema *TokenRing* e processo *Process1*

O processo contém as *threads* de cada estação (*sta, stb* e *stc*) e contém a *thread* da rede *tnet*. Na implementação do processo são feitas as declarações das conexões entre todas as *threads*. A declaração textual do sistema e do processo podem ser vistos na listagem 3.3.

Cada uma das estações é uma instância da implementação da *thread station*, que tem a mesma estrutura para cada uma delas. A rede, por sua vez, é uma instância da implementação *thread net*. As declarações das *threads station* e *net* podem ser vistas na listagem 3.4.

```
thread net
features
        receive_a : in event port;
        receive_b : in event port;
        receive_c : in event port;
        pass_a : out event port;
        pass_b : out event port;
        pass_c : out event port;
properties
        Dispatch_Protocol => Periodic;
end net;

thread station
features
        receive : in event port;
        pass : out event port;
properties
        Dispatch_Protocol => Periodic;
end station;
```

Listagem 3.4: Declarações das *threads net* e *station*

A implementação da *thread* que representa as esta- ções é apresentada na listagem 3.5. Consiste de uma *thread* periódica cujo comportamento possui três estados: *idle* (estação ociosa), *waitt* (estação aguarda pelo *token*) e *cs* (em seção crítica). A comunicação entre as estações e a rede é feita por meio de portas de eventos.

```
thread implementation station.i
        properties
                Period => 10ms;
        annex behavior_specification {**
                states
                idle : initial complete state;
                waitt : complete state;
                cs : state;
                transitions
                idle -[]-> waitt;
                waitt -[receive?]-> cs;
                waitt -[on receive'count = 0]-> waitt;
                cs -[]-> idle{pass!;};
        **};
end station.i;
```

Listagem 3.5: Modelo da *thread* de uma estação *token ring*

A *thread* que implementa a rede é modelada com o mesmo período das estações. Possui três estados referentes à cada estação, sendo no total nove estados. O estado *idlei* representa a chegada do *token* a uma estação *i*, *tokeni* representa que o *token* foi entregue a esta estação e *afteri* representa que o token pode passar para a próxima estação. O modelo pode passar diretamente de *idlei* para *afteri* se o *token* não for entregue à estação. O modelo simplificado da rede pode ser visto na listagem 3.6, na qual constam modelados somente os estados referentes a estação 1 com a transição para estação 2.

```
thread implementation net.i
        properties
                Period => 10ms;
        annex behavior_specification {**
                states
                idle1 : initial complete state;
                token1 : complete state;
                after1 : complete state;
                idle2 : complete state;
                        (...)
                transitions
                idle1 -[]-> token1{pass_a!;};
                token1 -[receive_a?]-> after1;
                token1 -[on receive_a'count = 0]-> token1;
                idle1 -[]-> after1;
                after1 -[]-> idle2;
                        (...)
        **};
end net.i;
```

Listagem 3.6: Modelo da *thread* da rede do modelo *token ring*

Por meio deste exemplo é possível observar a declaração de um modelo completo em AADL, com sua arquitetura e comportamento. Este mesmo modelo será utilizado no capítulo 4 para exemplificar as ferramentas desenvolvidas neste trabalho.

3.2 Ferramentas Para Modelagem em AADL

Um aspecto essencial em qualquer linguagem são as ferramentas que lhe oferecem suporte. No caso da AADL, a SAE disponibiliza a ferramenta OSATE, que consiste em um conjunto de *plug-ins* que executam sobre a plataforma *Eclipse*. O editor gráfico para modelagem em AADL, conhecido como ADELE, é produzido pelo projeto Topcased e integrado no *plug-in* OSATE. Entre os recursos do OSATE são oferecidas as seguintes funcionalidades [39]: editor para forma textual da AADL, editor para a forma em XML da AADL, conversor de XML para textual, conversor para a linguagem *MetaH* e *plug-ins* para checagem de consistência da arquitetura. Entre estes recursos da ferramenta OSATE, podem ser destacadas as seguintes funcionalidades:

- Análise de modelagem de erros - anexo *Error Model*:

  Este anexo para AADL [36] define uma sublinguagem que possibilita especificar informação relacionada a segurança de um modelo, como pressupostos de falha e reparo, propagação de erros e política de tolerância a falhas.

- Análise de Alocação de Recursos

  Possibilita a previsão e análise de recursos para processador, memória e largura de banda. Leva em conta fatores como o período das *threads*, *deadline*, capacidade do processador e política de escalonamento.

- Checagem de conexões

  Ao utilizar a propriedade *Required_Connection*, define-se que uma porta deve ter uma conexão durante toda a execução do modelo independente do modo de operação ativo. A análise de conexões possibilita checar se estas conexões são mesmo fornecidas.

- Checagem do Nível de Segurança

  O nível de segurança pode ser definido para cada componente por meio da propriedade *SecurityLevel*. Se esta propriedade não é declarada, seu valor é definido como zero, o menor nível de segurança. Esta ferramenta permite analisar se um componente tem o maior nível de segurança dentre os seus subcomponentes e checa nas conexões quais os componentes transmissores possuem um nível de segurança menor ou igual ao dos receptores.

- Análise do Nível de Segurança de Criticalidade

  Permite checar se um transmissor possui um nível de segurança menor do que o receptor, para que um componente não seja controlado por um outro componente com um nível de segurança menor.

- Análise de Latência em Fluxos

  Possibilita comparar o valor de latência de implementações de fluxo de um componente com o correspondente valor da especificação do fluxo deste componente. A latência para uma implementação de fluxo é determinada pela soma de todas as latências das conexões neste fluxo, que são definidas pela propriedade *Latency*.

- Geração de MetaH

  Possibilita a transformação de um modelo AADL em um modelo MetaH (linguagem que originou AADL).

- Checagem de Inversão de Prioridades

  Permite analisar se, considerando-se *threads* periódicas, não há a inversão de prioridades na chamada do processador. A prioridade para uma *thread* é definida por meio da propriedade *SEI::Priority*.

- Análise de Escalonabilidade

  Possibilita analisar a escalonabilidade por meio de um algoritmo de análise de *rate monotonic*.

Neste trabalho é utilizado o editor gráfico produzido pelo projeto Topcased (ADELE) e os editores do OSATE, mas as ferramentas de análise construídas para AADL não são utilizadas.

3.3 Verificação de Modelos AADL no Projeto Topcased

Na seção 3.2 foram apresentadas as ferramentas para modelar sistemas utilizando a linguagem AADL e as ferramentas para análise destes modelos. Apresentou-se ainda, no início deste capítulo, uma das vantagens das linguagens de descrição de arquitetura, que é a possibilidade de se empregar a engenharia dirigida a modelos (MDE), utilizando-se a transformação de modelos. Essa característica na linguagem AADL é amplamente utilizada e permite a transformação para modelos com um formalismo matemático, que possibilitam o emprego das técnicas de verificação formal de propriedades.

Dentro do contexto do projeto Topcased, ao qual este trabalho está inserido, o processo de verificação formal de propriedades de modelos AADL se dá utilizando a transformação de modelos em uma cadeia de verificação conforme exibido na figura 3.11 [1]. Primeiramente, os modelos AADL são transformados para uma linguagem intermediária conhecida como Fiacre. Esta transformação é feita por meio de um compilador desenvolvido nos trabalhos [16][1].

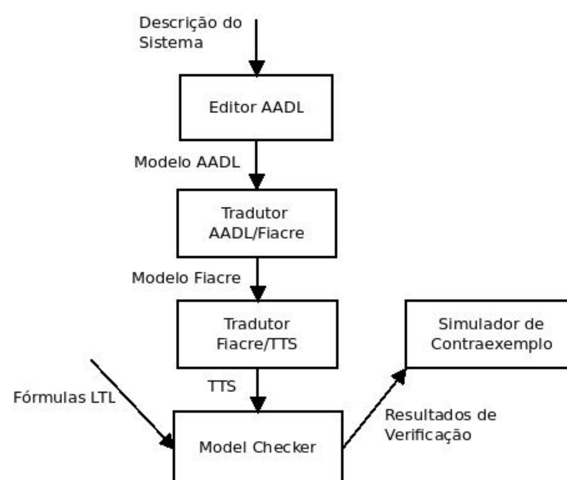

Figura 3.11: Cadeia de verificação no projeto Topcased.

Na sequência, os modelos em Fiacre são transformados em modelos de ainda mais baixo nível, os sistemas TTS (*Time Transition System*). O modelo em TTS, por sua vez, é

transformado em uma estrutura de Kripke, que é uma das entradas da ferramenta de *model-checking*. A ferramenta utilizada com esse propósito é o SELT (*State/Event LTL model-checker*), apresentada no capítulo 2, que confronta a estrutura de Kripke de um modelo TTS com propriedades de verificação no formato LTL.

Os resultados da verificação são gerados no nível dos sistemas TTS, como apresentado na seção 2.2. Estes representam o resultado de propriedade e um contraexemplo para as propriedades falsas.

Nas seções seguintes será detalhada a linguagem intermediária Fiacre e o funcionamento da cadeia de verificação, ressaltando-se a análise dos resultados de verificação.

3.3.1 Linguagem Fiacre

Fiacre [4] (formato intermediário para arquiteturas de componentes distribuídos embarcados) é uma linguagem de domínio específico desenvolvida no contexto do projeto Topcased com o propósito de ser uma linguagem intermediária na cadeia de verificação.

Esta linguagem traz algumas facilidades no processo de transformação de modelos, pois a tradução de modelos feitos em linguagens de alto nível (como UML, AADL, entre outras) para Fiacre é mais fácil de que destas linguagens diretamente para os formalismos matemáticos utilizados pelas ferramentas de verificação.

As ferramentas da cadeia de verificação apresentada na figura 3.11 funcionam independentemente, contudo nos trabalhos [32] e [31] foi desenvolvido um pacote que integra as ferramentas do nível Fiacre ao nível do *model-checker* e permite a edição de modelos na linguagem Fiacre, com o objetivo de facilitar e automatizar a verificação É uma poderosa linguagem para a modelagem de sistemas de tempo real, pois possibilita a representação do comportamento e aspectos temporais de um sistema.

Os programas Fiacre são entendidos como sistemas de transição de estados e podem ser sintetizados em dois elementos principais: processos e componentes. Processos representam o comportamento sequencial de elementos independentes. São definidos como um conjunto de estados e transições, com acesso a variáveis e portas, podendo ser declaradas guardas com controle de fluxo. Componentes representam o sistema com uma organização hierárquica de processos e componentes. São definidos como uma composição paralela de processos e outros componentes, que pode ser síncrona ou não. Fiacre é uma linguagem que usa o conceito de tipos de dados, exigindo que haja uma correspondência entre os dados e entre as portas de comunicação para que não haja erros em tempo de execução.

3.3.2 Funcionamento da Cadeia de Verificação

Nesta seção serão explicados alguns conceitos a respeito da transformação de modelos na cadeia de verificação. Na tradução de modelos AADL em Fiacre, as *threads* e *devices* AADL são transformados em processos Fiacre, o sistema que os contém é transformado em um *component*, enquanto que os *data types* são transformados em *types*. Estes processos Fiacre, por sua vez, não se comunicam diretamente entre si, mas sim, por meio de um outro processo denominado *GLUE*. O processo *GLUE* serve para gerenciar o disparo e comunicação entre as *threads*. A figura 3.12 ilustra como se dá a comunicação entre os processos do sistema transformado.

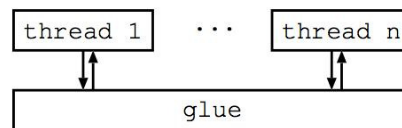

Figura 3.12: Comunicação entre os processos Fiacre transformados de um modelo AADL.

No momento da transformação de modelos, um estado de um componente do modelo AADL pode ser desmembrado em alguns estados Fiacre, devido a questões relativas ao disparo das *threads*. Nos itens abaixo segue um exemplo retirado do modelo do *token-ring*, que contém uma *thread* e o processo no qual é transformada, ressaltando a questão dos estados desmembrados. Nota-se que a *thread* contém os estados *idle*, *waitt* e *cs*, mas o processo, além destes, contém os estados que são desmembrados.

- *Thread* AADL:

```
thread implementation station.i
        annex behavior_specification {**
                states
                idle : initial complete state;
                waitt : complete state;
                cs : state;
```

Listagem 3.7: Simplificação da *Thread* de uma estação do *token-ring*

- Processo Fiacre:

```
process station[F_dispatch_ : in 0..1,F_complete_ : out none ...]
    is
        states idle, F_c_idle, waitt, F_c_waitt, cs, st_54087,
                st_54641, st_55273, st_55880, st_56205
        var receive : 0..1 := 0
```

Listagem 3.8: Simplificação do Processo referente a uma estação do *token-ring*

Na etapa seguinte, todos os estados dos processos Fiacre são transformados em lugares do sistema TTS. Alguns exemplos dos lugares do sistema TTS podem ser observados abaixo:

```
pl station_1_sidle (1)
pl station_1_sF__c__idle (0)
pl station_1_swaitt (0)
pl station_1_sF__c__waitt (0)
pl station_1_sst__54087 (0)
```

O processo de verificação segue a mesma sequência apresentada na seção 2.2 que apresenta o *model-checker* SELT. O SELT toma como uma das entradas a estrutura de Kripke referente ao sistema TTS e como outra entrada as fórmulas lógicas no formato LTL.

Propriedades de Verificação As propriedades do sistema devem ser especificadas no nível do sistema TTS, por meio de fórmulas lógicas em LTL. Desta forma, é necessário que o usuário acompanhe todo o processo de transformação de modelos e pesquise no sistema TTS qual é a correspondência entre os estados dos componentes do modelo AADL.

Resultados da Verificação Os resultados de verificação provenientes do SELT dizem respeito ao sistemas TTS, o que gera algumas dificuldades para analisar estes resultados e os contraexemplos, visto que os nomes apresentados não são os do modelo AADL. Um modelo AADL de um sistema de grande porte pode possuir muitos componentes, e assim muitos estados, e o fato de estes estados serem ainda desmembrados na transformação de modelos faz com que os contraexemplos se tornem ilegíveis do ponto de vista do usuário. Mesmo para os sistemas de pequeno porte é comum que os contraexemplos apresentem centenas de passos de execução.

Entretanto, os resultados da verificação podem ser analisados pelo simulador de contraexemplos da ferramenta TINA, como explicado na seção 2.2. A correspondência entre estes resultados, que dizem respeito ao sistema TTS, e o modelo AADL em questão deve ser feita manualmente pelo usuário.

3.4 A Linguagem BLESS – Uma Alternativa para a Verificação de Modelos

Esta seção apresenta uma visão geral sobre a linguagem BLESS [27], uma proposta para assegurar o correto funcionamento de um modelo AADL, como uma alternativa à verificação de modelos. O detalhamento desta linguagem pode ser visto no anexo A. Esta linguagem é

utilizada em um estudo de caso de um sistema de marcapasso, que é apresentado no anexo B. Este estudo de caso possibilita compará-la com as ferramentas apresentadas nesta dissertação, o que será melhor discutido no capítulo 5.

A linguagem BLESS teve a sua criação motivada em escrever modelos matematicamente consistentes de sistemas. Desenvolvida por *Multitude Corporation*, esta linguagem é definida como uma sublinguagem de anexo para AADL. Se baseia em conceitos da matemática discreta como lógica proposicional [41] e prova automática de teoremas [20]. A grande contribuição desta linguagem é permitir que sejam desenvolvidos modelos corretos, que cumprem as suas especificações.

É dividida em três sublinguagens: *Assertion* define asserções e declara propriedades do sistema sobre o tempo, utilizando lógica de primeira ordem, *subBLESS* serve como anexo comportamental para subprogramas e *BLESS* serve como anexo comportamental para *threads*. Estas três sublinguagens interagem entre si em um modelo AADL, o que permite que o mecanismo de provas possa utilizar as declarações do modelo para provar o seu correto funcionamento.

A ferramenta para se trabalhar com a linguagem é integrada ao plug-in Eclipse OSATE. Esta ferramenta faz uso de um mecanismo de prova que, juntamente com as suas sublinguagens, objetiva verificar automaticamente a correção de um dado modelo AADL, utilizando a prova automática de teoremas. Entretanto, a prova para cada componente do modelo AADL é feita individualmente, não sendo considerada a comunicação entre eles.

3.5 O Projeto COMPASS

Esta seção apresenta uma ferramenta para modelagem, validação e verificação de modelos AADL denominada COMPASS[1] [8] (*Correctness, Modeling, and Performance of Aerospace Systems*), promovida pela *European Space Agency* (ESA).

COMPASS utiliza um conjunto de ferramentas já existentes, como o *model-checker* NuSMV [12] e o *model-checker* MRMC [25]. As entradas para estas ferramentas são obtidas por meio da transformação de modelos e padrões de especificação de requisitos. Estes padrões são utilizados como uma forma de capturar requisitos do sistema.

Esse projeto faz uso do anexo *Error Model* [36] para AADL, inserindo neste elementos adicionais para permitir a validação e verificação formais. Com isso, a linguagem resultante dispõe de algumas funcionalidades. Permite a modelagem do comportamento de falhas, descrevendo falhas de *hardware* e *software*, propagação de erros, falhas permanentes e

[1] http://compass.informatik.rwth-aachen.de/

esporádicas, bem como modos de falha. Permite também especificar o comportamento temporal do sistema, além de modelar aspectos probabilísticos como falhas randômicas e reparos e requisitos de observabilidade, essenciais para análises de FDIR (Fault Detection, Isolation and Recovery).

Por meio das ferramentas disponibilizadas, o modelo completo do sistema compreende três diferentes partes, o modelo nominal (sistema em operação normal), o comportamento de erros e a especificação de injeção de falhas, que define como o comportamento de erros influencia o modelo nominal do sistema.

Especificação das Propriedades de Verificação Os requisitos do sistema devem ser especificados como propriedades de verificação, segundo um sistema de padrões de propriedades definido em [17]. Estes padrões de propriedades servem para auxiliar o processo de capturar requisitos de segurança, correção, desempenho e confiabilidade.

Validação dos Requisitos Para se assegurar a qualidade dos requisitos, estes podem ser validados independentemente do sistema. Para isto os requisitos são convertidos automaticamente para LTL ou CTL. Esta análise inclui consistência de propriedades (testa se os requisitos não são mutuamente exclusivos), asserção de propriedades (testa onde uma asserção é uma consequência dos requisitos), possibilidade de propriedade (checa a compatibilidade entre as possibilidades e os requisitos). Estas análises permitem assegurar que os requisitos são adequados e consistentes.

Verificação Funcional Logo após a validação dos requisitos, estes, juntamente com o modelo AADL, são as entradas para a análise de correção operacional. Consiste na verificação de que o sistema irá operar corretamente em relação ao conjunto de requisitos funcionais, sob a hipótese de ausência de falhas de *hardware* e *software*.

Segurança e Confiabilidade Esta análise serve para investigar o comportamento do sistema em condições de falha, ou seja, quando alguma parte do sistema não tem o seu correto funcionamento devido a falhas. O objetivo é assegurar que o sistema cumpra com seus requisitos de segurança nestas condições. Para isso, o pacote de ferramentas gera alguns artefatos, como árvore de falhas e tolerância a falhas.

Garantia de Desempenho Esta análise serve para garantir o correto desempenho do sistema, mesmo na presença de falhas. Os resultados desta análise podem ser utilizados em decisões de projeto referente a custos e riscos. A análise se baseia no modelo AADL estendido com o seu modelo de erros e injeção de falhas. O modelo estendido pode ser transformado no seu correspondente modelo de Markov e utilizado com a ferramenta MRMC.

Detecção de Falhas, Isolamento e Reparo (FDIR) Os modelos de um sistema podem incluir a descrição sobre a detecção de falhas e isolamento de subsistemas, assim como as ações para reparo a serem tomadas. Com base nisto, são providenciadas análises em como as falhas são detectadas, onde os requisitos de observabilidade são suficientes para diagnóstico do sistema, como os sistemas FDIR isolam as falhas e como o reparo deve agir.

Visualização dos Resultados de Verificação As ferramentas disponibilizadas pelo projeto COMPASS permitem a visualização dos resultados de verificação e simulação de contraexemplos. Nas simulações é permitido que o usuário escolha a transição que deve ser disparada. Os contraexemplos podem ser gerados com ou sem a injeção de falhas no sistema.

As ferramentas desenvolvidas pelo projeto COMPASS foram validadas por meio de dois estudos de caso, a estratégia de FDIR de um satélite e um sistema de regulação térmica em um satélite. Estes estudos de caso mostraram que AADL proporciona uma expressividade suficiente para modelar os componentes de *software* e *hardware* de satélites e sistemas aviônicos.

3.6 Conclusão

Este capítulo apresentou a definição das linguagens de descrição de arquitetura e a definição da linguagem AADL detalhando todos os seus elementos, mostrando que a linguagem AADL é uma ferramenta poderosa para modelagem de sistemas embarcados de tempo real.

Este capítulo relacionou dois trabalhos desenvolvidos com o mesmo foco de garantir a corretude de modelos AADL, o projeto COMPASS, que faz a integração de ferramentas conhecidas de análise e a linguagem BLESS, que utiliza um mecanismo de prova de teoremas que difere das técnicas de verificação e servirá como parâmetro de comparação com as ferramentas desenvolvidas neste trabalho.

No que concerne ao projeto Topcased, apesar da existência das ferramentas de análise, o processo de verificação formal de propriedades na linguagem AADL ainda está em fase de desenvolvimento. Este processo consiste em uma cadeia de verificação que conta com algumas ferramentas de compilação e tradução de modelos, e também com uma ferramenta de *model-checking*. Nesta cadeia de verificação é empregada uma linguagem intermediária para facilitar a tradução de modelos, a linguagem Fiacre. Uma das limitações desta cadeia de verificação é que não existe uma maneira de se especificarem as propriedades de verificação no nível da linguagem AADL, somente no nível dos sistemas TTS, obrigando o usuário a acompanhar a tradução dos modelos e a transformação dos nomes dos componentes e estados. Outra limitação é que os resultados da verificação não podem ser visualizados no nível AADL, somente no nível TTS. Sendo estes os pontos que este trabalho visa atacar.

Capítulo 4

Contribuições para Especificação de Propriedades e Análise de Resultados de Verificação em AADL

O capítulo anterior nos permite concluir que as ferramentas empregadas no projeto Topcased para modelagem e verificação de sistemas não permitem a especificação de propriedades em alto nível. Estas devem ser especificadas em um nível mais baixo de abstração, ou seja, um nível posterior na transformação de modelos. As propriedades são tipicamente especificadas utilizando lógica temporal, exigindo do usuário este conhecimento como pré-requisito para utilização destas ferramentas. Com isso, este projeto se propõe a proporcionar uma forma de especificar as propriedades de verificação no nível da linguagem AADL, sem utilizar diretamente a lógica temporal.

Como a AADL é uma linguagem de alto nível, ao modelar um sistema é desejável que seus requisitos possam ser capturados de uma forma clara e facilitada. Para isto, esta dissertação utiliza um sistema de padrões de propriedades que se baseia em um subconjunto da linguagem natural como forma de expressar o comportamento desejado do modelo. Este sistema de padrões, descrito na próxima seção, é inspirado no trabalho de Dwyer et al. [17], onde o autor afirma que a maior parte das propriedades utilizadas na verificação de modelos se encaixam nestes padrões. Estes padrões são referentes a propriedades de segurança, resposta, alcançabilidade, entre outras, que têm formatos pré-definidos de fórmulas em lógica temporal.

No entanto, somente alguns destes padrões podem ser considerados os mais importantes como propriedades de um sistema. Com isso, nesta dissertação, estes padrões são redefinidos em uma nova classificação, buscando as propriedades mais utilizadas na verificação. Estes

padrões são utilizados para traduzir os requisitos do sistema em propriedades de verificação utilizando-se a linguagem natural, sem fazer menção à lógica temporal, ou seja, excluindo a exigência de que o usuário a conheça previamente.

Com o objetivo de integrar a especificação de propriedades no nível AADL com a cadeia de verificação do projeto Topcased, desenvolveu-se neste trabalho uma ferramenta que faz a tradução destes padrões de propriedades. Esta ferramenta executa a transformação destes padrões diretamente para lógica temporal (LTL), que é o formato de entrada usado pela ferramenta de verificação empregada.

Para auxiliar a especificação de propriedades, a ferramenta desenvolvida oferece um assistente de escrita de propriedades, onde o usuário pode selecionar o padrão desejado e inserir o nome dos respectivos elementos do modelo AADL que se pretende verificar.

Por fim, uma parte do pacote de ferramentas desenvolvidas neste trabalho oferece recursos para visualizar, analisar e simular os resultados de verificação diretamente no nível AADL. Estes resultados são gerados pela ferramenta de *model-checking* (SELT) e integrados no nível AADL, de forma que os contraexemplos possam ser simulados.

Na figura 4.1 pode-se observar como as ferramentas desenvolvidas, em destaque na figura, se integram na cadeia de verificação do projeto Topcased. Os requisitos do sistema são transformados em padrões de propriedades com o auxílio do **assistente de propriedades** e, em seguida, traduzidos para o formato LTL, por meio do tradutor de propriedades. Os resultados de verificação, por sua vez, são apresentados no nível AADL pelas ferramentas de análise dos resultados, que compreendem uma janela com o resultado de cada propriedade, uma com os componentes do modelo AADL, uma com o contraexemplo para cada propriedade tida como falsa, e um simulador de contraexemplos que faz a integração entre estas janelas.

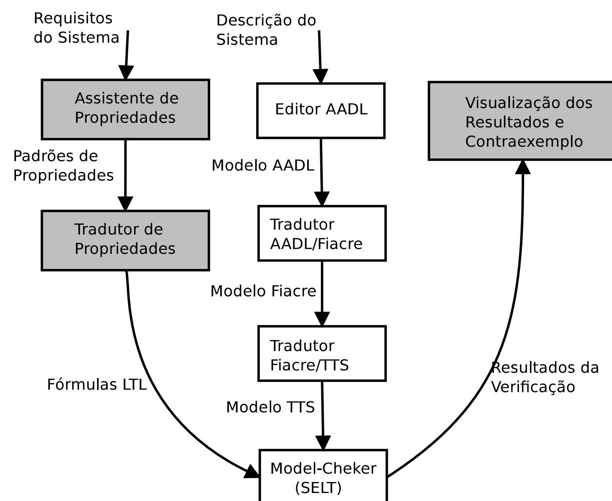

Figura 4.1: Cadeia de verificação do Topcased integrada com as ferramentas desenvolvidas.

4.1 Sistema de Padrões de Propriedades

Com o objetivo de facilitar a especificação de propriedades este trabalho utiliza uma classificação das fórmulas lógicas utilizadas na verificação em um sistema de padrões de propriedades. Esta classificação é inspirada no trabalho [17], que tenta cobrir o número máximo de fórmulas lógicas com estes padrões. Contudo, nesta dissertação, são feitas algumas modificações nesta classificação, visando cobrir somente as propriedades mais importantes na verificação.

4.1.1 Padrões na Especificação de Propriedades

Esta seção resume um trabalho, no qual esta dissertação é inspirada, intitulado Padrões na Especificação de Propriedades para Verificação de Estados Finitos [17]. Neste trabalho é feita uma proposta de definição de padrões de propriedades de verificação, visando cobrir a maior parte das propriedades utilizadas em processos de verificação. Seu o objetivo é facilitar o processo de especificação de propriedades, de uma forma geral, ao se empregar a verificação formal de modelos.

Foram definidos oito padrões de propriedades divididos em duas categorias, que os classificam como propriedades de ocorrência, citando ausência, existência, existência limitada e universalidade ou de ordem, citando precedência, resposta, precedência de cadeia e resposta de cadeia. A figura 4.2 ilustra essa classificação.

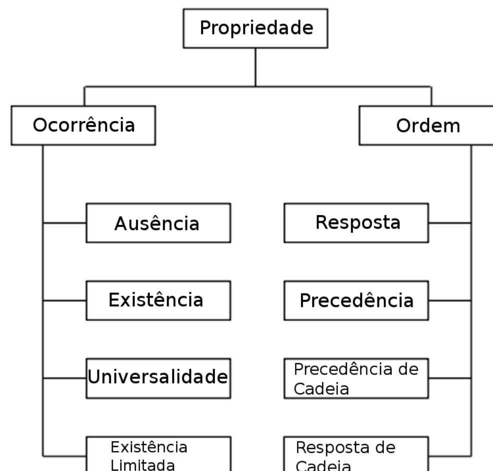

Figura 4.2: Padrões na Especificação de Propriedades.

Para cada uma destas propriedades é definido um conjunto de cinco diferentes escopos de análise, que delimitam em que parte da execução do modelo a propriedade deve ser considerada:

- Globalmente (*Globally*)
- Antes (*Before*)
- Depois (*After*)
- Entre (*Between*)
- Enquanto não (*After Until*)

Neste mesmo trabalho, foi coletado um conjunto de 555 de propriedades, retiradas de *papers* e outros projetos. Estas propriedades foram então submetidas à classificação sugerida, sendo possível observar que somente um pequeno número delas não se encaixa em algum destes padrões (8%) e que os padrões com maior número de ocorrência são, respectivamente, os de resposta, universalidade, ausência, precedência e existência. Por outro lado, os padrões que tem uma utilização muito baixa e que serão desconsiderados na nova classificação são os de existência limitada, precedência de cadeia e resposta de cadeia.

4.1.2 Proposta para Sistema de Padrões de Propriedades

A partir da seção anterior, pode-se concluir que há um conjunto limitado de propriedade que podem ser considerado como relevante para a verificação. Por outro lado, dentre estas propriedades, nem todas são consideradas como realmente importantes neste processo. Para garantir que um determinado sistema não apresente uma situação indesejável, é comum a especificação de propriedades de verificação que busquem a garantia de segurança, no sentido de *safety*. Além disso, visando assegurar que um sistema apresente um comportamento desejável, é comum especificar propriedades de vivacidade, no sentido de *liveness*. A partir disto, e tendo como inspiração o trabalho apresentado na seção anterior, esta seção reformula a classificação destas propriedades, relacionando as mais importantes para o processo de verificação.

Nesta proposta, estas propriedades são divididas em seis categorias (vide tabela 4.1): *ausência, existência, justiça, universalidade, resposta* e *precedência*. Cada uma destas categorias é dividida em possíveis escopos de análise dentro da execução do modelo, como demonstrado na tabela.

Ausência - Global - Antes - Depois - Entre - Enquanto não	Universalidade - Global - Antes - Depois - Entre - Enquanto não
Existência - Global - Antes - Depois - Entre - Enquanto não	Resposta - Global - Antes - Depois - Entre - Enquanto não
Justiça - Global - Antes - Depois - Entre - Enquanto não	Precedência - Global - Antes - Depois - Entre - Enquanto não

Tabela 4.1: Padrões de Propriedades

Nesta nova classificação é incluída a propriedade de justiça (*fairness*), tendo em vista a sua importância na verificação [9]. Por outro lado, além dos escopos temporais, que foram mantidos, observou-se a necessidade da inclusão das opções para especificar a propriedade livre de bloqueio e alcançabilidade.

No processo de verificação formal, as propriedades especificadas varrem todas as possibilidades de execução do modelo em busca de proposições que representam os estados dos componentes do modelo AADL. Estes estados dos componentes possuem seus nomes definidos no modelo AADL, e para serem inseridos nos padrões de propriedades devem seguir uma sintaxe específica definida na seção 4.2.1, que trata sobre o tradutor de propriedades. Nas seções seguintes estes estados são denominados com as letras maiúsculas P, Q, R e S a título de exemplo, e os operadores temporais globalmente (G), eventualmente (F) e próximo (X) são representados respectivamente pelos símbolos $\Box$, $\Diamond$ e $\bigcirc$, utilizados no processo de tradução dos padrões de propriedades.

4.1.2.1 Propriedade de Ausência

Esta propriedade pode ser utilizada quando se deseja garantir a segurança de um sistema, como por exemplo, a exclusão mútua ou a ausência de bloqueio. As propriedades relacionadas com este padrão, na literatura, são conhecidas como *safety* [9]. As propriedades de ausência podem ser utilizadas quando se deseja constatar que um determinado estado não pode ser atingido em um dado momento. Nos itens que seguem são apresentados a sintaxe da definição da propriedade, seu significado e a sua fórmula em LTL.

- *Deadlock-Freeness*:

  Sintaxe: Absence.Deadlock

  Significado: A condição de *deadlock* nunca é atingida.

  Fórmula em LTL: $\Box\, \neg dead$ (*dead* é uma palavra reservada do *model-checker* SELT)

- *Globally*:

  Sintaxe: Absence.Globally(P)

  Significado: P não é atingido em toda a execução do modelo.

  Fórmula em LTL: $\Box\, \neg P$

- *Before*:

  Sintaxe: Absence.Before(P, Q)

  Significado: P não é atingido antes do sistema atingir Q.

  Fórmula em LTL: $\Diamond Q \Rightarrow (\neg P\ U\ Q)$

- *After*:

  Sintaxe: Absence.After(P, Q)

  Significado: P não ocorre depois do sistema atingir Q.

  Fórmula em LTL: $\Box(Q \Rightarrow \Box(\neg P))$

- *Between*:

  Sintaxe: Absence.Between(P, Q, R)

  Significado: P não ocorre entre Q e R.

  Fórmula em LTL: $\Box((Q \bigwedge \neg R \bigwedge \Diamond R) \Rightarrow (\neg P\ U\ R))$

- *After Until*:

  Sintaxe: Absence.AfterUntil(P, Q, R)

  Significado: depois da ocorrência de Q, P não pode ser atingido enquanto R não ocorrer.

  Fórmula em LTL: $\Box(Q \bigwedge \neg R \Rightarrow (\neg P\ W\ R))$

4.1.2.2 Propriedade de Existência

Como parte deste padrão estão classificadas a propriedade de existência, que afirma que algo certamente é atingido, e a propriedade de alcançabilidade, que afirma que algo pode ser atingido. Com isso, pode-se constatar que a alcançabilidade é mais fraca que a existência, apesar de serem mantidas na mesma categoria. Nota-se que a lógica LTL não permite a

representação da propriedade de alcançabilidade, possibilitando somente a representação da não alcançabilidade. Contudo, como LTL é a lógica temporal empregada nesta dissertação, a representação desta propriedade é mantida como a não alcançabilidade. Esta propriedade tem o mesmo significado da propriedade de ausência global, tendo, inclusive, a mesma fórmula lógica. Abaixo é apresentada a sintaxe da definição destas propriedades, seu significado e a sua fórmula em LTL.

- *(Un)Reachability*:

  Sintaxe: Unreachable(P)

  Significado: P é inatingível. Se esta propriedade resultar como falsa significa que P é atingível.

  Fórmula em LTL: $\Box \neg P$

- *Globally*:

  Sintaxe: Existence.Globally(P)

  Significado: P certamente é atingido em algum momento da execução do modelo.

  Fórmula em LTL: $\Diamond P$

- *Before*:

  Sintaxe: Existence.Before(P, Q)

  Significado: P certamente é atingido antes do sistema atingir Q.

  Fórmula em LTL: $\neg Q\ W(P \bigwedge \neg Q)$

- *After*:

  Sintaxe: Existence.After(P, Q)

  Significado: P certamente é atingido depois do sistema atingir Q.

  Fórmula em LTL: $\Box(\neg Q) \bigvee \Diamond(Q \bigwedge \Diamond P))$

- *Between*:

  Sintaxe: Existence.Between(P, Q, R)

  Significado: P certamente é atingido entre Q e R.

  Fórmula em LTL: $\Box(Q \bigwedge \neg R \Rightarrow (\neg R\ W\ (P \bigwedge \neg R)))$

- *After Until*:

  Sintaxe: Existence.AfterUntil(P, Q, R)

  Significado: depois da ocorrência de Q, P certamente é atingido enquanto R não ocorrer.

  Fórmula em LTL: $\Box(Q \bigwedge \neg R \Rightarrow (\neg R\ U\ (P \bigwedge \neg R)))$

4.1.2.3 Propriedade de Justiça

Esta propriedade expressa que algo ocorre infinitamente muitas vezes. Abaixo é apresentada a sintaxe da definição da propriedade, seu significado e a sua fórmula em LTL.

- *Globally*:

  Sintaxe: Fairness.Globally(P)

  Significado: P ocorre infinitamente muitas vezes durante toda a execução do modelo.

  Fórmula em LTL: $\Box\Diamond P$

- *Before*:

  Sintaxe: Fairness.Before(P, R)

  Significado: P ocorre infinitamente muitas vezes antes do sistema atingir R.

  Fórmula em LTL: $\Diamond R \Rightarrow (\Box\Diamond P \; U \; R)$

- *After*:

  Sintaxe: Fairness.After(P, Q)

  Significado: P ocorre infinitamente muitas vezes depois do sistema atingir Q.

  Fórmula em LTL: $\Box(Q \Rightarrow \Box\Diamond(P))$

- *Between*:

  Sintaxe: Fairness.Between(P, Q, R)

  Significado: P ocorre infinitamente muitas vezes entre Q e R.

  Fórmula em LTL: $\Box((Q \bigwedge \neg R \bigwedge \Diamond R) \Rightarrow (\Box\Diamond P \; U \; R))$

- *After Until*:

  Sintaxe: Fairness.AfterUntil(P, Q, R)

  Significado: depois da ocorrência de Q, P ocorre infinitamente muitas vezes enquanto o estado R não ocorrer.

  Fórmula em LTL: $\Box(Q \bigwedge \neg R \Rightarrow (\Box\Diamond P \; W \; R))$

4.1.2.4 Propriedade de Universalidade

O significado deste padrão é que determinado estado de um componente está habilitado durante todo o escopo de análise. Abaixo é apresentada a sintaxe da definição da propriedade, seu significado e a sua fórmula em LTL.

- *Globally*:

  Sintaxe: Universality.Globally(P)

  Significado: P está habilitado durante toda a execução do modelo.

  Fórmula em LTL: $\Box P$

- *Before*:

  Sintaxe: Universality.Before(P, R)

  Significado: P está habilitado antes do sistema atingir R.

  Fórmula em LTL: $\Diamond R \Rightarrow (P\ U\ R)$

- *After*:

  Sintaxe: Universality.After(P, Q)

  Significado: P está habilitado depois do sistema atingir Q.

  Fórmula em LTL: $\Box(Q \Rightarrow \Box(P))$

- *Between*:

  Sintaxe: Universality.Between(P, Q, R)

  Significado: P está habilitado entre Q e R.

  Fórmula em LTL: $\Box((Q \bigwedge \neg R \bigwedge \Diamond R) \Rightarrow (P\ U\ R))$

- *After Until*:

  Sintaxe: Universality.AfterUntil(P, Q, R)

  Significado: depois da ocorrência de Q, P é habilitado enquanto R não ocorrer.

  Fórmula em LTL: $\Box(Q \bigwedge \neg R \Rightarrow (P\ W\ R))$

Esta propriedade, no entanto, não está habilitada no assistente de propriedades, visto que o seu resultado da verificação não é correto. Pois, como apresentado na seção 3.3.2, no momento da transformação de modelos, os estados dos compoentes AADL são desmembrados em mais estados. Desta forma, a ferramenta de *model-checking* não consegue identificar quais são os estados que derivam do estado original. Logo, mesmo que o modelo se mantenha em um destes estados desmembrados, que correspondem ao estado original, o *model-checker* assume como um estado diferente, invalidando a utilização deste padrão de propriedade.

4.1.2.5 Propriedade de Precedência

O significado deste padrão é que determinado estado de um componente somente pode ser atingido depois da ocorrência de algum outro estado dentro do escopo de análise. Abaixo é apresentada a sintaxe da definição da propriedade, seu significado e a sua fórmula em LTL.

- *Globally*:

  Sintaxe: Precedence.Globally(S, P)

  Significado: P somente pode ser atingido depois da ocorrência de S.

  Fórmula em LTL: $\neg P \; W \; S$

- *Before*:

  Sintaxe: Precedence.Before(S, P, R)

  Significado: S sempre é atingido antes de P, até a ocorrência de R.

  Fórmula em LTL: $\Diamond R \Rightarrow (\neg P \; U \; (S \bigvee R))$

- *After*:

  Sintaxe: Precedence.After(S, P, Q)

  Significado: P é sempre falso antes da ocorrência de S, depois que Q é atingido.

  Fórmula em LTL: $\Box \neg Q \bigvee \Diamond (Q \bigwedge (\neg P \; W \; S))$

- *Between*:

  Sintaxe: Precedence.Between(S, P, Q, R)

  Significado: Entre a ocorrência de Q e R, P somente pode ser atingido depois da ocorrência de S.

  Fórmula em LTL: $\Box ((Q \bigwedge \neg R \bigwedge \Diamond R) \Rightarrow (\neg P \; U \; (S \bigvee R)))$

- *After Until*:

  Sintaxe: Precedence.AfterUntil(S, P, Q, R)

  Significado: Entre a ocorrência de Q e R, P somente pode ser atingido depois da ocorrência de S, mas nesse caso R não precisa necessariamente ser atingido.

  Fórmula em LTL: $\Box (Q \bigwedge \neg R \Rightarrow (\neg P \; W \; (S \bigvee R)))$

4.1.2.6 Propriedade de Resposta

O significado deste padrão é que determinado estado de um componente implica na ocorrência de algum outro estado dentro de um dado escopo de análise. Esta propriedade pode ser utilizada quando se deseja verificar a vivacidade do modelo (vide seção 2.1.7). Abaixo é apresentada a sintaxe da definição da propriedade, seu significado e a sua fórmula em LTL.

- *Globally*:

  Sintaxe: Response.Globally(S, P)

  Significado: S responde a P, ou seja, P implica na ocorrência de S.

  Fórmula em LTL: $\Box (P \Rightarrow \Diamond S)$

- *Before*:

  Sintaxe: Response.Before(S, P, R)

  Significado: P implica na ocorrência de S, antes do modelo atingir R.

  Fórmula em LTL: $\Diamond R \Rightarrow (P \Rightarrow (\neg R\, U\ (S \bigwedge \neg R)))\ U\ R$

- *After*:

  Sintaxe: Response.After(S, P, Q)

  Significado: P implica na ocorrência de S, depois do modelo atingir R.

  Fórmula em LTL: $\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond S))$

- *Between*:

  Sintaxe: Response.Between(S, P, Q, R)

  Significado: Entre a ocorrência de Q e R, P implica na ocorrência de S

  Fórmula em LTL: $\Box((Q \bigwedge \neg R \bigwedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R\, U\ (S \bigwedge \neg R)))\ U\ R)$

- *After Until*:

  Sintaxe: Response.AfterUntil(S, P, Q, R)

  Significado: Entre a ocorrência de Q e R, P implica na ocorrência de S, mas nesse caso R não precisa necessariamente ser atingido.

  Fórmula em LTL: $[](Q \bigwedge \neg R \Rightarrow ((P \Rightarrow (\neg R\, U\ (S \bigwedge \neg R)))\ W\ R)$

4.2 Assistente para Especificação das Propriedades

O Assistente para especificação de propriedades é uma ferramenta desenvolvida com o objetivo de facilitar a especificação de propriedades por parte do usuário. As propriedades definidas na tabela 4.1 são apresentadas ao usuário em uma estrutura arborescente no assistente, a qual permite que seja selecionada a propriedade desejada de forma clara em um menu lateral, como mostra a figura 4.3. Conforme uma propriedade é selecionada, é apresentada uma janela, contendo o significado da propriedade em uma frase, em linguagem natural, correspondendo à afirmação da fórmula lógica. Nesta interface são habilitadas caixas de texto para a inserção dos estados dos componentes AADL, de forma a completar a especificação da propriedade.

A figura 4.3 exemplifica a especificação da propriedade de ausência com o escopo *before*. Neste exemplo o assistente mostra uma frase para ser completada como a seguinte: *station.2.cs is never reached before net.1.token2*, onde *station.2.cs* e *net.1.token2* são estados dos componentes AADL *station* e *net* inseridos pelo usuário em caixas de texto. A forma correta para inserção dos estados dos componentes AADL é detalhada na próxima seção. A

janela *Process View*, que será detalhada mais a frente, pode ser utilizada para auxiliar nesta tarefa, visto que fornece a informação dos nomes dos estados.

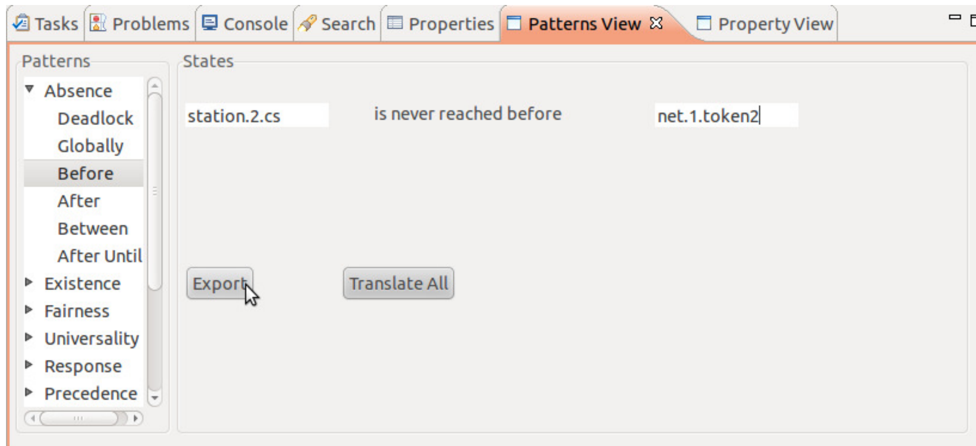

Figura 4.3: Assistente para especificação das propriedades de verificação.

No momento de exportar a propriedade escolhida, o assistente checa a validade dos nomes dos estados dos componentes, evitando que sejam exportados nomes com caracteres inválidos. O assistente não permite que sejam exportados nomes que contenham o caractere *underscore* (_), apesar de ser permitido na linguagem AADL. Esta limitação existe devido ao fato de esse caractere ser considerado no momento da filtragem dos estados para construção da *Process View* e dos contraexemplos, como será discutido posteriormente.

É permitido que sejam inseridas a conjunção ou disjunção de estados na caixa de texto, utilizando-se os respectivos operadores /\ e \/. Um exemplo da utilização da conjunção de dois estados é *station.1.idle* /\ *net.1.token1* e da disjunção é *station.1.idle* \/ *net.1.token1*.

O assistente conta com dois botões. O botão *Export* exporta a propriedade escolhida para um arquivo específico e faz a chamada da classe de tradução das propriedades para LTL. Outro botão existente é o *Translate All* que serve para traduzir todas as propriedades que já foram exportadas sem a necessidade de exportar uma nova.

As propriedades são exportadas para um arquivo específico, e as informações que são gravadas nele são o padrão de propriedade com os estados inseridos e a seu significado em forma de comentário. Um exemplo das propriedades exportadas pode ser visto a seguir:

Absence.Globally(**station.1.cs**) //station.1.cs is never reached
Fairness.Globally(**station.2.waitt**) //station.2.waitt occurs infinitely often globally

4.2.1 Tradutor de Propriedades

As propriedades especificadas devem ser convertidas para o seu formato LTL. Para tanto, desenvolveu-se o chamado tradutor de propriedades. Dois aspectos são levados em conta pelo tradutor: o nome dos estados dos componentes AADL e o padrão de propriedade escolhido. Esses estados dizem respeito ao comportamento de componentes AADL e devem estar declarados em um *annex behavior*, detalhado na seção 3.1.7.

Tradução dos nomes dos estados dos componentes do modelo AADL Para a inserção dos nomes dos estados dos componentes foi criada uma sintaxe que relaciona o nome do componente AADL, a instância desse componente e o estado desejado do comportamento deste componente. Assim esses estados devem ser inseridos da seguinte forma:

Sintaxe para inserção dos estados dos componentes: **component.i.state**

A palavra *component* é o nome do componente AADL onde se encontra o estado. Pode ser uma *thread* ou um *device*. A letra i deve ser um número inteiro que significa a ordem da instância deste componente. Por exemplo, se uma *thread* for instanciada três vezes e o estado desejado estiver contido na terceira instância, o valor de i é 3. Por fim, a palavra *state* é o nome do estado de comportamento desejado.

O tradutor executa a transformação do nome do estado do componente AADL diretamente para um lugar do sistema TTS. Os lugares do sistema TTS que são transformados dos modelos AADL são escritos da seguinte forma:

Lugares nos sistemas TTS: **component_i_sstate**

Os lugares nos sistemas TTS podem ser divididos em três partes, sendo que cada uma se refere a um elemento do modelo AADL. A palavra *component* se refere ao nome do componente AADL onde se encontra o estado. O número representado por i é a ordem da instância deste componente. A palavra *state* se refere ao nome do estado de comportamento.

Tomando como exemplo o modelo do *token-ring* apresentado na seção 3.1.8, a listagem 4.1 mostra como a *thread station* é instanciada três vezes, enquanto que a *thread net* é instanciada uma vez pelo processo *Process1*.

```
process implementation Process1.i
subcomponents
        sta : thread station.i;
        stb : thread station.i;
        stc : thread station.i;
        tnet : thread net.i;
                (...)
```

Listagem 4.1: Declaração de processo destacando as instâncias das *threads*

Assim, se o estados que se desejam inserir são o *idle* de cada uma das *threads station* e o *token1* da *thread net*, o nome dos estados inseridos nas propriedades são:

```
station.1.idle
station.2.idle
station.3.idle
net.1.token1
```

Para este exemplo, a tradução dos nomes destes estados, inseridos segundo a sintaxe criada, resultam nos seguintes lugares do sistema TTS:

```
station_1_sidle
station_2_sidle
station_3_sidle
net_1_stoken1
```

Na próxima seção será detalhada a janela *Process View*, que auxilia na escolha dos nomes dos estados dos componentes do modelo AADL, visto que estes nomes são apresentados nesta janela exatamente com a mesma sintaxe que devem ser inseridos nas propriedades.

Tradução dos padrões de propriedades O tradutor faz a transformação dos padrões de propriedade escolhidos em fórmulas lógicas LTL apresentadas na seção 4.1.2, inserindo os nomes dos lugares do sistema TTS.

Um ponto considerado pelo tradutor é que algumas fórmulas lógicas se utilizam do operador temporal *weak until* (**W**). Entretanto, a ferramenta de *model-checking* empregada não implementa este operador, apesar de possibilitar uma forma de se definir novos operadores juntamente com as fórmulas lógicas. Desta forma, no momento da tradução de cada propriedade que utiliza **W**, este operador é criado automaticamente pelo tradutor, inserindo a seguinte linha no arquivo com as fórmulas LTL:

Definição do operador **W**: $infix\ a\ W\ b = (a\ U\ b) \bigvee ([]a)$;

Exemplificando a tradução dos padrões de propriedade, o padrão de propriedade especificado é o de resposta global, o qual afirma que o estado *cs* da *thread station* sempre responde à ocorrência do estado *waitt*. A tradução para LTL é mostrada logo abaixo da propriedade.

Propriedade de resposta global:	Response.Globally(**station.1.cs**, **station.1.waitt**)
Tradução em LTL:	$\Box(\textbf{station_1_waitt} \Rightarrow \Diamond\textbf{station_1_cs})$

Neste exemplo, o nome do estado é definido como *waitt*, pois o assistente não permite a declaração do nome *wait*, visto que esta é uma palavra reservada em Fiacre.

4.3 Visualização dos Resultados de Verificação e Contraexemplo

Os resultados de verificação devem ser analisados para que seja possível certificar-se de que o sistema cumpre com as suas propriedades. Estes resultados são provenientes do *model-checker* SELT e são aplicáveis aos sistemas TTS, como detalhado na seção 2.2. Estes resultados podem ser compreendidos como o valor verdade das propriedades verificadas e os contraexemplos gerados para aquelas tidas como falsas.

Nesta seção, apresentam-se as interfaces desenvolvidas neste trabalho, que possibilitam a visualização, análise e simulação dos contraexemplos. Estas interfaces são inspiradas no trabalho [31], que organiza os resultados da simulação no nível TTS. Dentre estas ferramentas pode ser citada uma janela que apresenta os componentes do modelo AADL; uma janela que contém as propriedades de verificação especificadas com os seus respectivos resultados; uma janela que apresenta os contraexemplos gerados e, por fim, um simulador de contraexemplos, ativado quando se seleciona uma propriedade falsa e que interage com estas janelas mostrando como o sistema se comporta em cada passo de execução do contraexemplo.

Propriedades de Verificação Utilizadas nos Exemplos As propriedades de verificação utilizadas na apresentação das interfaces são as empregadas no modelo do *token-ring* apresentado na seção 3.1.8. Essas propriedades são especificadas com auxílio do assistente de propriedades e utilizadas na verificação deste modelo. Os padrões de propriedade utilizados são apresentados nos itens abaixo:

- Ausência de *deadlock*:

  Deadlock-freeness

- Alcançabilidade da seção crítica:

  station.3.cs is unreachable

- Garantia de resposta:

  station.1.cs responds to net.1.token1 $\bigwedge$ *station.1.waitt*

- Funcionamento contínuo da estação:

  station.1.waitt occurs infinitely often globally

- Exclusão mútua:

  station.1.cs $\bigwedge$ *station.2.cs* $\bigwedge$ *station.3.cs is never reached*

- Garantia que uma estação somente entra em seção crítica depois de receber o token:

  net.1.token1 precedes station.1.cs

- Uma estação não permanece indefinidamente com o *token*:

  net.1.idle2 certainly exists after station.1.cs

A figura 4.4 apresenta as interfaces desenvolvidas com o propósito de visualização dos resultados de verificação, juntamente com o modelo em AADL do protocolo *token-ring*. Na parte superior esquerda da figura pode ser observada a Janela dos Componentes do Modelo AADL, denominada *Process View*; na parte inferior esquerda pode ser observada a Janela dos Contraexemplos, denominada *Trace View* e na parte inferior direita observa-se a Janela das Propriedades Verificadas, denominada *Property View*.

O simulador de contraexemplos pode ser entendido como uma funcionalidade que integra estas três interfaces, sendo ativado no momento que uma propriedade verificada como falsa é selecionada. Nas próximas seções são detalhadas estas interfaces e a maneira em que elas interagem entre si por meio do simulador.

4.3.1 Janela das Propriedades Verificadas

A janela denominada *Property View*, apresenta o significado de todas as propriedades especificadas no nível AADL, relacionando a cada uma delas o resultado da sua verificação. Em outras palavras, indica se a propriedade é falsa ou verdadeira. Esta janela pode ser observada no canto inferior direito da figura 4.4.

A construção desta janela é feita a partir da leitura de dois arquivos: o dos padrões de propriedades exportados e também o dos resultados da verificação. Então é feita a correspondência entre esses arquivos e, para cada propriedade verdadeira, aparece um círculo

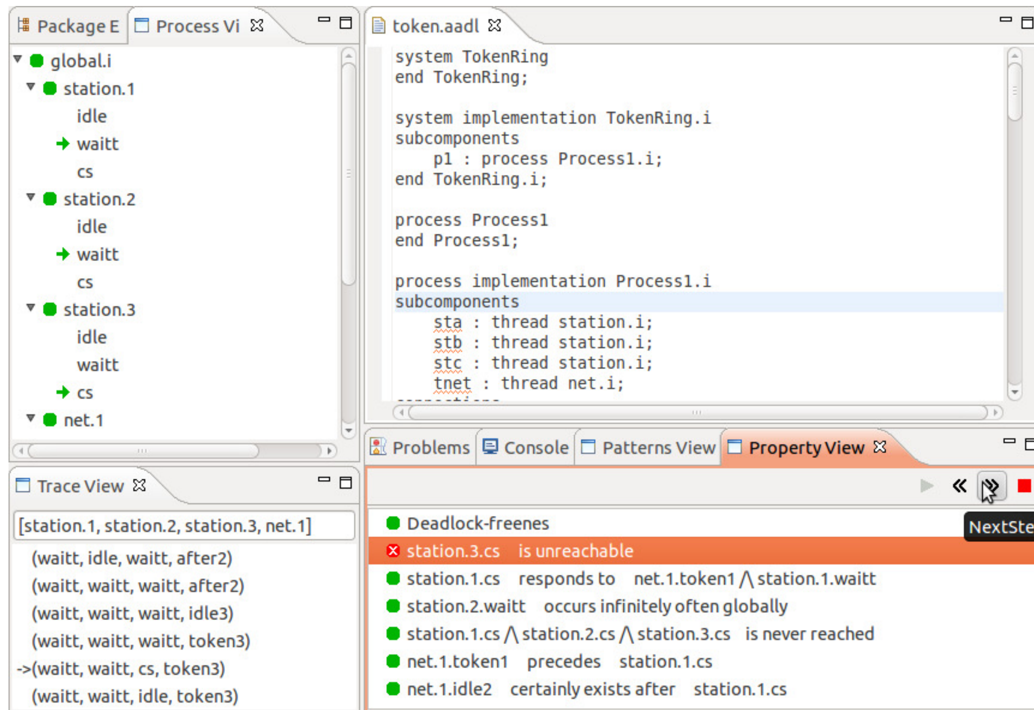

Figura 4.4: Visualização dos resultados de verificação.

verde preenchido e, para cada falsa, um círculo vermelho com uma letra 'x' indicando o seu resultado.

É apresentada nesta *view* somente o significado de cada propriedade, isto é, a informação que aparece no comentário do arquivo das propriedades, como apresentado na seção 4.2. Pode-se notar no canto direito superior desta janela os botões do simulador de contraexemplos. Ao selecionar uma propriedade falsa, o botão (*play*) é habilitado, permitindo ao usuário das início à simulação.

Abaixo são apresentadas as propriedades utilizadas na verificação do modelo do *token-ring* e que são apresentadas na figura 4.4. A cada propriedade é relacionado o seu resultado de verificação, mostrando se é verdadeiro ou falso.

Deadlock-freeness	(Verdadeiro)
station.3.cs is unreachable	(Falso)
station.1.cs responds to net.1.token1 $\bigwedge$ *station.1.waitt*	(Verdadeiro)
station.1.waitt occurs infinitely often globally	(Verdadeiro)
station.1.cs $\bigwedge$ *station.2.cs* $\bigwedge$ *station.3.cs is never reached*	(Verdadeiro)
net.1.token1 precedes station.1.cs	(Verdadeiro)
net.1.idle2 certainly exists after station.1.cs	(Verdadeiro)

4.3.2 Janela dos Componentes do Modelo AADL

A janela denominada *Process View* apresenta os componentes do modelo AADL em uma forma arborescente com seus respectivos estados referentes ao comportamento do sistema. Na parte superior esquerda da figura 4.4 esta janela pode ser observada. Para a sua construção, os lugares do sistema TTS são lidos e passam por uma filtragem que seleciona somente aqueles correspondentes aos estados dos componentes do modelo AADL, desconsiderando os estados desmembrados provenientes da transformação de modelos (conforme detalhado na seção 3.3.2).

Na etapa seguinte, estes lugares TTS são traduzidos para sua respectiva sintaxe em AADL, fazendo as mesmas correspondências apresentadas na seção sobre o tradutor de propriedades 4.2.1. Com isso, os componentes são apresentados todos no mesmo nível hierárquico, não levando em conta a hierarquia do modelo AADL. Para exemplificar, é escolhida a estação 1 que aparece no topo da janela *Process View* da figura 4.4. Os lugares do sistema TTS que são correspondentes ao sistema AADL são os seguintes:

```
station_1_sidle
station_1_swaitt
station_1_scs
```

A filtragem separa o nome da *thread* do nome dos estados e, assim, os nomes que aparecem na *view* são inseridos com a mesma sintaxe de inserção de estados no assistente de propriedades. Desta forma, esta *view* pode ser observada no momento da especificação das propriedades de verificação para a consulta dos nomes dos estados. Na *Process View* estes mesmos lugares TTS do exemplo aparecem como segue:

```
station.1
    idle
    waitt
    cs
```

Como ilustrado na figura 4.4, no decorrer da simulação os estados ativos de cada componente AADL, que correspondem aos estados do contraexemplo, são indicados por uma seta.

4.3.3 Janela dos Contraexemplos

A janela denominada *Trace View*, permite visualizar com clareza o caminho de execução dos contraexemplos. Na figura 4.4, esta *view* pode ser observada na parte inferior esquerda.

Os nomes dos componentes AADL são apresentados na linha superior (entre colchetes) e, nas linhas logo abaixo, são apresentados os estados de cada um desses componentes (entre parênteses), conforme a execução do contraexemplo. Esta *view* é ativada no momento em que o simulador é ativado e tem a função de apresentar todos os passos dos contraexemplos relacionados às propriedades verificadas como falsas. Como pode ser observado na figura 4.4, uma seta aponta qual é o conjunto de estados ativos de cada componente, percorrendo os estados do contraexemplo conforme a evolução da simulação.

Para a sua construção, é lido o arquivo com o contraexemplo proveniente do *model-checker* SELT, onde são buscados os nomes dos lugares TTS. Os estados desmembrados, que não pertencem ao modelo AADL, são ignorados e uma evolução para um próximo passo de execução no contraexemplo em AADL somente ocorre quando um estado do componente AADL é atingido.

Um contraexemplo simplificado para uma propriedade do modelo do *token-ring* é apresentado abaixo. Neste exemplo são mostrados somente três passos de execução e desconsideradas demais informações contidas no contraexemplo. Nota-se que o contraexemplo contém os nomes do sistema TTS, que incluem os estados desmembrados (conforme detalhado na seção 4.2.1) e os estados do elemento Fiacre GLUE, que é criado na transformação de modelos (conforme explicado na seção 3.3).

```
F__GLUE(...) net_1_sidle1 station_1_sidle station_2_sidle station_3_sidle(...)
F__GLUE(...) net_1_sidle1 station_1_sst__54087 station_2_sidle station_3_sidle(...)
F__GLUE(...) net_1_sidle1 station_1_sst__54087 station_2_sst__54087 station_3_sidle(...)
```

Cada um dos passos de execução passa por uma filtragem que exclui o componente GLUE e os estados que são desmembrados no processo de transformação. Neste processo os estados são traduzidos para a sintaxe de inserção dos estados dos componentes do modelo AADL e são ordenados de forma a serem apresentados na *view*. Os componentes AADL e seus respectivos estados são ordenados conforme a *Process View*. Abaixo pode-se observar como é a apresentação de estados na *Trace View*, onde somente o primeiro passo de execução corresponde ao contraexemplo em TTS apresentado no exemplo anterior.

```
[station.1, station.2, station.3, net.1]
(idle, idle, idle, idle1)
(idle, idle, waitt, idle1)
(idle, idle, waitt, token1)
```

Neste exemplo nota-se que são percorridos três passos de execução no contraexemplo em TTS, que correspondem somente ao primeiro passo no contraexemplo em AADL. Isto ocorre desta forma, pois no exemplo mostrado não houve uma transição para um estado de um componente AADL, somente para estados desmembrados.

4.3.4 Simulador de Contraexemplos

O simulador pode ser entendido como uma funcionalidade que integra as três janelas detalhadas anteriormente. A figura 4.4 exemplifica o funcionamento do simulador no modelo do *token-ring*. Quando uma propriedade que é verificada como falsa é selecionada na *Property View*, o simulador pode ser ativado. A exemplo do que ocorre na figura, quando o botão para o próximo passo de simulação é selecionado (*NextStep*) o caminho de execução do contraexemplo é percorrido. A cada passo de execução são indicados na *Trace View* os estados de todos os componentes AADL no momento e os mesmos estados são apontados na *Process View*, que facilita a visão do sistema. O simulador permite que se busquem com facilidade erros de modelagem, ou mesmo inconsistências no sistema modelado.

Outros dois botões existentes são o *PreviousStep* e *StopSimulation*, cujas funcionalidades são respectivamente voltar um passo de simulação e interromper a simulação voltando ao início do contraexemplo.

Os resultados de verificação são apresentados na figura 4.4, onde pode-se observar que a única propriedade falsa é a de não alcançabilidade. A figura exemplifica o simulador de contraexemplos em funcionamento; na *process view* são indicados os estados de cada componente em um dado passo de execução e todos os estados do contraexemplo são apresentados na *trace view*, facilitando acompanhar a execução.

Ao analisar esta propriedade com o auxílio do simulador, chega-se a conclusão que o seu resultado é satisfatório. A intenção de se utilizar esta propriedade de não alcançabilidade é verificar se o estado *station.3.cs* é atingível. Como seu resultado é falso, significa que é falso que este estado nunca seja atingido, ou seja, que *station.3.cs* é alcançável. Acompanhando os passos de simulação pode-se observar um contraexemplo onde a estação 3 atinge a seção crítica em resposta a uma requisição.

4.4 Conclusão

Neste capítulo foram apresentadas as ferramentas desenvolvidas para a especificação de propriedades e para a análise de resultados de verificação na linguagem AADL. A integração destas ferramentas com a cadeia de verificação do projeto *Topcased* foi apresentada, consistindo na transformação das propriedades especificadas, segundo o padrão de propriedades, para a lógica LTL e a apresentação dos resultados à nível AADL, com módulos que permitem a simulação e análise destes resultados.

O sistema de padrões de propriedades de verificação adotado mostrou-se suficientemente capaz de especificar as propriedades para os sistemas testados. A aplicação das ferramentas

no modelo do *token-ring*, permitiu que se exemplificasse de maneira integral as facilidades proporcionadas por essas ferramentas e a sua eficácia na verificação de modelos.

Capítulo 5

Estudo de Caso - Modelagem e Verificação de um Marcapasso Usando AADL

Este capítulo apresenta um estudo de caso onde é realizado, utilizando as ferramentas desenvolvidas neste trabalho, todo o processo de especificação das propriedades, verificação formal e análise dos resultados de verificação do modelo AADL de um sistema de marcapasso cardíaco, cuja especificação pode ser encontrada no documento [38].

Este sistema é modelado baseando-se na mesma especificação utilizada no trabalho [24] e no trabalho [27] discutido na seção 3.4, com o objetivo de possibilitar a comparação entre as ferramentas desenvolvidas nesta dissertação e outras propostas com o mesmo foco, que é apresentada ao final do capítulo.

O marcapasso é um sistema indicado a pacientes que apresentam insuficiência de batimentos cardíacos, que pode ser constante ou por curtos períodos. Consiste de um aparelho eletrônico dividido em um gerador de estímulos elétricos e um fio condutor que leva estes estímulos ao coração.

O coração humano possui quatro cavidades responsáveis pela circulação sanguínea no organismo, dois átrios e dois ventrículos. Os átrios recolhem o sangue das veias, enquanto que os ventrículos lançam este sangue nas artérias. Pode ser dividido em uma metade esquerda, onde circula o sangue arterial (enriquecido de oxigênio), e uma metade direita, onde circula o sangue venoso. A taxa normal de batimentos varia de 60 a 100 batimentos por minuto.

Os sistemas de marcapasso podem ser configurados em diversos modos, que seguem uma classificação segundo um sistema de três letras. A primeira letra se refere à cavidade que recebe o estímulo elétrico, a segunda se refere à cavidade que será monitorada, a terceira

se refere ao tipo de resposta aos sinais monitorados. O modo analisado neste capítulo é o VVI, no qual um ventrículo é monitorado (primeiro V), o que promove a inibição de estímulos (I) neste ventrículo (segundo V).

Neste modo de operação, o marcapasso deve monitorar os batimentos cardíacos (*sense*) em um ventrículo e, conforme estes se encontrem em uma taxa abaixo de um certo valor estipulado (*Lower Rate Limit* - LRL), o sistema envia um sinal elétrico (*pace*) ao ventrículo, estimulando a contração do coração. Do contrário, se os batimentos ocorrerem normalmente na taxa LRL, o sistema não deve enviar estímulos. Além disso, depois de uma contração ventricular, seja ela natural ou estimulada, há um período chamado de *Ventricular Refactor Period* (VRP), onde a monitoração é desabilitada, para evitar que haja uma estimulação causada por alguma medição inesperada.

O sistema possui um *clock* interno que é zerado a cada estímulo *sense* ou *pace*. A partir deste ponto, aguarda um intervalo RI (*Rate Interval*) para enviar um novo estímulo. Sempre que ocorrer um estímulo *pace*, o valor de RI deve corresponder a um período LRI (*Lower Rate Interval*). Mas sempre que ocorre um evento natural *sense*, o valor de RI passa a corresponder a um período maior HRI (*High Rate Interval*). Esta variação do período RI, de LRI para HRI, ocorre quando um evento natural é detectado e é chamada de modo de histerese.

Considerando a taxa de 60 batimentos por minuto para LRL, o valor para LRI é de 1000 ms e para HRI é considerado 1200 ms. O valor do período VRP é considerado 320 ms, e sua contagem é iniciada juntamente com RI, ou seja, VRP está contida em RI.

As propriedades de verificação desejadas para o correto funcionamento deste sistema são: (i) ausência de bloqueio; (ii) o intervalo RI deve valer LRI quando o modo de histerese estiver desabilitado; (iii) o intervalo RI deve valer HRI quando o modo de histerese estiver habilitado e (iv) o sistema não deve monitorar os batimentos durante o período VRP.

5.1 Modelagem em AADL

O modelo AADL é composto por um processo que representa o marcapasso e um *device* que representa o coração. Além disso, são utilizados dois observadores, representados por *devices* que servem para capturar o valor de duas das variáveis visando a verificação. Os componentes do sistema em sua forma gráfica, são apresentados na figura 5.1. Este modelo gráfico é, então, convertido para sua forma textual, que deve ser editada para inserir o comportamento por meio do anexo comportamental. O modelo AADL completo do sistema é apresentado no apêndice A.

O modelo do coração é apresentado na listagem 5.1. O comportamento deste componente conta com dois estados: *ready* é o estado inicial e *wd* representa uma contração. Um

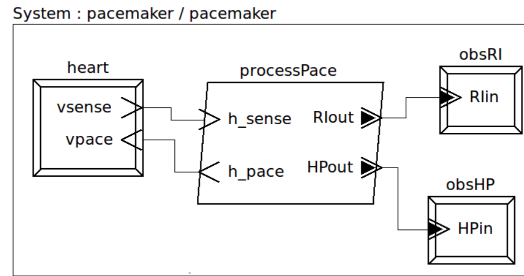

Figura 5.1: Modelagem do sistema do marcapasso em AADL.

evento natural do ventrículo é sinalizado por meio do envio do sinal *vsense* e o recebimento de um estímulo é sinalizado pelo recebimento do sinal *vpace*. A transição de *ready* para *wd* ocorre em razão do estouro de um *timeout*. A transição de *wd* para *ready* ocorre com o recebimento de um estímulo, ou com o envio de um sinal *sense*.

```
device implementation heart.i
        annex behavior_specification {**
        states
        ready: initial complete state;
        wd: state;
        transitions
        ready -[on timeout 480ms]-> wd{delay(480ms,680ms);};
        wd -[]-> ready {vsense!;};
        ready -[vpace?]-> ready;
        wd -[vpace?]-> ready;
        **};
end heart.i;
```

Listagem 5.1: Modelo AADL do coração

O modelo da *thread* que controla o marcapasso é apresentado na listagem 5.2. Esta *thread* conta com quatro estados, sendo que os principais são *wlri* e *whri*. O estado *wlri* representa o envio de estímulos *pace* a uma taxa LRL de 60 bpm, com um período LRI igual a 1000 ms. Contudo, faz uso de um estado auxiliar, denominado *wvrp1*, que representa a espera pelo período VRP igual a 320ms sem monitorar os batimentos, depois de ter enviado um estímulo. Assim, para que se complete o período de 1000 ms entre cada estímulo, a transição de *wlri* para *wvrp1* ocorre com o estouro de um *timeout* de 680 ms.

O estado *whri* representa a ocorrência de eventos naturais *sense* no coração em um período HRI de 1200 ms, com o modo de histerese habilitado. Também faz uso de um estado auxiliar, denominado *wvrp2*, que representa a espera pelo período VRP igual a 320ms sem monitorar os batimentos, depois de ser detectado um evento cardíaco. Similarmente ao que ocorre com *wlri*, para que se complete o período de 1200 ms entre cada evento, a transição de *whri* para *wvrp2* ocorre com o estouro de um *timeout* de 880 ms.

Em quaisquer dos estados *wlri* e *whri*, quando uma contração voluntária não é detectada, o sistema envia o sinal *pace* e passa para o estado *wvrp1*, desabilitando o modo de histerese (*hp := false*). Da mesma forma, em quaisquer dos estados *wlri* e *whri*, quando ocorre um evento natural *sense* o sistema passa para o estado *wvrp2*, habilitando o modo de histerese (*hp := true*).

```
thread implementation pace.i
 subcomponents
 hp: data behavior::boolean;
 hpenable: data behavior::boolean;
 started: data behavior::boolean;
 RI: data behavior::integer;

 annex behavior_specification {**
  states
  wlri: initial complete state;
  whri: complete state;
  wvrp1: state;
  wvrp2: state;
  transitions
  wlri -[on timeout 680ms]-> wvrp1 {e_pace!; hp:=false;};
  wvrp1 -[on timeout 320ms]-> wlri
        {hpenable := hp; started := true; RI :=1000;
        t_d_out!(RI); t_b_out!(hp);};
  wlri -[e_vsense?]-> wvrp2 {hp := true;};
  wvrp2 -[on timeout 320ms]-> whri
        {hpenable := hp; started := true; RI :=1200;
        t_d_out!(RI); t_b_out!(hp);};
  whri -[e_vsense?]-> wvrp2 {hp := true;};
  whri -[on timeout 880ms]-> wvrp1 {e_pace!; hp:=false;};
 **};
end pace.i;
```

Listagem 5.2: Modelo AADL do marcapasso

Os dois observadores são dispositivos externos passivos e servem para indicar o valor das variáveis RI e *hp* da *thread* do marcapasso. Desta forma, quando RI tem seu valor setado para 1000, o observador *obsRI* tem o estado *lri* ativado e quando RI tem seu valor setado para 1200, o observador *obsRI* tem o estado *hri* ativado. No caso do observador obsHP, quando o modo de histerese é ativado (*hp = true*), o seu estado de comportamento *hpe* é ativado, mas quando o modo de histerese é desativado (*hp = false*), o seu estado de comportamento *hpd* é ativado.

5.2 Verificação Formal do Modelo

Para assegurar o correto funcionamento do sistema, devem ser verificadas as suas propriedades. Em primeiro lugar, deseja-se que (i) o sistema não possua situação de *deadlock*, em seguida, deseja-se que (ii) o valor do período RI seja LRI quando o modo de histerese estiver desabilitado e (iii) seja HRI quando o modo de histerese estiver habilitado e, por fim, deseja-se que (iv) durante o período VRP o sistema não monitore os batimentos. Nos itens a seguir são listadas as propriedades desejadas para o sistema de marcapasso e sua respectiva especificação utilizando-se os padrões de propriedades.

- (i) Ausência de *deadlock*:

Deadlock-freeness

- (ii) RI deve valer LRI com o modo de histerese desabilitado:

pace.1.wlri $\bigwedge$ *obsRI.1.lri certainly exists globally*

  Ao atingir *pace.1.wlri*, o sistema desabilita o modo de histerese. O estado *obsRI.1.lri* do observador indica que RI é igual a LRI.

- (iii) RI deve valer HRI com o modo de histerese habilitado:

pace.1.whri $\bigwedge$ *obsRI.1.hri certainly exists globally*

  Ao atingir *pace.1.whri*, o sistema habilita o modo de histerese. O estado *obsRI.1.hri* do observador indica que RI é igual a HRI.

- (iv) o sistema não deve monitorar os batimentos durante o período VRP:

pace.1.wvrp1 precedes pace.1.wlri after pace.1.wvrp1 *pace.1.wvrp2 precedes pace.1.whri*

  Assegura que o sistema somente passe a monitorar os batimentos (*pace.1.wlri* e *pace.1.whri*) depois de esperar o período VRP. Em razão de *pace.1.wlri* ser o estado inicial do sistema, a propriedade só pode ser válida depois do primeiro ciclo.

A especificação destas propriedades é feita utilizando-se o assistente de propriedades apresentado no capítulo anterior. Na figura 5.2, pode ser observada a especificação da propriedade (iv).

Os resultados da verificação para todas estas propriedades são verdadeiros e são apresentados pela Janela das Propriedades Verificadas, como mostra a figura 5.3. Para a primeira

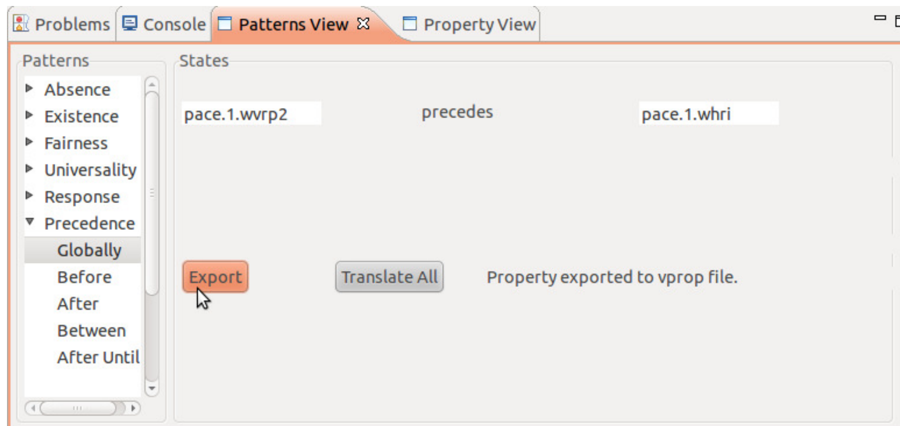

Figura 5.2: Especificação das propriedades de verificação.

propriedade este resultado é satisfatório, indicando que não há situação de *deadlock* no sistema. Para a segunda e terceira propriedade o resultado é satisfatório, indicando que RI tem o seu valor correto dependendo do modo de histerese. Para a quarta propriedade, o resultado também é satisfatório, pois indica que em qualquer situação o sistema aguarda o período VRP antes de monitorar os eventos cardíacos.

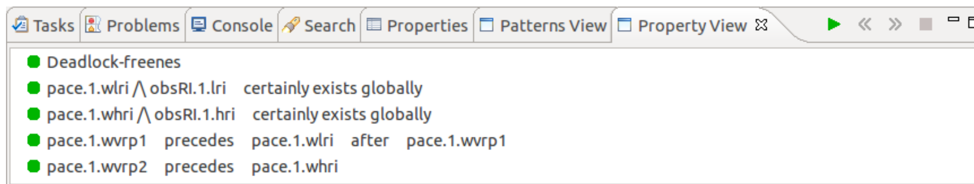

Figura 5.3: Resultados da verificação.

Com isso, pode-se observar que o sistema modelado em AADL está correto, com as propriedades desejadas satisfeitas. Na próxima subseção é exemplificada uma situação onde um erro de modelagem é inserido propositadamente e identificado com o auxílio do simulador.

5.2.1 Análise do Contraexemplo

Esta seção mostra como as ferramentas desenvolvidas podem ser utilizadas para detecção de erros de modelagem. Para isto, um erro é embutido no modelo da *thread* do marcapasso para mostrar a funcionalidade.

A alteração foi feita na transição que sai do estado *wvrp1*. Esta transição representa o final do período VRP, no qual a monitoração dos batimentos não ocorre, e leva ao estado *wlri* que indica que não houve um evento natural do coração, ou seja, um estímulo *pace* foi enviado. No entanto, no modelo com o erro embutido, esta transição leva ao estado *whri*,

que indicaria que um evento natural foi detectado, o que gera inconsistência no modelo. Este trecho do modelo modificado pode ser observado na listagem 5.3.

```
thread implementation pace.i       (...)
 annex behavior_specification {**
  transitions
  wlri -[on timeout 680ms]-> wvrp1 {e_pace!; hp:=false;};
  wvrp1 -[on timeout 320ms]-> whri
        {hpenable := hp; started := true; RI :=1000;
        t_d_out!(RI); t_b_out!(hp);};
(...)
 **};
end pace.i;
```

Listagem 5.3: Erro embutido na *thread pace*

Com este novo modelo, as mesmas propriedades são testadas, entretanto, nesta nova condição, os resultados são diferentes, indicando a existência de um erro no modelo. A propriedade que afirma que com o modo de histerese desabilitado o valor de RI é igual a LRI, que anteriormente tinha seu valor verdadeiro, é verificada como falsa. Além disso, a propriedade que afirma que o estado *wvrp2* sempre precede o estado *whri* também é verificada como falsa.

pace.1.wlri $\bigwedge$ *obsRI.1.lri certainly exists globally*	(Falso)
pace.1.wvrp2 precedes pace.1.whri	(Falso)

Para a primeira destas propriedades, este resultado sinaliza que existe alguma situação em que o modo de histerese está desabilitado (estado *wlri*), mas o valor de RI é indicado como HRI pelo observador. Para a segunda propriedade, o resultado indica que existe uma situação onde o estado *whri* é atingido sem ser precedido pelo estado *wvrp2*.

Na figura 5.4 pode-se observar a Janela das Propriedades Verificadas apresentando este resultado. Esta figura montra a situação em que o simulador é utilizado para esta análise. Como pode-se observar na visão do contraexemplo, é mostrada uma situação em que o estado *whri* é atingido, sendo precedido por *wvrp1*.

Com isso, constata-se que a transição que apresenta um problema é a que sai do estado *wvrp1*. Na janela *Process View*, pode-se observar que o estado *whri* está ativo na *thread pace*, enquanto que o estado *lri* está ativo na *thread* obsRI. Desta forma, ao observar no modelo a transição que parte do estado *wvrp1*, nota-se que ela leva ao estado *whri*, indicando o erro de modelagem.

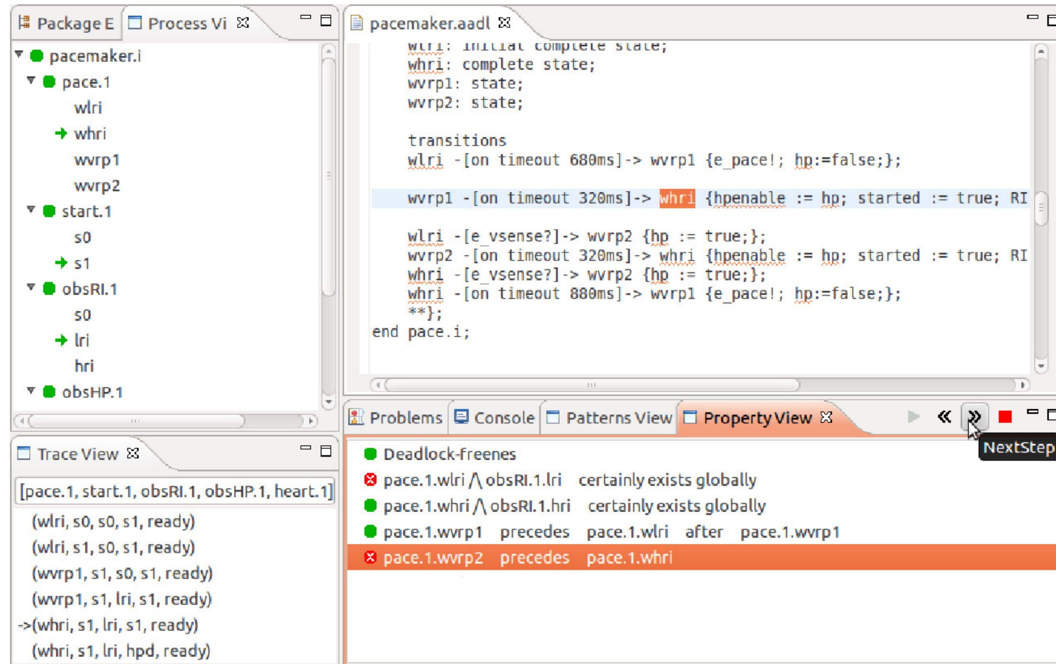

Figura 5.4: Simulador utilizado para detectar erro de modelagem.

5.3 Comparação com Outras Propostas

As seções anteriores apresentaram a utilização das ferramentas desenvolvidas neste trabalho para modelar e verificar um sistema de marcapasso. Este mesmo sistema é utilizado como estudo de caso da linguagem BLESS [27], discutida na seção 3.4, e também do trabalho em [24], o qual faz a verificação usando a ferramenta Uppaal [2]. Esta seção tem o objetivo de fazer uma comparação entre estas três propostas.

Especificação das propriedades de verificação Observa-se que a especificação de propriedades de verificação é facilitada com o assistente construído, pois os padrões de propriedades empregados são apresentados na forma de uma linguagem natural ao usuário. O conjunto de todas as propriedades especificadas neste estudo de caso é apresentado na listagem 5.4. Para fins de comparação com as outras abordagens destaca-se a última propriedade, a qual afirma que sempre antes do estado *whri* ser atingido, ocorre um período VRP, que é o significado do estado *wvrp2*.

```
Deadlock−freeness
pace.1.wlri /\ textbackslash obsRI.1.lri certainly exists globally
pace.1.whri /\ textbackslash obsRI.1.hri certainly exists globally
pace.1.wvrp1 precedes pace.1.wlri after pace.1.wvrp1
pace.1.wvrp2 precedes pace.1.whri
```

Listagem 5.4: Propriedades utilizadas no estudo de caso do marcapasso.

No caso da linguagem BLESS a especificação dos requisitos do sistema deve ser feita no momento de construção do modelo, utilizando a sintaxe particular da sua sublinguagem *Assertion*. Nota-se que, BLESS não permite a especificação de propriedades que utilizem sequências de estados, pois não é permitido o uso de operadores temporais como os empregados na lógica temporal. Observa-se também, como detalhado na seção 3.4, que a linguagem BLESS não permite a verificação de componentes paralelos concorrentes, somente de componentes individuais. Um exemplo de definição de uma propriedade do sistema por meio de um *Assertion* pode ser visto na listagem 5.5. Esta propriedade tem o mesmo objetivo da propriedade de precedência citada no parágrafo anterior, que é assegurar que durante o período correspondente a VRP os batimentos não sejam monitorados no modelo do marcapasso. Esta propriedade afirma que os eventos *vp* ou *nr_vs* sempre ocorrem fora do intervalo VRP. Observa-se pela listagem 5.5 que BLESS especifica esta propriedade sem declarar a precedência de estados, mas para isto utiliza o conceito de intervalos de tempo. Além disso, a maneira como a especificação da propriedade é feita em BLESS é bem menos intuitiva do que na abordagem proposta.

```
<<notVRP: : (vp or nr_vs)@last_vp_or_vs and
  (now−last_vp_or_vs)>=pp.ventricular_refractory_period>>
```

Listagem 5.5: Propriedade sobre o período VRP definida em um *Assertion* em BLESS.

Além disso, nota-se com o estudo de caso que a utilização de observadores não é permitida com a linguagem BLESS. Essa linguagem não permite a verificação de componentes paralelos concorrentes. As propriedades utilizadas no estudo de caso da linguagem BLESS, declaradas na forma de *Assertions*, são apresentadas na listagem 5.6. Estas propriedades têm os mesmos objetivos das apresentadas na listagem 5.4, e permitem observar a maior dificuldade de compreensão das mesmas.

Considerando a ferramenta Uppaal, a especificação das propriedades é feita diretamente por meio da lógica CTL. A mesma propriedade sobre o período VRP é especificada como a última apresentada na listagem 5.7. Esta propriedade afirma que qualquer instante de tempo em que o sistema estiver nos estados *Ventricle.WaitRI* e *Ventricle.start* simultaneamente é maior do que o período VRP. Por outro lado, esta ferramenta não trabalha com a linguagem AADL, mas com autômatos temporizados, que estão em um nível de abstração mais baixo, podendo ser comparável a ferramenta SELT. O estudo de caso apresentado em [24], que faz a verificação do sistema do marcapasso em Uppaal, verifica as mesmas propriedades utilizadas neste capítulo, apresentadas na listagem 5.7.

```
<<LRL:theTime: exists t:Timing_Properties::Time
  in (theTime-pp.lower_rate_limit_interval)..theTime
  that (nr_vs@t or vp@t) >>

<<URL:theTime: vp@theTime
  => not (exists t:Timing_Properties::Time
      in (theTime-pp.lower_rate_limit_interval),,theTime
      that nr_vs@t or vp@t)>>

<<notVRP: : (vp or nr_vs)@last_vp_or_vs and
    (now-last_vp_or_vs)>=pp.ventricular_refractory_period>>

<<VS: : vs@now and notVRP() >>

<<VP: : stat_pace@now or
  ((vp or nr_vs)@(now-pp.lower_rate_limit_interval)
  and
    not (exists t:Timing_Properties::Time
      in (now-pp.lower_rate_limit_interval),,now
      that (nr_vs or vp)@t)) >>

<<PACE:theTime:vp@last_vp_or_vs and
  (exists t:Timing_Properties::Time
    in (theTime-pp.lower_rate_limit_interval)..theTime
    that vp@t) >>

<<SENSE:theTime:nr_vs@last_vp_or_vs and
  (exists t:Timing_Properties::Time
    in (theTime-pp.lower_rate_limit_interval)..theTime
    that nr_vs@t) >>

<<LAST: :(vp or nr_vs)@last_vp_or_vs>>
```

Listagem 5.6: Propriedades utilizadas em BLESS.

```
A[](not deadlock)
A[][](!Ventricle:hpenable imply Ventricle:x <= Ventricle:LRI)
A[](Ventricle:hpenable imply Ventricle:x <= Ventricle:HRI)
A[]((Ventricle:WaitRI && Ventricle:started) imply Ventricle:x >=
    Ventricle:VRP)
```

Listagem 5.7: Propriedades utilizadas em [24] com a ferramenta Uppaal.

Visualização dos Resultados de Verificação Por meio do estudo de caso apresentado, pode-se observar como as ferramentas desenvolvidas neste trabalho facilitam a visualização e análise dos resultados de verificação, permitindo a simulação de contraexemplos.

A proposta da linguagem BLESS não permite a simulação dos resultados de verificação. Além disto, não é gerado um contraexemplo, pois o seu processo de verificação somente permite que se observe se o modelo é correto ou não e também qual é a Asserção falsa, no caso de não ser correto.

A ferramenta Uppaal, apesar de trabalhar com uma linguagem de mais baixo nível, permite a visualização e análise dos resultados de verificação, por meio da simulação de contraexemplos.

5.4 Conclusão

Este capítulo apresentou a modelagem em AADL do sistema de um marcapasso, a especificação e verificação das suas propriedades de verificação com o auxílio das ferramentas desenvolvidas nesta dissertação.

A partir deste estudo de caso, pode-se observar as facilidades proporcionadas pelo assistente de propriedades, assim como a eficácia da cadeia de verificação. Além disso, é possível constatar que a abrangência do sistema de padrões de propriedades é suficiente para a verificação da maioria dos sistemas.

Neste capítulo podem ser observadas as facilidades trazidas pelas ferramentas de visualização dos resultados de verificação, permitindo que se encontrem erros de modelagem e inconsistências no sistema, por meio da simulação de contraexemplos.

Por fim, são apresentadas as vantagens e as desvantagens das ferramentas desenvolvidas em relação a outras propostas, como a linguagem BLESS e a ferramenta Uppaal. Estas vantagens são correspondentes à facilidade para especificação de propriedades de verificação em alto nível e a possibilidade de simulação de contraexemplos de verificação no nível da linguagem AADL.

Capítulo 6

Conclusões

Esta dissertação apresentou a cadeia de verificação formal de propriedades utilizada no projeto Topcased, particularmente a sua relação com a linguagem de descrição de arquitetura AADL. O projeto Topcased tem como objetivo a criação de métodos e ferramentas para o desenvolvimento de sistemas embarcados críticos, sendo um dos seus focos a verificação formal de propriedades, que assegura que o sistema está sendo construído corretamente, antes da fase de implementação e testes.

A primeira parte do trabalho desenvolvido constitui na elaboração de um assistente para especificação de propriedades de verificação em modelos de alto nível, cuja aplicação é voltada para a linguagem AADL. Esse assistente se baseia em um sistema de padrões de propriedades, o qual é redefinido nesta dissertação. Buscou-se fazer uma simplificação, adotando apenas as propriedades cruciais e as mais utilizadas na verificação, com o objetivo de facilitar a especificação de propriedades, abstraindo o formalismo da lógica temporal utilizado pelas ferramentas de verificação.

Em um segundo momento, este trabalho integrou o assistente de propriedades na cadeia de verificação do Topcased, desenvolvendo também ferramentas para análise dos resultados de verificação e simulação de contraexemplos. Estas ferramentas se mostraram importantes para correta compreensão do modelo e correção de possíveis erros de modelagem.

A aplicação das ferramentas desenvolvidas nesta dissertação é validada por meio de um estudo de caso, onde é desenvolvido o modelo de um sistema de marcapasso, utilizando a linguagem AADL. Este estudo de caso permitiu que se fizessem comparações entre as ferramentas desenvolvidas e os trabalhos relacionados, demonstrando o potencial deste trabalho no sentido da especificação das propriedades e simulação de contraexemplos.

Como limitação deste trabalho, pode-se citar que, apesar dos padrões de propriedades se mostrarem abrangentes, não são completos e não varrem todas as possibilidades. Isso obriga o utilizador a realizar a definição manual daquelas propriedades que não se encaixam em um

destes padrões. Outra limitação da cadeia de verificação é que não é possível a verificação de componentes individuais do modelo, sendo que o ambiente deve ser modelado juntamente.

A respeito da propriedade de universalidade, conclui-se que esta não deve ser utilizada com a cadeia de verificação atual, pois o seu resultado não é correto com estas ferramentas da forma com que estão implementadas. Para sua utilização, deve haver uma maior interação entre as ferramentas de transformação de modelos, como destacado nas perspectivas deste trabalho.

As perspectivas futuras deste trabalho vão na direção de encontrar uma maneira de identificar os estados desmembrados no momento da transformação de modelos, facilitando a verificação de alguns dos padrões de propriedades, como o de universalidade. Outro ponto desejável é a melhoria da apresentação da estrutura do modelo AADL, representando os componentes com a mesma estrutura hierárquica com que foram declarados no modelo.

Apêndice A

Apêndice – Código AADL do Modelo do Marcapasso

```
system pacemaker
end pacemaker;

system implementation pacemaker.i
        subcomponents
        processPace: process pcmkr.i;
        oRI: device obsRI.i;
        oHP: device obsHP.i;
        dheart: device heart.i;
        connections
        event data port processPace.p_d_out -> oRI.RIin;
        event data port processPace.p_d1_out -> oHP.HPin;
        event port dheart.vsense -> processPace.h_sense;
        event port processPace.h_pace -> dheart.vpace;
end pacemaker.i;

process pcmkr
        features
        p_d_out: out event data port behavior::integer;
        p_d1_out: out event data port behavior::boolean;
        h_pace: out event port;
        h_sense: in event port;
end pcmkr;

process implementation pcmkr.i
        subcomponents
        tpace: thread pace.i;
        connections
        event data port tpace.t_d_out -> p_d_out;
        event data port tpace.t_b_out -> p_d1_out;
        event port h_sense -> tpace.e_vsense;
```

```
        event port tpace.e_pace -> h_pace;
end pcmkr.i;

thread pace
        features
        e_pace: out event port;
        e_vsense: in event port;
        t_d_out: out event data port behavior::integer;
        t_b_out: out event data port behavior::boolean;
        properties
        Dispatch_Protocol => Aperiodic;
end pace;

thread implementation pace.i
        subcomponents
        hp: data behavior::boolean;
        hpenable: data behavior::boolean;
        started: data behavior::boolean;
        RI: data behavior::integer;

        annex behavior_specification {**
        states
        wlri: initial complete state;
        whri: complete state;
        wvrp1: state;
        wvrp2: state;
        transitions
        wlri -[on timeout 680ms]-> wvrp1 {e_pace!; hp:=false;};
        wvrp1 -[on timeout 320ms]-> wlri
                {hpenable := hp; started := true; RI :=680;
                t_d_out!(RI); t_b_out!(hp);};
        wlri -[e_vsense?]-> wvrp2 {hp := true;};
        wvrp2 -[on timeout 320ms]-> whri
                {hpenable := hp; started := true; RI :=880;
                t_d_out!(RI); t_b_out!(hp);};
        whri -[e_vsense?]-> wvrp2 {hp := true;};
        whri -[on timeout 880ms]-> wvrp1 {e_pace!; hp:=false;};
        **};
end pace.i;

device heart
        features
        vsense: out event port;
        vpace: in event port;
        properties
        Device_Dispatch_Protocol => Aperiodic;
end heart;

device implementation heart.i
```

```
        annex behavior_specification {**
        states
        ready: initial complete state;
        wd: state;

        transitions
        ready -[on timeout 680ms]-> wd{delay(680ms,880ms);};
        wd -[]-> ready {vsense!;};
        ready -[vpace?]-> ready;
        wd -[vpace?]-> ready;
        **};
end heart.i;

device obsRI
        features
        RIin: in event data port behavior::integer;
        properties
        Device_Dispatch_Protocol => Aperiodic;
end obsRI;

device implementation obsRI.i
        annex behavior_specification {**
        states
        s0: initial state;
        lri: complete state;
        hri: complete state;
        transitions
        s0 -[RIin? when RIin = 1200]-> hri;
        s0 -[RIin? when RIin = 1000]-> lri;
        hri -[RIin? when RIin = 1000]-> lri;
        lri -[RIin? when RIin = 1200]-> hri;
        **};
end obsRI.i;

device obsHP
        features
        HPin: in event data port behavior::boolean;
        properties
        Device_Dispatch_Protocol => Aperiodic;
end obsHP;

device implementation obsHP.i
        annex behavior_specification {**
        states
        s1: initial state;
        hpe: complete state;
        hpd: complete state;
        transitions
        s1 -[HPin? when HPin = true]-> hpe;
```

```
        s1 -[HPin? when HPin = false]-> hpd;
        hpe -[HPin? when HPin = false]-> hpd;
        hpd -[HPin? when HPin = true]-> hpe;
        hpd -[HPin? when HPin = false]-> hpd;
        hpe -[HPin? when HPin = true]-> hpe;
        **};
end obsHP.i;

thread start
        features
        st_out: out event port;
        properties
        Dispatch_Protocol => Background;
end start;

thread implementation start.i
        annex behavior_specification {**
        states
        s0: initial complete state;
        s1: complete state;
        transitions
        s0 -[]-> s1 {st_out!;};
        **};
end start.i;
```

Listagem A.1: Código AADL do modelo do marca passo

Anexo A

Anexo – Linguagem BLESS

É dividida em três sublinguagens: *Assertion* define asserções e declara propriedades do sistema sobre o tempo, utilizando lógica de primeira ordem, *subBLESS* serve como anexo comportamental para subprogramas e *BLESS* serve como anexo comportamental para *threads*. Estas três sublinguagens interagem entre si em um modelo AADL, o que permite que o mecanismo de provas possa utilizar as declarações do modelo para provar o seu correto funcionamento. Os componentes AADL onde *BLESS* se aplica são *Subprogram*, *Threads*, *Devices* e *Systems*, além de todas as suas implementações associadas e componentes estendidos.

A.1 *Assertion*

Define fórmulas lógicas com restrições temporais[1], escritas entre aspas angulares (<<>>). Segundo a lógica proposicional, uma asserção é uma sentença declarativa e uma proposição é uma asserção avaliada com um valor verdade (verdadeira ou falsa). Em *BLESS*, um anexo *Assertion* contém um predicado ou outros Assertions. Uma asserção em *BLESS* pode ser entendida como uma fórmula lógica com parâmetros formais, como o tempo, que se relacionam com *features*, como portas. Cada *feature* pode estar relacionada a um *Assertion* servindo, nesse caso, como um rótulo (*label*) que será relacionado em uma fórmula lógica. Permite o uso de operadores lógicos (conjunção, disjunção, implicação), mas não permite o uso de operadores temporais (próximo, no futuro) como os utilizados na lógica temporal.

Esta sublinguagem é que define as propriedades que o sistema deve cumprir sobre o tempo. O trecho de código abaixo (Listagem A.1) foi retirado de um modelo AADL referente ao controle de um marcapasso que é apresentado no apêndice B, ele exemplifica um anexo *Assertion* servindo como rótulo para as portas *vp* e *nr_vs*.

[1] O autor define um *Assertion* como uma fórmula lógica temporal. Entretanto os operadores temporais como **G** (globalmente), **F** (futuro), **X** (próximo) não são permitidos em *BLESS*, o que descaracteriza a lógica temporal.

```
vs: in event port;
vp: out event port
        {BLESS::Assertion=>"<<VP()>>"};
nr_vs: out event port
        {BLESS::Assertion=>"<<VS()>>"};
```

Listagem A.1: exemplo de rótulos em *features*

No exemplo do trecho de código abaixo (Listagem A.2), também retirado do modelo do marcapasso no apêndice B, pode-se observar três asserções definidas em um anexo AADL. A primeira, LRL, define um intervalo de tempo entendido como *lower_rate_limit*, que serve como o período mínimo para ocorrência de eventos nas portas *vp* ou *nr_vs*, ou seja, esta asserção se torna uma proposição verdadeira sempre que ocorrer um evento em alguma destas portas dentro deste intervalo de tempo. A asserção notVRP é verdadeira quando ocorrer um evento nas portas *vp* ou *nr_vs* e o instante em que este evento ocorrer for maior do que um dado intervalo de tempo definido como VRP.

```
<<LRL:theTime: exists t:Timing_Properties::Time
  in (theTime-pp.lower_rate_limit_interval)..theTime
  that (nr_vs@t or vp@t) >>

<<notVRP: : (vp or nr_vs)@last_vp_or_vs and
  (now-last_vp_or_vs)>=pp.ventricular_refractory_period>>
```

Listagem A.2: *Assertion* definindo propriedades.

A.2 *SubBLESS*

Esta sublinguagem serve como anexo a subprogramas, com o objetivo de definir o seu comportamento. Juntamente com um anexo *subBLESS* podem ser declarados *Assertions*, e a maneira com que esses elementos interagem é analisada por meio do mecanismo de provas. O anexo *SubBLESS* procura garantir que um dado subprograma garanta as suas especificações. Os seus elementos são:

- *Availability*: É usado com modelos PLE (*Product Line Engeneering*) para controlar quais características um produto possui[2];
- *Assertions*;
- Pré-condição: predicado que precisa ser verdadeiro nos parâmetros do subprograma;

[2] PLE é um processo na produção de novos softwares e o plug-in que estende AADL para suportar PLE é Fault Tree Analysis (FTA).

- Pós-condição: predicado que será verdadeiro após a execução do subprograma;
- *Existential lattice quantification*: define variáveis e comportamento da ação.

O anexo *subBLESS* não permite acesso a portas e tempo.

O trecho de código abaixo (Listagem A.3) exemplifica a utilização do anexo *subBLESS*, onde podem ser observadas as pós-condições, asserções, declaração de variáveis e o comportamento da ação. O exemplo define uma função para fazer a multiplicação de dois números, que depois é usada em outra função para calcular a potência cúbica deste número.

```
data number
end number;

subprogram mul
  features
    x : in parameter number;
    y : in parameter number;
    z : out parameter number;
  annex subBLESS {**
    post z=x*y --postcondition relating result to values of inputs
    { z := x*y <<z=x*y>> }
  **};
end mul;

subprogram cube
  features --cube is also a function
    x : in parameter number;
    y : out parameter number;
    mul : requires subprogram access mul;
  annex subBLESS {**
    post y=x*x*x
    variables --existential quantification introduces local variables
    tmp : number;
    { mul(x,x,tmp) --invoke mul as subprogram
      ; <<tmp=x*x>>
      --sequential composition
      y := mul(tmp,x) <<y=x*x*x>> } --invoke mul as function
  **};
end cube;
```

Listagem A.3: exemplo de anexo *subBLESS*

A.3 *BLESS*

A sublinguagem *BLESS* é aplicada para definir o comportamento de *threads*, podendo também ser usada em *devices* e *systems*. Em um anexo *BLESS* podem ser declarados os

seguintes elementos:

- *Availability*: É usado com modelos PLE (*Product Line Engeneering*) para controlar quais features um produto possui;
- *Assertions*: declaração de fórmulas lógicas, exprimindo as propriedades do sistema;
- Invariante: a proposição declarada pelo invariante deve ser sempre verdadeira, é o que garante que o sistema cumpra com as especificações, segundo o funcionamento do mecanismo de provas;
- Variáveis;
- Sistema de transição de estados.

Pode ser inserida em modelos AADL através de anexos. Se aplicada a um componente, se torna aplicável a todas as implementações associadas. Se um componente é estendido, anexos definidos em um ancestral são aplicados em seus descendentes, exceto quando este último define seu próprio anexo. Um anexo pode ser especificado para um determinado modo. Se não é específico a algum modo, é aplicado a todos os modos. O exemplo apresentado na listagem A.4 mostra um sistema de transição de estados que conta o número de ocorrências de eventos na porta *tick* e envia este valor pela porta *sp*.

```
thread speed
   features
      tick: in event port { Dequeue_Protocol => AllItems;};
      sp: out data port BLESS::integer;
   properties
      Dispatch_Protocol => periodic;
      Period => 1 s;
end speed;

thread implementation speed.i
   annex BLESS {**
      states
      s0: initial complete state;
      s1: final state; --entered by invoking finalize entrypoint
  transitions
      s0 -[ ]-> s0 { sp := tick'count }
   **};
end speed.i
```

Listagem A.4: exemplo de anexo *BLESS*

A.4 Mecanismo de Provas

O mecanismo de prova da linguagem BLESS funciona da seguinte maneira: o objetivo majoritário é provar que o invariante é sempre verdadeiro. A partir disto é construída uma árvore de obrigações de prova, onde o invariante é o primeiro nó a ser inserido, sendo seguido pelos estados completos (*complete states*) do anexo comportamental que devem todos implicar no invariante.

Dessa forma é construída a árvore de provas, onde cada nodo é um *Assertion*, sendo que o primeiro é o invariante, os imediatamente abaixo são os estados completos, os abaixo destes são os estados que implicam nestes e assim por diante segundo o sistema de transição de estados modelado.

Após construída a árvore, o mecanismo de prova se utiliza da lógica proposicional para provar cada uma das obrigações, criando teoremas. Cada um dos nós da árvore é provado até se chegar na prova do invariante, que diz que o sistema cumpre com as especificações.

O trecho de código abaixo (Listagem A.5), referente ao modelo apresentado no apêndice B, apresenta a declaração de um invariante, VVI, e de duas asserções referenciadas a estados completos, PACE e SENSE.

```
<<PACE:theTime:vp@last_vp_or_vs and
   (exists t:Timing_Properties::Time --there is a time
   in (theTime-pp.lower_rate_limit_interval)..theTime
      --in the previous LRL interval ms
   that vp@t) >> --with a ventricular pace
      --a ventricular sense occurred in the previous LRL interval

<<SENSE:theTime:nr_vs@last_vp_or_vs and
   (exists t:Timing_Properties::Time --there is a time
   in (theTime-pp.lower_rate_limit_interval)..theTime
that nr_vs@t) >>

invariant
   --goal is to prove VVI is always true
   --When turned on, there will always be a paced or sensed
   --heartbeat in the previous LRL interval
   <<VVI: : LRL(now) >>
```

Listagem A.5: Declaração de asserções

Na Listagem A.6 observam-se as provas que são geradas a partir deste trecho de código, onde os estados completos devem implicar no invariante.

```
P << >>
S [73]=>
Q [73] <<LRL(now)>>
Reason: Initial Thread Obligations
What for: All initial proof obligations of thread
have correctness proofs.

P [86] <<PACE(now)>>
S [73]=>
Q [73] <<LRL(now)>>
What for: <<M(pace)>> => <<I>> from invariant I when complete state
pace has Assertion <<M(pace)>> in its definition.

P [90] <<SENSE(now)>>
S [73]=>
Q [73] <<LRL(now)>>
What for: <<M(sense)>> => <<I>> from invariant I when complete state
sense has Assertion <<M(sense)>> in its definition.
```

Listagem A.6: Mecanismo de prova cria a árvore de provas a partir das asserções

Na listagem A.7 apresenta-se um exemplo de um teorema criado pela ferramenta de provas, utilizado para provar um dos nós da árvore de prova.

```
Theorem (24)
[serial 1066]
102 {P} <<vp@now and now = last_vp_or_vs>>
101 S =>
86 {Q} <<PACE(now)>>
by Substitution of Assertion Labels
Normalization Axioms:
Add Unnecessary Parentheses For No Good Reason: a = (a)
Reflexivity of Equality: (a=b) = (b=a)
and theorem 23:
Theorem (23) [serial 1152] used for:
substituted Assertions' predicates for labels [serial 1066]
```

Listagem A.7: Exemplo de um teorema gerado para as provas

O suporte à linguagem BLESS é providenciado por meio de uma ferramenta desenvolvida pelo mesmo instituto criador da linguagem. O objetivo desta ferramenta é ser integrada ao plug-in OSATE, que oferece suporte à linguagem AADL. BLESS *Prof Tool* executa o *parsing* do modelo AADL, identificando os seus elementos e os elementos que devem ser usados no mecanismo de prova, e logo após forma a árvore de provas e executa o mecanismo completo.

Anexo B

Anexo – Modelo do Marcapasso em AADL com o anexo de Linguagem BLESS

```
--VVI.aadl
--simple single-chamber pacemaker, VVI mode

system VVI
  features
  stat_pace: in event port;  --immediately start pacing
  vs: in event port;     --a ventricular contraction has been sensed
  vp: out event port     --pace ventricle
    {BLESS::Assertion=>"<<VP()>>"};
  nr_vs: out event port --non-refractory ventricular sense
    {BLESS::Assertion=>"<<VS()>>"};

annex BLESS
{**
assert
      --lower rate limit: there was a pace or non-refractory
      --sense before theTime, within the LRL interval
  <<LRL:theTime: exists t:Timing_Properties::Time
    in (theTime-pp.lower_rate_limit_interval)..theTime
    that (nr_vs@t or vp@t) >>

      --upper rate limit, same as url=lrl for VVI
  <<URL:theTime: vp@theTime
    => not (exists t:Timing_Properties::Time
        in (theTime-pp.lower_rate_limit_interval),,theTime
        that nr_vs@t or vp@t)>>

      --vetricular refractory period:
```

```
    --there was a pace or non-refractory sense
    --before theTime, within VRP
<<notVRP: : (vp or nr_vs)@last_vp_or_vs and
    (now-last_vp_or_vs)>=pp.ventricular_refractory_period>>

    --meaning of VS marker is nr_vs out port event
    --should get from Assertion annexed to AADL port from OSATE
<<VS: : vs@now and notVRP() >>

    --meaning of VP marker is vp out port event
    --for VVI, vp! means (vp or nr_vs) occurred LRL interval ago
    --and not since
<<VP: : --last pace or sense LRL interval ago,or stat pace
  stat_pace@now or
  ((vp or nr_vs)@(now-pp.lower_rate_limit_interval)
  and --there does not exist a time
    not (exists t:Timing_Properties::Time
      --since then, note ",,"
      in (now-pp.lower_rate_limit_interval),,now
      --with a non-refractory ventricular sense or pace
      that (nr_vs or vp)@t)) >>

    --a ventricular pace occurred in the previous LRL interval

<<PACE:theTime:vp@last_vp_or_vs and
  (exists t:Timing_Properties::Time --there is a time
    in (theTime-pp.lower_rate_limit_interval)..theTime
      --in the previous LRL interval ms
    that vp@t) >> --with a ventricular pace

    --a ventricular sense occurred in the previous LRL interval
<<SENSE:theTime:nr_vs@last_vp_or_vs and
  (exists t:Timing_Properties::Time --there is a time
    in (theTime-pp.lower_rate_limit_interval)..theTime
      --in the previous LRL interval ms
    that nr_vs@t) >> --with a non-refractory VS

    --the last VP or non-refractory VS occurred at last_vp_or_vs
<<LAST: :(vp or nr_vs)@last_vp_or_vs>>

invariant
  --goal is to prove VVI is always true
  --When turned on, there will always be a paced or sensed
  --heartbeat in the previous LRL interval
  <<VVI: : LRL(now) >> --LRL is true, whenever "now" is

variables
  last_vp_or_vs : persistent Timing_Properties::Time;
    --time of last ventricular pace or sense
```

```
states
  start : initial state  --powered-up, awaiting implant
    <<notVRP()>>;
  off : final state;  --upon "stop"
  pace : complete state
      --a ventricular pace has occured in the
      --previous LRL-interval milliseconds
    <<PACE(now)>>;
  sense : complete state
      --a ventricular sense has occured in the
      --previous LRL-interval milliseconds
    <<SENSE(now)>>;
  check_pace_vrp : state
      --an execute state to check if vs is in vrp
    <<vs@now and PACE(now)>>;
  check_sense_vrp : state
      --need a different check state for sense
    <<vs@now and SENSE(now)>>;

transitions
  T1_START_BY_STAT_PACE:           --start pacing on STAT PACE
  start -[on dispatch stat_pace]-> pace
    {<<stat_pace@now>>vp!<<vp@now>>   --cause first pace
    &last_vp_or_vs:=now<<last_vp_or_vs=now>>};

  T2_START_BY_VS:          --first event ventricular sense
  start -[on dispatch vs]-> sense
        --record time of most recent event,
        --start timeouts with nr_vs!
    {last_vp_or_vs:=now<<last_vp_or_vs=now>>
    &  <<VS()>>
    nr_vs!<<nr_vs@now>>};

  T3_PACE_LRL_AFTER_VP: --pace when LRL times out
  pace -[on dispatch timeout (vp nr_vs)
        pp.lower_rate_limit_interval ms]-> pace
    { <<VP()>>
    vp!<<vp@now>>
    &last_vp_or_vs:=now<<last_vp_or_vs=now>>};

  T4_VS_AFTER_VP:          --sense after pace=>check if in VRP
  pace -[on dispatch vs]-> check_pace_vrp{};

  T5_VS_AFTER_VP_IN_VRP:  -- vs in VRP,  go back to "pace" state
  check_pace_vrp -[ --a and c and b and x and y and x and z
      (now-last_vp_or_vs)<pp.ventricular_refractory_period
    ]-> pace{};
```

```
  T6_VS_AFTER_VP_IS_NR: --vs after VRP,
    --go to "sense" state, send nr_vs!, reset timeouts
  check_pace_vrp -[(now-last_vp_or_vs)>=
      pp.ventricular_refractory_period]-> sense
    { <<VS()>>
    nr_vs!<<nr_vs@now>> --send nr_vs! to reset timeouts
    &last_vp_or_vs:=now<<last_vp_or_vs=now>>};

  T7_PACE_LRL_AFTER_VS: --pace when LRL times out after VS
  sense -[on dispatch timeout (vp nr_vs)
      pp.lower_rate_limit_interval ms]-> pace
    {<<VP()>>
    vp!<<vp@now>>
    &last_vp_or_vs:=now<<last_vp_or_vs=now>>};

  T8_VS_AFTER_VS:           --check if vs in VRP
  sense -[on dispatch vs]-> check_sense_vrp{};

  T9_VS_AFTER_VS_IN_VRP: -- vs in VRP, go back to "sense" state
  check_sense_vrp -[(now-last_vp_or_vs)<
      pp.ventricular_refractory_period]-> sense{};

  T10_VS_AFTER_VS_IS_NR: --vs after VRP is non-refractory
  check_sense_vrp -[(now-last_vp_or_vs)>=
      pp.ventricular_refractory_period]-> sense
    --reset timeouts with nr_vs! port send
    { <<VS()>>
    nr_vs!<<nr_vs@now>> --non-refractory ventricular sense
    &last_vp_or_vs:=now<<last_vp_or_vs=now>>};

  T11_STOP:                 --turn off pacing
  start,pace,sense -[on dispatch stop]-> off{};
**}; --end of annex subclause

end VVI;
--end of VVI.aadl
```

Listagem B.1: Código fonte do modelo do marca passo VVI.aadl

Referências Bibliográficas

[1] L. B. Becker, T. Correa, J. Bodeveix, J. Farines, and M. Filali. Verification based development process for embedded systems. *ERTS*, May 2010.

[2] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. *Department of Computer Science, Aalborg University, Denmark*, November 2004.

[3] B. Berthomieu, P. O. Ribet, and F. Vernadat. The tool tina - construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 2004.

[4] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an intermediate language for model verification in the topcased environment. *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France)*, January 2008.

[5] B. Berthomieu, J. P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, and F. Vernadat. Formal verification of aadl specifications in the topcased environment. *Reliable Software Technologies - Ada Europe 2009, Brest (France), 15p. Springer-Verlag, lncs 5570,*, pages 207–221, 2009.

[6] B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, and F. Vernadat. Formal verification of aadl models with fiacre and tina. *Embedded Real-time Software and Systems Conference (ERTS2010)*, May 2010.

[7] B. W. Boehm. Software engineering: R & d trends and defense needs. *Research Directions in Software Technology (P. Wegner, Ed.). MIT Press, Cambridge*, 1979.

[8] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Xavier Olive. Formal verification and validation of aadl models. *Proc. of Embedded Real Time Software and Systems Conf. (ERTS 2010)*, 2010.

[9] B. Bérard et. al. *Systems and Software Verification - Model Checking Techniques and Tools. Editora Springer.* 1999.

[10] J. C. Campos, J. Machado, and E. Seabra. Property patterns for the formal verification of automated production systems. *Proceedings of the 17th World Congress The International Federation of Automatic Control. Seoul, Korea,*, July 2008.

[11] J. Cardoso and R. Valette. *Redes de Petri. Editora da UFSC.* 1997.

[12] A. Cimatti, E. Clarke, F. Giunchiglia, , and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2000.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* 2010.

[14] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *lncs*, pages 52–71. springer, 1981.

[15] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[16] T. Correa. Translation, Validation and Improving of AADL to Fiacre Process in the TOPCASED Project, Monograph. *Universidade Federal de Santa Catarina*, 2009.

[17] M. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. *In Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[18] Jean-Marie Farines, Joni da Silva Fraga, and Rômulo Silva de Oliveira. *Sistemas de Tempo Real.* 12ª Escola de Computação, IME-USP, São Paulo-SP, 2000.

[19] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, 2004.

[20] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 0-387-94593-8.

[21] Radu Mateescu Hubert Garavel, Frédéric Lang. An overview of CADP. Technical report, 2001.

[22] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition).* Cambridge University Press, 2004. ISBN 052154310X.

[23] IEEE. Project 802, local networks standards, 1983.

[24] Eunkyoung Jee, Shaohui Wang, Jeong Ki Kim, Jaewoo Lee, Oleg Sokolsky, and Insup Lee. A safety-assured development approach for real-time software. In *Proceedings of the 2010 IEEE 16th International Conference on Embedded and Real-Time Computing*

Systems and Applications, RTCSA '10, pages 133–142, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4155-6.

[25] J.-P. Katoen, I.S. Zapreev, E. M. Hahn, H. Hermanns, and D.N. Jansen. The ins and outs of the probabilistic model-checker MRMC. *Quantitative Evaluation of Systems (QEST), IEEE CS Press*, pages 167 – 176, 2009.

[26] Konstantin Korovin. Notas de aula para lógica em ciências da computação. *School of Computer Science, The University of Manchester*, 2006. URL `http://www.cs.man.ac.uk/~korovink/cs2142/`.

[27] Brian R Larson. Behavior Language for Embedded Systems with Software Annex Sublanguage for AADL DRAFT Version v0 . 13. 2010.

[28] David C. Luckham, James Vera, and Sigurd Meldal. Foundations of component-based systems. chapter Key concepts in architecture definition languages, pages 23–45. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-77164-1.

[29] Grant Malcolm. Behavioural equivalence, bisimulation, and minimal realisation. In *Recent Trends in Data Type Specifications*, pages 359–378. Springer, 1996.

[30] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. *SIGSOFT Softw. Eng. Notes*, 22:60–76, November 1997. ISSN 0163-5948.

[31] K. G. Oliveira. Utilization of the Chain Fiacre-Tina for Systems Verification, Monograph. *Universidade Federal de Santa Catarina*, 2010.

[32] R. G. Oliveira. Integration of the Fiacre Language in the Topcased Environment, Monograph. *Universidade Federal de Santa Catarina*, 2009.

[33] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981. ISSN 0304-3975.

[34] J. M. Roussel and J.J. Lesage. Validation and verification of grafcet using state machine. 1996.

[35] SAE. SAE AS5506, Architecture Analysis and Design Language (AADL). Technical report, Novembro 2004.

[36] SAE. SAE AS5506/1, Architecture Analysis and Design Language (AADL) Error Model Annex Volume 1. Technical report, junho 2006.

[37] SAE. SAE AS5506, Annex Behavior Specification v1.6. Technical report, 2007.

[38] Boston Scientific. Pacemaker system specification. Technical report, January 2007.

[39] AADL Team SEI. An Extensible Open Source AADL Tool Environment(OSATE). Technical report, junho 2006.

[40] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 2003.

[41] J. N. Souza. *Lógica para Ciência da Computação. Ed. Campus*. 2002.

[42] A. S. Tanenbaum. *Computer Networks, Third Edition*. 1996.

[43] Topcased. Toolkit in open-source for critical applications and systems development, 2011. URL `http://www.topcased.org`.