RICARDO BEDIN FRANÇA

UMA ABORDAGEM PARA MODELAGEM E VERIFICAÇÃO DE PROTOCOLOS SÍNCRONOS DE BARRAMENTOS DE COMUNICAÇÃO

FLORIANÓPOLIS
2009

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

UMA ABORDAGEM PARA MODELAGEM E VERIFICAÇÃO DE PROTOCOLOS SÍNCRONOS DE BARRAMENTOS DE COMUNICAÇÃO

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.

RICARDO BEDIN FRANÇA

Florianópolis, março de 2009.

Uma Abordagem para Modelagem e Verificação de Protocolos Síncronos de Barramentos de Comunicação

Ricardo Bedin França

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia de Automação e Sistemas, Área de Concentração em 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.

Jean-Marie Farines, Dr.
Orientador

Leandro Buss Becker, Dr.
Co-orientador

Mamoun Filali Amine, Dr.
Co-orientador

Eugênio de Bona Castelan, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia de Automação e Sistemas

Banca Examinadora:

Jean-Marie Farines, Dr.
Orientador

Leandro Buss Becker, Dr.
Co-orientador

Guilherme Bittencourt, Dr.

Luigi Carro, Dr.

Luiz Claudio Villar dos Santos, Dr.

Max Hering de Queiroz, Dr.

À minha família.

Agradecimentos

Agradeço à tropa “da casa” (meus “velhos”, minha irmã, meu cunhado e nosso pássaro) por suportar a minha ausência durante um ano, e, principalmente, meu retorno em tempos de fim de dissertação, inundações e rebaixamento do Figueirense.

Agradeço a meus orientadores Jean-Marie, Leandro e Mamoun pela confiança em meu trabalho, por todos os ensinamentos desde a graduação, e por todas as portas abertas a partir de nosso trabalho.

Agradeço à minha amiga Diana, ela faz o que pode para que eu não esqueça meu lado humano em alguma de minhas andanças.

E agradeço aos amigos Guga e Diogo pela disposição permanente a apagar as agruras da vida.

Resumo

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

UMA ABORDAGEM PARA MODELAGEM E VERIFICAÇÃO DE PROTOCOLOS SÍNCRONOS DE BARRAMENTOS DE COMUNICAÇÃO

Ricardo Bedin França

Março/2009

Orientador: Jean-Marie Farines, Dr.

Área de Concentração: Sistemas Computacionais.

Palavras-chave: Protocolos de barramento, protocolos síncronos, AADL, Event-B, sistemas parametrizados.

Número de Páginas: 144.

Este trabalho apresenta um estudo a respeito de protocolos para barramentos de comunicação, enfatizando os aspectos de parametrização e sincronização vistos em tais protocolos. De acordo com estas características e sua influência na modelagem e verificação de sistemas que utilizam os barramentos, buscaram-se métodos e ferramentas adaptados a sistemas embarcados, síncronos e parametrizados. A abordagem utilizada neste trabalho consiste na especificação de protocolos em duas perspectivas distintas para ressaltar tanto os aspectos da arquitetura dos sistemas com barramento quanto o comportamento descrito pelos protocolos. A modelagem de arquitetura foi realizada com a utilização da linguagem de descrição de arquitetura AADL. A modelagem de comportamento utilizou a linguagem síncrona LUSTRE para permitir a criação de um modelo de fácil compreensão e simulação. O método Event-B foi escolhido para a modelagem e verificação comportamental e sua semântica orientada a refinamentos permitiu a criação de um modelo de base abstrato e genérico que pode ser reutilizado em protocolos síncronos com arbitração centralizada. A partir deste modelo, os protocolos PCI e AMBA foram utilizados como estudo de caso para especificação e verificação.

Abstract

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

AN APPROACH FOR MODELING AND VERIFICATION OF SYNCHRONOUS BUS PROTOCOLS

Ricardo Bedin França

March/2009

Advisor: Jean-Marie Farines, Dr.

Area of Concentration: Computational Systems.

Keywords: bus protocols, synchronous protocols, AADL, Event-B, parameterized systems.

Number of Pages: 144.

This dissertation presents a study of communication bus protocols, with emphasis on parameterization and synchronization aspects seen in such protocols. According to these characteristics and their influence in the modeling and verification of systems that use buses, suitable methods and tools were sought to fulfill the needs of synchronous, embedded systems with parameterization. This work describes an approach based on the specification of protocols from two distinct perspectives in order to underline the architecture of bus systems and the protocol behavior. The system architecture was modeled with the architecture description language AADL. The protocol behavior was modeled firstly with the LUSTRE synchronous language, in order to have a specification which is easy to understand and simulate. The Event-B method was used to the behavioral specifications and verification – its refinement-oriented semantics was used to create an abstract, generic model that can be reused as a basis to synchronous bus protocols with centralized arbitration. The PCI and AMBA bus protocols were used as case studies for specification and verification using this abstract model.

Sumário

Lista de Figuras

Lista de Siglas e Abreviaturas

AADL Architecture Analysis and Design Language

ADL Architecture Description Language

AHB Advanced High-performance Bus

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

ASB Advanced System Bus

CTL Computational Time Logic

MDL Machine Description Language

ML MetaLanguage

OSATE Open-Source AADL Tool Environment

PCI Peripheral Component Interconnect

SoC System-on-a-Chip

SPICES Support for Predictable Integration of mission Critical Embedded Systems

TLA Temporal Logic of Actions

TOPCASED Toolkit in OPen-source for Critical Application & SystEms Development

UML Unified Modeling Language

XML eXtensible Markup Language

Parte I

Introdução

1 Introdução

1.1 Motivação

O barramento de comunicação é um elemento fundamental na arquitetura dos sistemas embarcados, dada sua importância tanto na arquitetura física quanto lógica dos sistemas:

- Do ponto de vista da arquitetura de hardware, ele constitui um elemento de base para o dimensionamento da configuração do sistema: os periféricos físicos devem estar em conformidade com o protocolo físico definido pelo barramento, além disso, o desempenho dos periféricos (medido com indicadores tais como velocidade de transmissão de dados e tempo de latência) será determinado pelo desempenho do barramento.
- Na arquitetura de software, o barramento define uma topologia de comunicação e uma estrutura de configuração dinâmica dos componentes de software que se comunicam através dele.

Barramentos de comunicação são caracterizados por terem protocolos comumente especificados sem que o número de dispositivos conectados no barramento seja conhecido com antecedência. Isso faz com que os barramentos sejam caracterizados como *sistemas parametrizados*: "sistemas que são formados por um número potencialmente alto de instâncias de módulos simples" [45]. Visto que sistemas parametrizados são comumente encontrados no domínio de projeto de protocolos e circuitos eletrônicos [31], seu estudo está em voga na concepção de sistemas embarcados.

1.1.1 Os projetos TOPCASED e SPICES

O projeto *Toolkit in OPen-source for Critical Application & SystEms Development* (TOPCASED) [57] tem como objetivo a criação de métodos e ferramentas *open-source* para o desenvolvimento de sistemas críticos, com os seguintes princípios em mente [58]:

- O ciclo de vida das ferramentas deve ser longo, ao contrário da tendência atual imposta pelas ferramentas proprietárias que são criadas e extintas de acordo com interesses econômicos e estratégicos dos desenvolvedores.
- Os custos de aquisição das ferramentas devem ser minimizados. As ferramentas proprietárias costumam ter o custo de aquisição proporcional ao número de usuários que as utilizam numa empresa, e as ferramentas TOPCASED devem ter custos mais constantes por terem o código aberto.
- Os avanços obtidos no meio acadêmico devem ser integrados ao projeto rapidamente.

Diversas empresas e instituições acadêmicas, incluindo o Departamento de Automação e Sistemas da UFSC, tomam parte no projeto TOPCASED. Algumas ferramentas desenvolvidas do escopo deste projeto já encontram-se disponíveis no site TOPCASED.

O projeto *Support for Predictable Integration of mission Critical Embedded Systems* (SPICES) [53] busca desenvolver metodologias para o desenvolvimento de sistemas embarcados críticos para sistemas de aviação, vôo espacial e telecomunicações. Empresas francesas, belgas e israelenses estão envolvidas neste projeto, que utiliza a linguagem AADL [1] como ponto central: a partir de especificações feitas nesta linguagem, deseja-se fazer análises de escalonabilidade e comportamento, simulação e geração de código [54]. Também busca-se a integração das ferramentas desenvolvidas no projeto com as ferramentas TOPCASED.

1.2 Objetivos

O trabalho aqui apresentado situa-se no contexto dos projetos TOPCASED e SPICES, e possui como objetivo o desenvolvimento de uma abordagem de modelagem e verificação parametrizada de protocolos de barramentos de comunicação, utilizando métodos e ferramentas atualmente disponíveis.

Especificamente, deseja-se estudar os barramentos e seus protocolos em dois aspectos:

Arquitetura A modelagem e análise da arquitetura de sistemas com barramentos permite que seus componentes e conexões sejam descritos com precisão. Neste ponto, utiliza-se a linguagem AADL para descrever e analisar arquiteturas típicas de sistemas com barramentos.

Comportamento A modelagem e verificação do comportamento dos protocolos utilizados pelos dispositivos conectados ao barramento tem como objetivo validar o funcionamento destes protocolos através da verificação formal de suas propriedades desejadas. Tenciona-se analisar a viabilidade de utilização do método Event-B para a modelagem e verificação parametrizada dos protocolos que descrevem o comportamento destes barramentos.

1.3 Organização do Documento

Esta dissertação contém cinco partes: a parte I consiste nesta introdução, a parte II contém a base teórica utilizada no trabalho, a parte III expõe os estudos de caso realizados, a parte IV apresenta as conclusões do trabalho e a parte V contém o código completo das especificações criadas.

A parte II é composta de quatro capítulos. As definições básicas de barramentos de comunicação são vistas no capítulo 2, que também contém descrições dos protocolos utilizados como estudos de caso e características do problema de modelagem deste tipo de sistema, bem como trabalhos anteriores que são relacionados a esta dissertação. O capítulo 3 contém explicações relevantes à teoria de arquitetura de sistemas, às linguagens de descrição de arquitetura (ADLs) e aprofunda-se na descrição da linguagem AADL, utilizada posteriormente. O capítulo 4 apresenta o paradigma das linguagens síncronas de programação e modelagem, juntamente com a linguagem LUSTRE. O método B e sua variação Event-B são apresentados no capítulo 5.

A parte III também é composta de quatro capítulos. O capítulo 6 introduz a abordagem utilizada para os estudos de caso dos capítulos seguintes. O capítulo 7 expõe as especificações de arquitetura criadas em AADL e analisadas com as ferramentas desta linguagem. O capítulo 8 apresenta as especificações realizadas em LUSTRE e Event-B para descrever o comportamento dos protocolos estudados, bem como as verificações necessárias para garantir a correção destas especificações.

A parte IV contém o capítulo 9, que expõe as conclusões dos trabalhos realizados e possíveis trabalhos futuros.

A parte V contém dois apêndices: o apêndice A contém o código AADL do exemplo de fluxo de dados visto no capítulo 3 e do estudo de caso do barramento PCI, e o apêndice B contém todas as especificações feitas em Event-B ao longo deste trabalho.

Parte II

Fundamentos teóricos

2 Barramentos de Comunicação

2.1 Introdução

Um barramento é um caminho que dois ou mais dispositivos utilizam para se comunicar [55]. Este caminho é compartilhado por todos os dispositivos nele conectados, assim, todos têm a possibilidade de ler as informações transmitidas e também de transmitir suas próprias informações. A consequência evidente deste tipo de estrutura é que o acesso ao barramento deve ser controlado, para que apenas um dispositivo transmita uma informação em um dado instante, já que a sobreposição de sinais no barramento iria corromper as informações recebidas pelos dispositivos que as aguardam.

Normalmente, um barramento é composto por um conjunto de linhas de comunicação, onde cada uma transmite um valor binário. Estas linhas podem ser de três tipos:

Linhas de dados As linhas de dados são utilizadas para a transmissão de informações de um dispositivo para outro. A velocidade de transmissão de dados depende diretamente da quantidade de linhas de dados disponíveis.

Linhas de endereço As linhas de endereço são utilizadas para a transmissão de informações relativas aos endereços e portas de entrada e saída dos dispositivos que participam das trocas de dados, permitindo que eles possam reconhecer seu próprio endereço quando forem solicitados a participar de uma transferência.

Linhas de controle As linhas de controle são utilizadas para o gerenciamento do acesso ao barramento. Através dessas linhas são transmitidas as requisições e autorizações de utilização das linhas de dados e de endereço.

Os barramentos podem ser classificados de acordo com suas características físicas e lógicas. Neste trabalho, adota-se a classificação utilizada por Stallings [55], que distingue os barramentos de acordo com o tipo de utilização de suas linhas, o método de arbitração

do acesso, a temporização das ações, a largura dos barramentos e os tipos de operações permitidas.

Tipos de Barramentos As linhas de um barramento podem ser dedicadas a realizar sempre a mesma função, ou a trocar de função de acordo com a fase de uma transferência, caracterizando a multiplexação no tempo. Um exemplo típico de multiplexação é a utilização do mesmo conjunto de linhas para realizar as funções de linhas de endereço, no início de uma transferência, e para a transmissão dos dados em seguida. Neste caso, uma linha auxiliar é usada para a confirmação de que o endereço dado foi reconhecido por algum dispositivo.

Métodos de Arbitração O acesso às linhas de dados e controle de um barramento pode ser gerenciado por um dispositivo centralizado, usualmente conhecido como árbitro, mas também pode ser gerenciado de forma conjunta por todos os dispositivos de acordo com algum processo decisório para conceder o controle do barramento a um deles.

Temporização das ações O comportamento temporal de um barramento pode ser síncrono ou assíncrono. Em barramentos síncronos, todos os dispositivos têm suas ações coordenadas por um relógio comum. A leitura dos valores do barramento ocorre no início do ciclo de relógio, e em seguida novos valores podem ser atribuídos para serem lidos no ciclo seguinte. Em barramentos assíncronos, os dispositivos podem realizar ações em qualquer instante, ocasionando eventualmente reações de outros dispositivos.

Largura dos barramentos A largura do barramento tem influência direta na taxa de transmissão de dados e na quantidade de endereços que podem ser representados.

Tipos de transferências Além das operações básicas de leitura e escrita, os barramentos podem permitir operações mais elaboradas, tais como leitura seguida de escrita e transferência de um bloco de informações.

O funcionamento de um protocolo de barramento pode-se dividir em duas fases: arbitração e transferência. Na arbitração, um dos dispositivos recebe a autorização para enviar sinais às linhas de dados e endereço, tornando-se o dispositivo "mestre"; na transferência, o mestre envia um comando (como leitura ou escrita) e o endereço do dispositivo que deve lhe enviar ou receber dados (o "escravo"), para em seguida realizar a troca

de dados. Deste modo, a fase de transferência pode ser dividida em duas outras fases: endereço e dados.

Para facilitar a especificação destes protocolos, convém identificar as características que estão presentes em diferentes níveis de abstração. A temporização do barramento pode ser tratada como o elemento de mais alto nível, já que todos os dispositivos (incluindo o árbitro, se este for centralizado) devem ter seu comportamento baseado no fato do barramento ser ou não síncrono. O segundo elemento mais abstrato é o método de arbitração, pois este define a inclusão ou não de um árbitro no barramento. Em um nível algo mais baixo, tem-se o tipo de transferência, que define o modelo da fase de transferência do protocolo. O tipo e a largura do barramento são estudados e detalhados em modelos mais próximos da implementação, já que lidam com variáveis concretas e aspectos físicos do sistema.

Considerando que a estrutura de um protocolo de barramento muda fundamentalmente de acordo com sua temporização e seu comportamento de alto nível depende do método escolhido para arbitração, a metodologia apresentada neste trabalho será focada para um tipo específico de temporização e um método específico de arbitração. Assim, foram escolhidos os protocolos de temporização síncrona e arbitração centralizada, que são amplamente utilizados em sistemas embarcados e comumente utilizados como casos de estudos em metodologias de projeto e verificação. Para facilitar a nomenclatura, o dispositivo responsável pela arbitração será chamado de "árbitro", e os dispositivos que desejam realizar transferências através do barramento serão chamados de "controladores".

2.2 O Protocolo PCI

O protocolo PCI (*Peripheral Component Interconnect* [47]) foi criado em 1992, com o objetivo de ligar periféricos de microcomputadores aos processadores e dispositivos de memória destes. De acordo com a classificação vista anteriormente, o PCI é um protocolo síncrono, com arbitração centralizada, com possibilidade de leituras/escritas simples ou em bloco, com linhas multiplexadas para dados e endereço, e largura de 32 ou 64 bits. Dada a disponibilidade de sua especificação e sua popularidade em arquiteturas conhecidas de microcomputadores[1], este protocolo foi escolhido para ser o principal caso de estudo deste trabalho.

[1] Os controladores de vídeo dos microcomputadores atuais costumam ser conectados através de barramentos *PCI Express*. Esta dissertação utiliza a versão *PCI Local Bus*.

Um controlador em conformidade com o protocolo PCI possui no mínimo 47 linhas, se não precisar receber o controle do barramento, ou 49, se puder exercer a função de "mestre" de uma transferência. Para o escopo deste estudo, assume-se que todos os controladores podem assumir o papel de mestre e de escravo. Para simplificar o funcionamento do protocolo, serão descartadas as linhas de registro de erros *PERR* e *SERR*, assim como a linha de paridade *PAR*, a linha de interrupção de transmissão *STOP*, a linha de reinicialização *RST* e a linha *IDSEL*, utilizada para configuração na inicialização do sistema. Portanto, os controladores especificados nesta dissertação utilizam 43 linhas:

REQ Linhas de controle que indicam requisições feitas pelos controladores. Cada controlador possui, portanto, direito de escrita sobre uma linha REQ, e o árbitro tem direito de leitura sobre todas as linhas REQ.

GNT Linhas de controle que indicam concessões de acesso ao barramento feitas pelo árbitro. Cada controlador possui direito de leitura sobre uma linha GNT e o árbitro tem direito de escrita sobre todas as linhas GNT.

AD Conjunto de 32 linhas utilizadas tanto para endereço quanto para dados. Todos os controladores têm acesso de leitura e escrita a este conjunto de linhas.

C/BE Quatro linhas de controle utilizadas para a transmissão do comando desejado de transferência e, durante a transferência, para enviar a quantidade de bits da linha AD que devem ser levados em conta. Neste estudo, apenas a função de comando é relevante. Todos os controladores podem acessar estas linhas, tanto para ler quanto para escrever.

IRDY, TRDY Linhas de controle que indicam, respectivamente, que os controladores mestre e escravo estão preparados para iniciar ou prosseguir uma transferência. Todos os controladores podem acessar e modificar estas linhas.

FRAME Linha de controle utilizada para indicar o início e a duração de uma transferência. Todos os controladores podem acessar e modificar esta linha.

DEVSEL Linha de controle utilizada para indicar que o endereço dado na linha AD foi reconhecido por um controlador. Todos os controladores podem acessar e modificar esta linha.

CLK Linha que transmite o sinal do relógio do sistema.

Um barramento PCI pode possuir outras linhas para acrescentar funcionalidades ao protocolo. Uma implementação 64 bits, por exemplo, tem necessidade de outras 32 linhas *AD*, quatro linhas *C/BE*, uma nova linha de paridade, uma linha *REQ64* para requerer uma transferência 64 bits, e uma linha *ACK64* que é usada pelo escravo para sinalizar que ele reconheceu seu endereço para uma transferência que utilizará 64 bits.

A arbitração no PCI é caracterizada por ser baseada em acesso (ao contrário de protocolos baseados em fatias de tempo para cada controlador) e por ser "oculta", isto é, não consumir ciclos somente com arbitração, exceto quando o barramento encontra-se ocioso. Quando o barramento está ocioso e algum controlador faz uma requisição através de sua linha *REQ*, o árbitro concede o acesso enviando um sinal pela linha *GNT* deste controlador no período seguinte. Se o barramento está ocupado e o controlador que o ocupa retira o sinal da linha *REQ*, no período seguinte o árbitro acionará a linha *GNT* de algum controlador que esteja aguardando o acesso (sinalizando esta situação através de um pedido pela linha *REQ*), realizando assim uma troca de mestres sem perda de ciclos com arbitragem, ou ficará ocioso se nenhum controlador estiver aguardando a concessão de acesso. A política de fornecimento de acesso em caso de haver vários controladores requerendo acesso não é estabelecida no protocolo, podendo assim ser escolhida de acordo com sua aplicação.

Após a arbitração, segue-se a fase de transferência do protocolo. Neste trabalho são abordadas as operações de leitura e a escrita simples, por sua maior facilidade de compreensão. Uma operação de leitura normal inicia-se quando o controlador mestre recebe o acesso ao barramento. Assim que ele faz a leitura de uma autorização em sua linha *GNT*, ele aciona as linhas *FRAME*, sinalizando o início de transferência, e *AD*, com o endereço do controlador escravo que deve lhe enviar dados. No ciclo seguinte, as linhas *AD* entram em espera para serem utilizadas pelo escravo, procedimento que sempre ocorre em linhas que devem ser utilizadas por mais de um controlador. Neste mesmo ciclo, o mestre aciona a linha *IRDY*, sinalizando que está pronto para a transferência. Em seguida, o escravo ativa a linha *DEVSEL* para sinalizar que foi encontrado, a linha *TRDY* para indicar que está pronto, e envia o primeiro dado na linha *AD*. Os outros envios de dados seguem normalmente enquanto *IRDY* e *TRDY* estiverem ativados, e o mestre, possuindo conhecimento do número de dados a serem enviados, desativa as linhas *REQ* e *FRAME* enquanto recebe seu penúltimo dado. Assim, no período seguinte, o controlador escravo envia seu último dado e o árbitro ativa a linha *GNT* do controlador que for determinado por sua política de arbitração. Por fim, as linhas *IRDY*, *TRDY* e *DEVSEL* são desativadas e as linhas *C/BE*, *AD* e *FRAME* têm um ciclo de espera.

A operação de escrita apresenta algumas diferenças em relação à sequência anterior: após o ciclo em que o mestre coloca o comando e o endereço do escravo, ele envia imediatamente o primeiro dado na linha *AD*, já que não é necessário um ciclo de espera pois esta linha continua sendo controlada pelo mesmo controlador. Enquanto isso, o escravo deve ativar as linhas DEVSEL para mostrar que foi reconhecido, e *TRDY* para sinalizar que está preparado para receber dados. No ciclo seguinte, o mestre nota que *TRDY* está acionado e envia um novo dado, repetindo a operação até o penúltimo, onde a operação é efetuada normalmente enquanto a linha *REQ* é desativada. Ao enviar seu último dado, o mestre desativa a linha *FRAME* para sinalizar ao escravo de que não há mais dados, então, no ciclo seguinte, as linhas que são desativadas ou postas em espera são as mesmas da operação de leitura.

2.3 O Protocolo AMBA

O protocolo *Advanced Microcontroller Bus Architecture* (AMBA [7]) foi desenvolvido em 1995 pela empresa ARM [6]. Inicialmente utilizado nos processadores ARM, atualmente o protocolo AMBA difundiu-se principalmente no desenvolvimento de sistemas computacionais completos fabricados em um único chip (SoCs). Neste trabalho é utilizada a versão 2 do protocolo, que tem como características o sincronismo, a arbitração centralizada, transferências básicas ou em bloco, linhas dedicadas (que podem ser modificadas para multiplexação das linhas de leitura e escrita) e largura de 32 bits. Nesta versão, o protocolo AMBA pode ser implementado como um barramento de sistema normal (ASB), de alta performance (AHB) ou um barramento para periféricos (APB); a descrição de comportamento apresentada nesta seção corresponde ao barramento AMBA AHB. As principais linhas deste barramento são:

HBUSREQ Linhas de controle que indicam requisições feitas pelos controladores, de modo análogo às linhas *REQ* do protocolo PCI.

HGRANT Linhas de controle semelhantes às linhas *GNT* do protocolo PCI.

HMASTER Linha de controle utilizada pelo árbitro para indicar o controlador que está efetuando uma transferência no barramento.

HTRANS Duas linhas de controle utilizadas pelo controlador mestre, elas indicam o tipo de transferência que está sendo executado.

HADDR 32 linhas de endereço utilizadas para que o controlador mestre indique qual será o controlador escravo na transferência.

HWDATA 32 linhas de dados utilizadas para operações de escrita.

HRDATA 32 linhas de dados utilizadas para operações de leitura, pode-se ampliar a largura do barramento para 64 bits multiplexando-se as linhas *HWDATA* e *HRDATA*.

HREADY Linha de controle utilizada pelo controlador escravo para sinalizar se está pronto para receber ou enviar dados.

CLK Linha que transmite o sinal do relógio do sistema.

Assim como no protocolo PCI, a arbitração é "oculta" no protocolo AMBA, portanto, sempre que um controlador estiver ocupando o barramento e outro estiver em espera, as mudanças nos sinais *HGRANT* de ambos são feitas ao mesmo tempo, evitando ciclos de espera devidos à arbitração. Os controladores pedem acesso ao barramento através das linhas *HBUSREQ* e um deles recebe o controle através de sua linha *HGRANT*. Alguns tipos de transferência são realizados com uma duração pré-estabelecida em ciclos, e outros exigem a constante manutenção de um sinal na linha *HBUSREQ* do controlador mestre. No fim de uma transferência, o árbitro passa o controle a algum controlador que esteja aguardando, ou na ausência de controladores que tenham feito uma requisição para utilizar o barramento, o controle é passado para um controlador definido como "mestre padrão". Neste caso, o mestre padrão utiliza as linhas *HTRANS* para sinalizar que não está realmente ocorrendo uma transferência.

Uma operação básica de transferência é bastante simples devido às linhas dedicadas do barramento. No período seguinte à recepção do controle do barramento por um controlador, este coloca o endereço do escravo na linha *HADDR* e o árbitro coloca seu endereço na linha *HMASTER*. Nos períodos seguintes, o mestre coloca dados nas linhas *HWDATA* e o escravo coloca dados nas linhas *HRDATA*. No início do período da última transferência de dados, o árbitro realiza a troca de mestres, assim, numa transferência de tamanho não informado, o mestre em operação deve desativar sua linha *HBUSREQ* quando estiver enviando o antepenúltimo pacote de dados, para que o árbitro leia este sinal no período seguinte e desative o respectivo sinal *HGRANT* em seguida.

2.4 Modelagem e Verificação de Protocolos de Barramentos

2.4.1 Modelagem e Análise da Arquitetura

A modelagem de arquitetura tem como objetivo descrever os componentes e ligações de hardware e software do sistema, assim como o mapeamento hardware-software. Um bom modelo de arquitetura deve ser capaz de exprimir com facilidade as características físicas dos componentes de hardware e relacioná-los aos módulos de software que são neles executados. Além disso, o modelo deve ser expresso com formalismos poderosos o suficiente para que a representação dos componentes seja similar à vista no sistema real. Para que o modelo seja útil, ele deve ser adequado, no mínimo, para ser submetido a verificações básicas em sua arquitetura: por exemplo, deve-se verificar que os componentes estão corretamente conectados e que os mapeamentos estão de acordo com o desejado. No contexto de barramentos de comunicação, deve-se mostrar claramente no modelo que todos os componentes de hardware são conectados através de um componente "barramento", caracterizando-o com o maior número possível de detalhes sobre suas características físicas e de desempenho.

Para a criação de modelos de representação da arquitetura dos protocolos de barramento, este trabalho utiliza o paradigma das linguagens de descrição de arquitetura (ADLs), que serão estudadas no capítulo 3. Tais linguagens surgiram com o propósito de facilitar a especificação de arquiteturas de sistemas computacionais, e algumas delas são voltadas para os sistemas embarcados, assim, sendo opções viáveis para a representação de sistemas que utilizam barramentos. Outros tipos de linguagens também poderiam ser utilizados para modelar arquiteturas de sistemas com barramentos: entre as linguagens de modelagem, destacam-se os perfis UML-RT [60] e MARTE [46], que enfatizam os aspectos de tempo-real dos sistemas. Linguagens comumente utilizadas para a descrição de hardware também têm evoluído para permitir a descrição de sistemas em níveis abstratos, assim, há também a possibilidade de descrever arquiteturas com as linguagens SystemC [56] e SystemVerilog [37].

2.4.2 Modelagem e Verificação do Comportamento

A modelagem do comportamento enfatiza os aspectos específicos do protocolo que os controladores respeitam para utilizar o barramento. Os aspectos de hardware são abstraídos aqui e apenas o comportamento do software do sistema deve ser levado em conta.

Os modelos de comportamento devem ser feitos de modo que possam ser analisados por ferramentas de verificação formal, com o intuito de provar que eles realmente respeitam as propriedades desejadas pelos projetistas – deve-se provar, por exemplo, que um protocolo de barramento impede que dois controladores enviem sinais ao mesmo tempo para as linhas de dados ou endereço.

Uma dificuldade adicional vista em sistemas com barramento é que sua verificação comportamental precisa ser parametrizada, isto é, o protocolo deve ser validado independentemente da quantidade de controladores nele conectados. Isto reduz significativamente a quantidade de métodos e ferramentas adequadas para a modelagem e verificação do comportamento, já que muitas não são capazes de realizar verificações parametrizadas.

A técnica de verificação automática por *model-checking* [14], bastante utilizada para verificações formais, não é adequada para todos os casos de verificação parametrizada [5], por levar a um problema indecidível. Para algumas topologias, tais como a configuração em anel, foi provado que o resultado para o caso geral pode ser obtido a partir de um determinado caso específico [19]. A utilização de *model-checking* para verificação parametrizada precisa ser avaliada separadamente para cada sistema específico, a fim de provar que o problema é decidível para o caso em questão. Emerson e Namjoshi fazem uso de *model-checking* para especificar o protocolo de barramento SAE-J1850 [20]. Neste trabalho, a parametrização ocorre tanto no número de controladores conectados ao barramento quanto no atraso que pode ocorrer nos circuitos que detectam mudanças de estado no barramento. A verificação parametrizada para o número de controladores é feita através da divisão do sistema em um processo de controle e várias instâncias de um processo de usuário, e o método permite a verificação de fórmulas CTL separadamente para o processo de controle, para todas as instâncias do processo de usuário, ou para todos os pares distintos de instâncias do processo de usuário. A utilização de *model-checking* tem a grande vantagem de ser automática, porém as propriedades que podem ser verificadas devem ser dos tipos acima mencionados.

Pal *et al* criaram a linguagem *BUSpec* para descrever protocolos de barramentos [43]. Ela permite a modelagem parametrizada e hierárquica de protocolos síncronos, e sua compilação gera máquinas de estados finitos. A linguagem é bastante interessante para a modelagem de protocolos de barramentos, já que é especificamente voltada para tal, porém este trabalho não aborda o tema de verificação parametrizada, visto que o sistema pode ser modelado de forma parametrizada mas exige a especificação dos parâmetros para compilação e verificação.

Zimmermann utilizou o método Event-B para modelar, verificar e gerar código VHDL para circuitos eletrônicos, utilizando um protocolo de barramento como estudo de caso [63]. Event-B apresenta a desvantagem de suas técnicas de prova não apresentarem o grau de automatização disponível no *model-checking*, mas esta limitação não é crucial na especificação de um protocolo de barramento de comunicação, pois este tipo de protocolo não costuma ser excessivamente complexo. Visto que seu trabalho é voltado ao desenvolvimento de circuitos eletrônicos em geral, e não de barramentos, não há um foco em aspectos específicos à modelagem de protocolos de barramentos.

França *et al* [24] propuseram uma abordagem incremental de modelagem de protocolos de barramentos utilizando o Anexo Comportamental AADL e a linguagem de lógica temporal TLA+ [32]. Neste trabalho, a modelagem comportamental do sistema foi feita com ênfase no comportamento individual dos componentes, de modo que não são tratados os aspectos de verificação parametrizada. Outra dificuldade encontrada no uso do Anexo Comportamental é que sua semântica ainda não está completamente definida.

Oumalou *et al* [42] realizaram a modelagem, verificação e implementação do protocolo PCI utilizando UML [59], ASM [11] e SystemC. O modelo abstrato dos componentes do barramento e suas interações foi feito com o uso de diagramas UML, em seguida, tanto o modelo do sistema quanto as propriedades que deseja-se verificar são especificados em ASM, permitindo sua verificação com um *model-checker*. Por fim, é feita uma tradução das especificações em ASM para SystemC, permitindo sua implementação. Este mesmo método foi empregado com sucesso em [40], mas o problema da falta de parametrização do *model-checking* não é abordado.

Esta dissertação utiliza Event-B, descrito no capítulo 5, para a modelagem comportamental de protocolos de barramento, principalmente devido à sua orientação a refinamentos (que facilita a especificação passo-a-passo de um sistema) e sua naturalidade para representar sistemas parametrizados. A abordagem incremental de Event-B é utilizada para delinear os níveis de abstração que podem ser desejados na modelagem destes protocolos. Também utiliza-se nesta dissertação a linguagem LUSTRE [13] para criar e simular protótipos de protocolos síncronos, dada a natureza síncrona desta linguagem.

2.5 Conclusão

Neste capítulo, foram estudados os princípios dos barramentos de comunicação em sistemas computacionais. Visto que os barramentos são compartilhados pelos dispositivos

que são neles conectados, seus projetos devem ser corretamente especificados e validados, para manter a integridade das informações transmitidas e garantir um bom desempenho do sistema. Também foram apresentados os protocolo PCI e AMBA, que são os casos de estudo para a metodologia de modelagem e verificação que é apresentada em capítulos posteriores.

Dada a importância do barramento tanto nos aspectos de hardware quanto de software, seu estudo deve abranger tanto a arquitetura do sistema que o contém quanto o comportamento de seu protocolo. Para o estudo da arquitetura, o paradigma das linguagens de descrição de arquitetura (ADLs) é interessante, por ser voltado a resolver exatamente este problema. Para o estudo de comportamento, as técnicas conhecidas de *model-checking* foram preteridas a favor da utilização da verificação parametrizada propiciada pelo método Event-B e suas ferramentas, devido à dificuldade de empregá-las sem a necessidade de provar a sua decidabilidade para cada caso específico. A verificação de propriedades em Event-B apresenta a desvantagem de ser menos automatizada, mas possui vantagens importantes: é adaptada à verificação parametrizada e a modelagem é orientada a refinamentos, permitindo um desenvolvimento bastante seguro do ponto de vista metodológico.

3 Linguagens de Descrição de Arquitetura

3.1 Introdução

As linguagens de descrição de arquitetura (ADLs) surgiram como uma consequência dos conceitos de arquitetura de sistemas: a utilização de arquiteturas para representar sistemas complexos incentivou a busca de linguagens que aplicassem tais conceitos facilmente.

O termo "arquitetura" é bastante familiar, atualmente, aos projetistas de sistemas embarcados e *softwares*. Ainda assim, não há uma definição de arquitetura de sistemas aceita universalmente. Neste trabalho, utiliza-se a definição vista em [29]: "a organização fundamental de um sistema aplicada em seus componentes, as relações entre eles e também entre eles e o ambiente, e os princípios que guiam seu projeto e evolução". Uma definição bastante similar, para arquitetura de *software*, já era aceita por Garlan em 1995 [26]: "A estrutura de um programa, seus inter-relacionamentos, princípios e referências que governam seu projeto e evolução ao longo do tempo". Uma definição mais breve, porém útil para o entendimento dos fundamentos das linguagens de arquitetura, é dada por Luckham [36]: "a especificação de um sistema, identificando seus componentes e as interações entre eles, por meio de interfaces, conexões e restrições".

O uso da noção de arquitetura na informática para o projeto de sistemas complexos já é estudado há mais de duas décadas. Em 1987, Zachman [62] propôs um "framework" para representar a arquitetura de um sistema de informação sobre diversos pontos de vista diferentes, a fim de ter as informações necessárias para as diversas equipes de desenvolvimento de um sistema. Tal uso da arquitetura do sistema é baseado no uso tradicionalmente feito de arquiteturas de outros ramos, tais como edificações. A aplicação da arquitetura em vários papéis de um projeto é também mencionada por Clements [17] como uma sólida base para comunicação entre o pessoal envolvido no projeto, assim como um padrão para

projetos e um meio de ratificar decisões de projeto feitas no início do desenvolvimento do sistema.

3.2 Linguagens de Descrição de Arquitetura

As linguagens de descrição de arquitetura visam formalizar os princípios vistos na seção anterior e criar representações adequadas para estas arquiteturas. Nos anos 1990, várias ADLs foram desenvolvidas e aprimoradas para tentar satisfazer as necessidades de projetistas e desenvolvedores de diversos tipos de sistemas. Dadas as diferenças destas necessidades, não há um consenso no tocante à complexidade e ao escopo que uma ADL deve ter; a escolha de uma determinada ADL para um projeto de sistemas depende da complexidade do próprio sistema.

Distinguir as ADLs de outros paradigmas de linguagem nem sempre é uma tarefa trivial, mas as ADLs possuem, de fato, algumas características particulares. Medvidovic [38] destaca a utilização de "conectores" nas ADLs, enquanto outros tipos de linguagem não costumam representar explicitamente as conexões entre as partes. Clements [17] compara as ADLs com outros tipos de linguagem para delinear outras diferenças:

- As ADLs se diferenciam de linguagens de requisitos pois estas se concentram em descrever problemas, enquanto as ADLs descrevem arquiteturas de sistemas no espaço da solução.
- Em relação às linguagens de programação, pode-se constatar que as ADLs buscam uma descrição tão clara e útil quanto necessária da arquitetura do sistema, enquanto as linguagens de programação são utilizadas para dar soluções específicas e concretas.
- Tradicionalmente, as linguagens de modelagem se preocupam com o comportamento do sistema como um todo, enquanto ADLs representam a arquitetura componente a componente.

Com a evolução das linguagens de modelagem, torna-se ainda mais difícil distingui-las das ADLs, já que linguagens modernas de modelagem, tais como UML2 [59], agregaram noções vistas em ADLs, tais como "portas" em componentes, utilizadas para interação com o exterior. Ainda assim, o paradigma das ADLs pode ser distinto por seus elementos de base, descritos por Medvidovic [38]:

Componentes Unidades de computação ou armazenamento de dados. Componentes possuem interfaces que permitem sua comunicação com o ambiente. É comum que as ADLs possuam declarações separadas de tipos e implementações de componentes: cada tipo de componente possui uma interface definida, mas pode ser implementado de várias maneiras diferentes, possuindo diferentes estruturas internas. É importante que os componentes de uma ADL sejam extensíveis e reutilizáveis, e também que eles sejam capazes de conter outros componentes, para que possam ser declaradas hierarquias de componentes. As ADLs diferem entre si na capacidade e no modo de representar o comportamento interno de seus componentes.

Conectores Os conectores são entidades típicas das ADLs. Tais conectores formam as ligações entre as interfaces dos componentes, e, de acordo com a ADL, podem ser entidades simplificadas ou ter o mesmo nível de complexidade de um componente.

Configurações São grafos conectados de componentes e conectores, descrevendo uma estrutura de arquitetura. Uma ADL deve ser capaz de representar claramente as diferentes combinações de componentes e conectores, bem como seus respectivos comportamentos. Vários sistemas apresentam configurações que podem variar durante o funcionamento do sistema e as ADLs devem ser capazes de representar tais configurações dinâmicas.

As ADLs podem ser voltadas para *hardware* ou *software*. ADLs de *hardware*, também conhecidas como MDLs (*Machine Description Languages*), "buscam descrever detalhadamente a estrutura e o comportamento de arquiteturas de processadores" [39]. Linguagens como nML [21], ISDL [27], LISA [44] e EXPRESSION [28] são bastante utilizadas para descrever tais componentes e arquiteturas de *hardware*.

Neste trabalho, tal nível de detalhamento do *hardware* não é necessário, e, por outro lado, deseja-se obter uma representação da arquitetura e do mapeamento *hardware-software* dos sistemas com barramento. Assim, optou-se pela utilização de uma ADL de *software*, pois as ADLs de *software* proporcionam uma visão mais completa de todos os componentes presentes em sistemas embarcados.

Entre as principais ADLs de de *software* utilizadas atualmente, destacam-se algumas linguagens que representam arquiteturas de um modo mais abstrato e genérico, e outras que possuem tipos pré-definidos de componentes, apropriados para determinadas aplicações. A linguagem Rapide [35] representa as interações dos elementos de uma arquitetura através de conjuntos de eventos parcialmente ordenados. Seus componentes são

de uma categoria genérica e seus conectores são representados de maneira simplificada em declarações de arquiteturas, que representam configurações do sistema. Na linguagem ACME [2], as arquiteturas são representadas com componentes, conectores e configurações bastante genéricos a fim de facilitar a representação de um sistema descrito em ACME em outras linguagens, através de mapeamentos. Os componentes ACME podem conter propriedades em qualquer linguagem, os conectores são representados como elementos da mesma complexidade que componentes, e as configurações são representadas por sistemas, os quais agregam componentes, conectores e outros sistemas. Em Wright [4], componentes e conexões são vistos como elementos genéricos de mesma complexidade e a hierarquia de componentes é feita inserindo-se configurações nas descrições internas de componentes. A linguagem MetaH [61] apresenta características mais específicas de sistemas embarcados e de tempo-real, apresentando categorias pré-definidas de componentes para descrever elementos típicos de *hardware* e *software*, assim como tipos pré-definidos de conectores e uma biblioteca de propriedades adaptada a este contexto.

Levando-se em conta a orientação deste trabalho para sistemas embarcados críticos, a linguagem MetaH mostrou-se a mais interessante destas, por ter seu foco neste tipo de sistema, apresentando uma série de facilidades na representação de componentes, conectores e propriedades. Assim, o estudo das ADLs será focado na linguagem AADL, uma evolução da linguagem MetaH, que foi escolhida para modelar arquiteturas dos sistemas que aqui serão descritos.

3.3 Architecture Analysis and Design Language - AADL

3.3.1 Histórico

A criação da AADL teve início em 1993, nos laboratórios Honeywell[1], onde já tinha sido criada a linguagem MetaH. Nesta época, pretendia-se criar uma linguagem baseada em MetaH e voltada para o desenvolvimento de sistemas embarcados para a aeronáutica – inicialmente, a sigla AADL significava "Avionics Architecture Description Language". Assim, seu desenvolvimento, efetivamente, teve início com a criação da linguagem MetaH e suas ferramentas no fim da década de 1980. Esta linguagem era voltada especificamente para sistemas embarcados, apresentando uma semântica temporal bastante precisa e adequada para descrever tarefas e restrições de tempo real. Optou-se por utilizar MetaH como base para a AADL após constatar-se que o uso da linguagem e suas ferramentas

[1] `www.honeywell.com`

era de grande valia para desenvolver novos projetos, e os benefícios na manutenção e atualização de projetos existentes eram ainda maiores [49].

A AADL começou a ser padronizada em 2001, e em novembro de 2004 foi criado seu primeiro "standard". Atualmente, um novo "standard" está em curso de desenvolvimento, assim como anexos que estendem a capacidade da linguagem, tais como o modelo de erros [48], que permite uma representação detalhada de eventuais falhas que possam atingir o sistema, e o anexo comportamental [23] [25], que auxilia na descrição detalhada do comportamento interno de componentes.

Além da indústria aeronáutica, a AADL também tem como campos de interesse a indústria médica (equipamentos com alto grau de exigência de confiabilidade e segurança), controle automotivo, robótica e motores [61] – sistemas embarcados com restrições temporais e exigências que precisam ser respeitadas.

3.3.2 Especificações AADL

A arquitetura de um sistema é descrita em AADL com uma especificação AADL. Tal especificação contém declarações de componentes e pode eventualmente conter referências a especificações globais AADL: *packages* e *property sets*. *Packages* são estruturas globais semelhantes a pacotes vistos em outras linguagens: componentes nele declarados podem ser referenciados por todas as especificações AADL que têm acesso a ele. A função dos *property sets* será vista posteriormente.

Conforme já mencionado, o núcleo da linguagem AADL pode ser estendido por meio de anexos. Assim, os componentes de um sistema podem ter descrições referentes a anexos utilizados pelo projeto.

As hierarquias de componentes em AADL devem ter todas as instâncias de componentes declaradas individualmente, assim, não é possível fazer uma especificação parametrizada. Para especificar sistemas que utilizam barramentos de comunicação, deve-se conhecer previamente todos os dispositivos que serão conectados nele. Tal necessidade de um sistema completamente especificado impede verificações parametrizadas mas não atrapalha o estudo de sua arquitetura, já que esta deve ser especificada de acordo com o modo que o sistema realmente será implementado.

3.3.2.1 Componentes

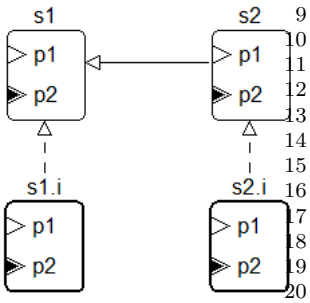
Um componente é "alguma entidade de *hardware* ou *software* que é parte de um sistema modelado em AADL" [50]. Um componente de uma especificação AADL normalmente é descrito em duas partes: uma declaração de tipo (*type*) que descreve sua interface, e uma ou mais declarações de implementação (*implementation*) que representam as diferentes características que o componente pode possuir internamente.

Descrições de componentes em AADL são iniciadas com a categoria do componente (uma dentre as pré-definidas na linguagem, tais como *system* ou *memory*), a palavra reservada *implementation* se for uma implementação de componente, e o nome deste componente. As características do componente são descritas com palavras reservadas AADL referentes a categorias de características – tais como interfaces externas, propriedades e subcomponentes – seguidas de sequências de asserções referentes à categoria de características desejada.Um exemplo desta sintaxe pode ser visto na figura 1.

Normalmente, os tipos de componentes possuem declarações de *features*. *Features* representam pontos de comunicação na interface de um componente, podendo ser portas (*ports*) que transmitem ou recebem dados e/ou eventos, acesso a componentes (*access*) que podem ser fornecidos ou requeridos, ou sub-rotinas que podem ser chamadas por outros componentes. As *features* que representam portas e acessos devem ser direcionadas: no caso de *ports*, como entrada (*in*), saída (*out*) ou ambos (*in out*) e no caso de *features* de *access*, o acesso pode ser fornecido (*provides access*) ou pedido (*requires access*). Os *types* também podem conter declarações de fluxo de dados *(*flows) em suas interfaces e propriedades (*properties*) aplicáveis para todas as suas possíveis implementações. *Types* podem ser declarados como extensões de outros *types*, herdando sua especificação e podendo agregar novas características de modo semelhante ao visto em programação orientada a objetos, além de refinar características que não foram completamente definidas no *type* original, e alterar valores de propriedades.

A implementação de um componente define seus aspectos internos: subcomponentes (*subcomponents*), conexões (*connections*) entre as interfaces dos subcomponentes ou entre a interface de um subcomponente e a interface do componente que o contém, propriedades específicas da implementação e descrições internas de fluxos de dados. Os subcomponentes são caracterizados por um nome de instância e um componente, que pode ser um *type* ou uma implementação. Visto que a implementação pode também conter subcomponentes, há a possibilidade de formar uma hierarquia de componentes.

Na figura 1 pode-se ver um exemplo simples de componentes em AADL: o componente *s2* estende o componente *s1*, *s1.i* implementa *s1* e *s2.i* implementa *s2*. Pode-se ver que *s2* refina uma *feature* de *s1* e altera um valor de propriedade. Na notação gráfica AADL, implementações são desenhadas com bordas em negrito para facilitar a compreensão dos diagramas.



(a) Visão gráfica

```
1  system s1
2   features
3    p1: in event port;
4    p2: in event data port;
5   properties
6    Load_Time => 5 sec .. 6 sec;
7  end s1;
8
9  system s2 extends s1
10  features
11   p2: refined to in event data port Behavior::Integer;
12  properties
13   Load_Time => 5 sec .. 8 sec;
14 end s2;
15
16 system implementation s2.i
17 end s2.i;
18
19 system implementation s1.i
20 end s1.i;
```

(b) Especificação textual

Figura 1: Relações de extensão e implementação

Os componentes AADL podem ser do tipo *hardware*, *software*, ou híbridos. A única categoria de components híbrida é *system*, que representa um sistema inteiro composto de *hardware* e *software*. O mapeamento de componentes de *software* para componentes de *hardware* é feito através de conexões e propriedades dentro da implementação de um sistema. Um sistema pode conter componentes de todas as categorias, tanto de *hardware* quanto de *software*.

Componentes de *Software* A AADL contém as seguintes categorias de componentes de *software*: *data*, *subprogram*, *thread*, *thread group* e *process*. A representação gráfica destes componentes pode ser vista na figura 2.

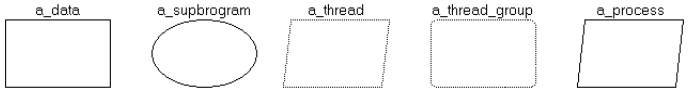


Figura 2: Componentes de *software* AADL

Data Representa todas as estruturas de dados que são transmitidas, armazenadas ou compartilhadas numa especificação AADL. Podem conter outras estruturas de dados

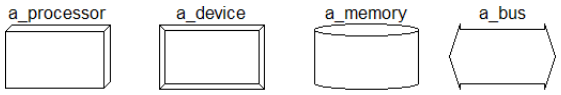


Figura 3: Componentes de *hardware* AADL

como subcomponentes, e sub-rotinas como "features". Neste caso, a semântica AADL é similar à orientação a objetos: o componente *data* é visto como uma classe e as sub-rotinas são vistas como os métodos desta classe, podendo ler ou modificar os dados.

Subprogram Sub-rotinas do *software* do sistema. Interfaces de *subprogram* podem conter parâmetros de entrada e saída (um *parameter* é o equivalente a uma *data port*), assim como *ports* para enviar eventos. As implementações de *subprogram* podem conter e chamar outros *subprogram*.

Thread Uma *thread* representa uma unidade escalonável que executa instruções no espaço virtual de memória ocupado pelo processo (*process*) que a contém. A interface de uma *thread* pode conter *ports*, acesso a dados e sub-rotinas – sub-rotinas declaradas na interface de uma *thread* podem ser chamadas remotamente por *threads* executadas em outros processadores. Uma implementação de *thread* pode conter estruturas de dados e pode realizar chamadas a sub-rotinas.

Thread Group Este componente possui a mesma interface de uma *thread*, e sua implementação pode conter estruturas de dados, *threads* e outros *thread groups* para melhor organizar as especificações AADL.

Process Um *process* é um espaço virtual onde *threads* são executadas. Sua interface é semelhante à das *threads* e sua implementação pode conter estruturas de dados, *threads* e *thread groups*.

Componentes da Plataforma de Execução Os componentes de *hardware* em AADL são: *processor*, *device*, *memory* e *bus*. A figura 3 mostra sua representação gráfica.

Processor Categoria de componente responsável por escalonar e executar *threads* como um processador. Sua interface pode se comunicar com outros componentes através de *ports* e acesso a barramentos (*bus*); sua implementação pode conter subcomponentes da categoria *memory*.

Device Componentes da categoria *device* correspondem a dispositivos que se comunicam com o ambiente físico externo ao sistema. Interfaces de *device* podem conter *ports*,

acesso a *bus* e acesso remoto a sub-rotinas. *Devices* não podem conter subcomponentes.

Memory Unidades de armazenamento que representam um tipo genérico de memória. Comunicam-se com processadores via acesso a barramento comum, ou diretamente se a memória em questão estiver contida no processador. Internamente, pode conter outros componentes do tipo *memory*.

Bus Componentes *bus* são abstrações de *hardware* e *software* responsáveis pela comunicação entre os outros componentes de *hardware*. A interface do componente *bus* pode possuir somente acesso a outros barramentos. Suas implementações não contêm outros subcomponentes, assim, as ligações de um componente *bus* com os outros é feita por meio de declarações de propriedades. Uma visão mais aprofundada deste componente e sua utilidade é vista no capítulo 7.

3.3.2.2 Conectores AADL

Os componentes AADL são ligados por meio de conectores simples, que são declarados diretamente nas implementações dos componentes que os contêm – subcomponentes também são declarados nas implementações de componentes, mas os componentes que os instanciam são declarados independentemente de outros componentes. Estas ligações, físicas ou lógicas, são chamadas *connections* e podem representar conexões de *ports* por onde passam dados e/ou eventos, parâmetros de sub-rotinas ou acesso a dados e barramentos (respectivamente *bus access* e *data access*). As conexões são direcionadas: subcomponentes são ligados na direção *out port - in port* e *provides access - requires access*, conexões de um subcomponente para seu componente de *out port* para *out port* e *provides access* para *provides access*, e conexões de um componente para um subcomponente são realizadas de *in port* para *in port* e *requires access* para *requires access*. A figura 4 ilustra um componente contendo todos os tipos de *ports* encontrados em AADL. A nomeação das conexões é opcional, mas estas só podem ser referenciadas na especificação se tiverem nomes.

Para simplificar a criação das conexões, *ports* podem ser agrupadas em conjuntos chamados *port group*, que são declarados da mesma forma que componentes, porém possuem apenas *features*. Assim, uma única conexão pode ligar vários pontos – uma *feature* pode ser um *port group* normal ou invertido (isto é, todas as *ports* nele contidas são as mesmas, porém em direções opostas). Conexões entre *port groups* de subcomponentes podem ser feitas se ambos contêm instâncias do mesmo *port group*, mas em sentidos opostos, e

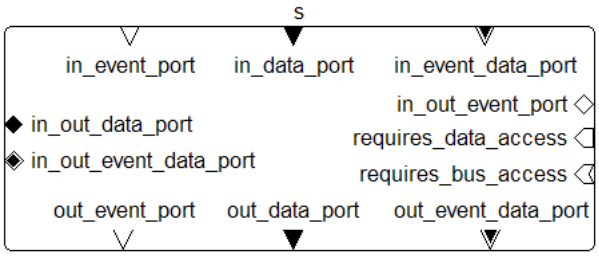
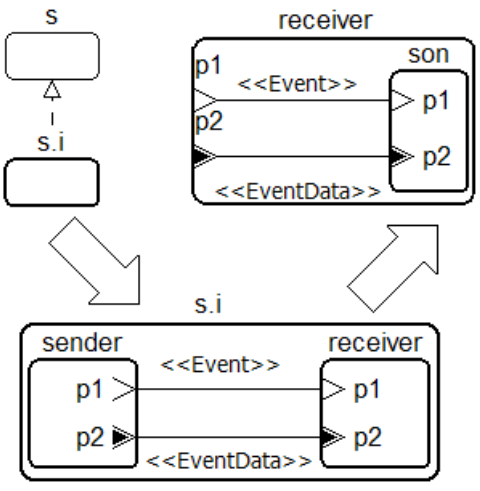


Figura 4: Tipos de *ports* disponíveis em AADL

conexões entre um componente e um subcomponente podem ser feitas se forem entre *port groups* idênticos e de mesmo sentido.

A figura 5 ilustra o modo correto de se conectar componentes: na declaração da implementação *s.i*, os subcomponentes *sender* e *receiver* são ligados por conexões que vão das portas *out* do *sender* para as portas *in* do *receiver*. Dentro do componente que instancia *receiver*, há conexões que ligam suas portas *in* às portas *in* de seu subcomponente.



(a) Visão gráfica

```
1  system implementation s2.i2
2   subcomponents
3    son: system s2.i;
4   connections
5    event port p1 -> son.p1;
6    event data port p2 -> son.p2;
7  end s2.i2;
8
9  system s3
10  features
11   p1: out event port;
12   p2: out event data port behavior::integer;
13 end s3;
14
15 system implementation s3.i
16 end s3.i;
17
18 system s
19 end s;
20
21 system implementation s.i
22  subcomponents
23   sender: system s3.i;
24   receiver: system s2.i2;
25  connections
26   event port sender.p1 -> receiver.p1;
27   event data port sender.p2 -> receiver.p2;
28 end s.i;
```

(b) Especificação Textual

Figura 5: Exemplo de conexões em AADL

A AADL também permite a declaração de *flows*, entidades que não têm um significado concreto e ilustram o fluxo de informações através de componentes e conexões. Em muitas especificações esta noção é interessante para que sejam feitas análises (tais como propagação de erros e qualidade de serviço) ao longo do caminho que os dados percorrem.

As interfaces de componentes podem conter declarações de *flow*: inícios de fluxo (*flow*

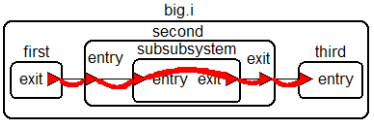
sources) podem ser associados a *ports* de saída, caminhos de fluxo (*flow paths*) podem ser associados a um caminho de uma *port* de entrada a outra de saída, e fins de fluxo (*flow sinks*) podem ser associados a *ports* de entrada.

Nas implementações de componentes cujas interfaces possuem especificações de *flow*, este fluxo pode ser descrito internamente, seja utilizando as mesmas *features* utilizadas na especificação de interface, seja detalhando o fluxo interno, descrevendo a passagem dos dados por subcomponentes (que, por sua vez, devem ter especificações de fluxo em suas interfaces) e conexões. O fluxo completo é declarado na implementação de componente que o contém, sob a forma de um *end to end flow*.

A figura 6 mostra um exemplo de fluxo ao longo de componentes. O fluxo inicia-se no subcomponente *first*, atravessa o subcomponente *second* e termina no subcomponente *third*. Na implementação do fluxo inteiro, só são "vistas" as especificações de fluxo das interfaces destes subcomponentes e as conexões que os ligam; o detalhamento de como o fluxo atravessa o subcomponente *second* é visto na implementação do componente que o instancia. Na figura 6(a), somente a especificação do componente exterior é mostrada, o código completo encontra-se no apêndice A. O caminho percorrido pelo fluxo, considerando todos os níveis hierárquicos, é visto em destaque na imagem.

```
1  system big
2  end big;
3
4  system implementation big.i
5   subcomponents
6    first: system flowsource.i;
7    second: system flowpath.i;
8    third: system flowsink.i;
9   connections
10   source2path: event port first.exit -> second.entry;
11   path2sink: event port second.exit -> third.entry;
12  flows
13   event_passage: end to end flow first.start ->
14   source2path -> second.going -> path2sink ->
15   third.finish;
16 end big.i;
```

(a) Especificação do componente exterior



(b) Ilustração do percurso dos dados

Figura 6: Um exemplo de *flow*

3.3.2.3 Configurações em AADL

O mecanismo elementar para representar configurações de componentes e conectores em AADL é a cláusula de modos de operação (*modes*), utilizável nas implementações de componentes. A representação de modos de operação utiliza estruturas semelhantes a autômatos finitos: um dos modos é declarado como inicial, e a passagem de um modo

para outro é feita quando a condição de guarda (uma *in event port* acionada) é satisfeita. As outras cláusulas vistas em implementações podem ser dependentes do modo de operação, para isto, basta terminar suas declarações com a expressão *in modes* e os modos de operação em que se deseja ativar a cláusula. A figura 7 mostra um pequeno sistema com modos de operação: a implementação de sistema *car.i* ilustra a comunicação do computador de bordo de um carro com a interface vista pelo usuário, através de um barramento. Seus subcomponentes são o computador *onboard_computer*, a interface *user_interface* e os barramentos principal (*main_bus*) e sobressalente (*backup_bus*). No caso da chegada de um evento *failure*, o modo de operação passa de normal para "backup", e, neste novo modo, a chegada de um evento *recovery* leva o sistema de volta ao modo normal. Nas linhas 16-19, pode-se ver que a propriedade *Actual_Connection_Binding*, que descreve o mapeamento de um *hardware* para uma conexão de *software*, tem como valor a referência do barramento principal para o modo normal, e do barramento sobressalente para o modo "backup".

```
1  system implementation car.i
2    subcomponents
3      onboard_computer: system computer;
4      user_interface: system gui;
5      main_bus: bus;
6      backup_bus: bus;
7    connections
8      comp2gui: port group onboard_computer.IO ->
9      user_interface.OI;
10   modes
11     normal: initial mode ;
12     backup: mode;
13     normal -[failure]-> backup;
14     backup -[recovery]-> normal;
15   properties
16     Actual_Connection_Binding => reference main_bus
17     applies to comp2gui in modes (normal);
18     Actual_Connection_Binding => reference backup_bus
19     applies to comp2gui in modes (backup);
20 end car.i;
```

Figura 7: Modos de operação em AADL

3.3.2.4 Propriedades em AADL

Componentes e conexões podem receber características adicionais com a inserção de propriedades nas especificações. A linguagem básica da AADL já possui uma série de propriedades pré-definidas que podem ser utilizadas, devendo-se apenas respeitar as restrições de componentes que são especificadas. Dada a ênfase da AADL em sistemas críticos e de tempo-real, boa parte das propriedades pré-existentes servem para dar características temporais ao sistema, tais como tempos de computação e *deadlines* para tarefas, tempos de propagação em barramentos e freqüência de processadores. Na figura 7, há o uso de

uma propriedade que define um elemento de *hardware* responsável por realizar uma determinada conexão de *software*. Conforme mostrado na figura, após o nome da propriedade, deve ser dado um valor a ela e uma referência ao elemento ao qual a propriedade se aplica, após as palavras reservadas *applies to*. Na omissão desta última parte, presume-se que o valor de propriedade aplica-se ao próprio componente onde ela é declarada. As restrições de aplicação das propriedades são dadas na forma vista na figura 8: seu valor deve ser uma lista de referências a barramentos, dispositivos ou sistemas, e ela pode ser aplicada a conexões de *ports*, sistemas, processos, *threads* ou grupos de *threads*. A palavra reservada *inherit* atesta que se a propriedade for especificada para um componente e não for especificada para algum subcomponente seu mas sua aplicação a ele é sintaticamente possível, o valor dado ao componente será herdado pelo subcomponente.

```
1  Actual_Connection_Binding: inherit list
2   of reference (bus, processor, device)
3   applies to (port connections, thread,
4   thread group, process, system);
```

Figura 8: Descrição de uma propriedade AADL

Com o uso desta sintaxe, usuários podem criar suas próprias propriedades, em especificações globais AADL chamadas *property sets*. Para referenciar propriedades de *property sets* criados pelo usuário, bem como componentes declarados em *packages*, deve-se utilizar o nome da especificação global seguido de "::" e o nome da propriedade ou componente.

3.3.3 Anexos em AADL: o Anexo Comportamental

O Anexo Comportamental da AADL foi criado para suprir a carência de mecanismos adequados para a representação de comportamentos internos dos componentes AADL. Ele inclui um *package* com declarações de tipos de dados para serem usados como as estruturas vistas em outras linguagens (tais como inteiros e booleanos), um *property set* com propriedades úteis na declaração das novas estruturas de dados, e uma sintaxe para ser usada em declarações de anexo de componentes. O Anexo Comportamental utiliza máquinas de estados mais detalhadas que as máquinas vistas nas declarações de modos de operação, oferecendo classificações para os estados (estados *return* que representam possíveis estados de fim de subrotinas, por exemplo), diferentes condições de disparo das transições (chegada de eventos, verificação de variáveis ou disparo de *timeouts*) e ações associadas a transições, como atribuições de valores a variáveis e envio de eventos. Estas máquinas de estados também podem ser usadas para declarar transições de modos de operação, e podem conter hierarquias, com modos de operação compostos e estados

compostos ou concorrentes.

A figura 9 ilustra um componente cujo comportamento interno é descrito pelo anexo. Trata-se de uma *thread* periódica, como visto na linha 3. As *threads* periódicas são disparadas através de um evento implícito na *in event port Dispatch*, presente em todos os componentes da categoria *thread*. Assim, as linhas 12-13 representam a transição do estado *ready* para o estado *calculating* assim que o componente é disparado, e a transição ocasiona uma computação que dura entre 50 e 100 milissegundos. Na linha 14, pode-se ver que o componente pode voltar para o estado *ready* sem precisar de alguma condição.

```
1  thread t1
2  properties
3  Dispatch_Protocol => Periodic;
4  end t1;
5
6  thread implementation t1.i
7   annex behavior_specification {**
8    states
9     ready: initial complete state;
10    calculating: state;
11   transitions
12    ready -[Dispatch?]-> calculating
13    {computation(50 ms,100 ms);};
14    calculating -[]-> ready;
15  **};
16 end t1.i;
```

Figura 9: Exemplo de uso do Anexo Comportamental

3.3.4 Ferramentas para projeto com AADL

As principais ferramentas existentes para o desenvolvimento de projetos com AADL encontram-se disponíveis em [1]. Para a criação de especificações textuais, uma ferramenta bastante usada é o OSATE (*Open-Source AADL Tool Environment*), que consiste numa série de *plug-ins* executados na plataforma Eclipse [2]. As especificações podem ser criadas em texto puro ou em arquivos semelhantes ao formato XML, num editor arborescente. O OSATE contém um *parser* para verificação sintáxica que também realiza algumas verificações semânticas elementares da sintaxe padrão AADL. Usuários do OSATE podem instalar também o *plug-in* opcional OSATE-BA que contém um *parser* para o Anexo Comportamental. No OSATE, sistemas podem ser instanciados, e análises podem ser feitas em componentes e conexões instanciados. A versão mais recente do OSATE possui algumas ferramentas simples para análise:

- Alocação de recursos (memória, capacidade de processamento, largura de banda

[2] `http://www.eclipse.org`

disponível em barramentos), onde os componentes analisados devem possuir propriedades especificando o quanto de memória, processador e largura de banda é fornecido ou consumido por eles.

- Segurança e criticalidade, onde é verificado que componentes não enviam informações a componentes de nível menos estrito de segurança ou mais críticos, evitando que um componente crítico seja comandado por um componente menos crítico ou que informações confidenciais cheguem em componentes menos seguros.

- Análises de conexões exigidas, onde verifica-se que todos os componentes cujas portas devem estar conectadas (o que pode ser especificado através da propriedade *Required_Connection*) realmente possuem estas conexões.

- Latência em fluxos de informação, onde é verificado se a implementação de um fluxo possui uma latência menor ou igual à de sua especificação.

- Inversão de prioridades, onde as *threads* periódicas, mapeadas para o mesmo processador e com prioridades especificadas são analisadas para verificar se há inversão de prioridades.

As especificações AADL também podem ser feitas no editor gráfico ADELE (figura 10, desenvolvido nos projetos TOPCASED e SPICES. O ADELE também permite a geração de código AADL textual a partir das especificações gráficas. Outra ferramenta disponível para edição de especificações AADL é a Ocarina: ela permite a criação de especificações AADL textuais e inclui, além de um *parser*, um gerador de código para Ada [3].

Outros tipos de ferramentas para AADL incluem o simulador ADeS [4], que simula a execução de uma especificação AADL, e *softwares* variados que trabalham com AADL, como a ferramenta de análise de escalonabilidade Cheddar [5] e o *software* Dia [6], utilizado para construção de diagramas.

Neste trabalho, foram utilizados o OSATE 1.5.5, o OSATE-BA 0.5.2 e o editor gráfico TOPCASED 1.0.0 (antecessor do ADELE).

[3] `http://www.adahome.com`
[4] `http://www.axlog.fr/aadl/ades_en.html`
[5] `http://beru.univ-brest.fr/~singhoff/cheddar`
[6] `http://live.gnome.org/Dia`

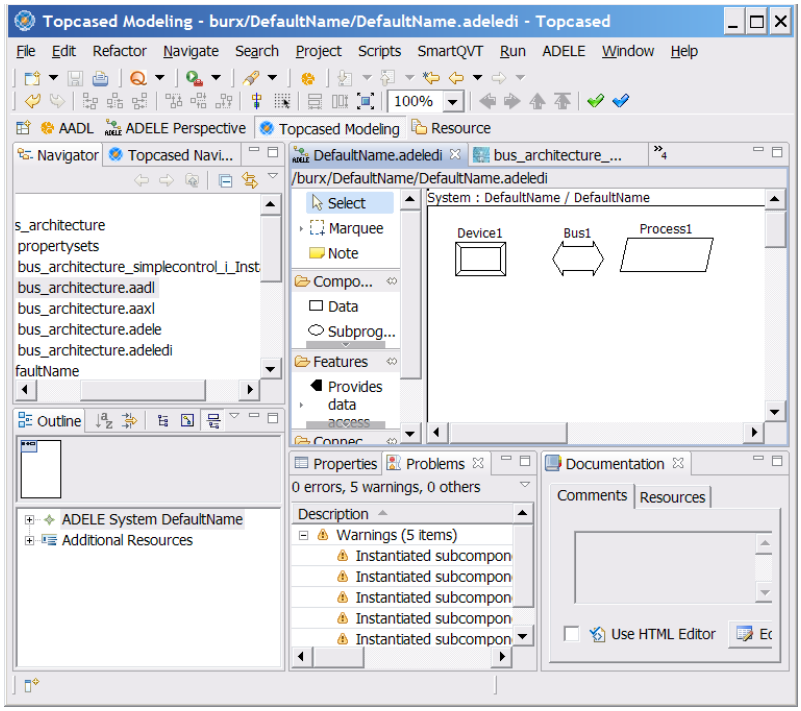


Figura 10: A interface da ferramenta ADELE

3.4 Conclusão

Este capítulo apresentou uma introdução a arquitetura de sistemas, assim como às linguagens de descrição de arquitetura e a AADL.

O paradigma das ADLs visa obter representações úteis da arquitetura dos sistemas, de modo a facilitar o desenvolvimento e integração desses sistemas. As arquiteturas são normalmente retratadas utilizando-se componentes, conectores e configurações como "blocos básicos" de montagem.

A linguagem AADL é uma ADL bastante focada em sistemas embarcados e críticos, possuindo abstrações para representar tanto componentes de *hardware* quanto de *software*, bem como uma série de propriedades que auxiliam na especificação dos aspectos temporais do sistema. Os componentes AADL são extensíveis e reutilizáveis, e a linguagem como um todo é extensível por meio de anexos e *property sets*. As configurações dos sistemas especificados com AADL podem ser expressas através de modos de operação e propriedades de mapeamento de *software* e *hardware*.

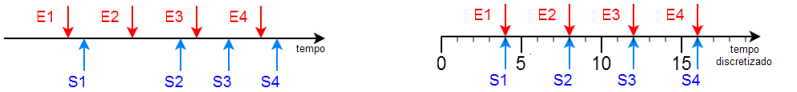
4 Linguagens Síncronas

4.1 Introdução

O conceito de programação síncrona surgiu na década de 1980 com a intenção de solucionar problemas então encontrados na concepção de sistemas embarcados complexos. Tais sistemas são freqüentemente reativos: sua velocidade de interação com o ambiente é determinada pelo próprio ambiente, seu comportamento deve ser determinístico e normalmente há operações que são executadas em paralelo em seus módulos. Assim, a decomposição destes sistemas em subsistemas deve ser feita com cuidado para manter o determinismo, já que a mera separação do sistema em tarefas implica na inclusão de fatores adicionais, como tempo de comunicação, ocasionando um aumento muito grande na dificuldade de prever o comportamento dos sistemas.

As linguagens síncronas de programação visam simplificar a modelagem de sistemas reativos concorrentes com o uso de duas hipóteses sobre seu comportamento temporal: tanto os tempos de comunicação entre os módulos de um sistema, quanto os tempos internos de cálculo dos valores de saída a partir dos valores de entrada são considerados nulos. Esta aproximação não reflete o comportamento real do sistema, mas pode ser aplicada sem problemas se for provado que os tempos de cálculo e comunicação são suficientemente baixos para que possam se manter dentro do intervalo de tempo que existe entre duas reações do sistema.

A figura 11 ilustra o comportamento temporal de sistemas reativos. Num sistema real, a saída $S_n = F(E_n)$ é calculada num tempo que pode sofrer variações, mas que é sempre inferior ao período imposto pelo ambiente. Na especificação síncrona do sistema, a saída é calculada imediatamente e o tempo é discretizado, mas se o cálculo de S_n no sistema real for concluído em um tempo desprezível, os comportamentos podem ser considerados como equivalentes.



(a) Comportamento real

(b) Visão em linguagens síncronas

Figura 11: Comportamento temporal de sistemas reativos

4.2 Linguagens Síncronas de Programação

As linguagens síncronas aplicam na programação o paradigma síncrono, que já era conhecido em outras áreas, como circuitos elétricos (máquinas de estados finitos síncronas) e automação (equações a diferenças, diagramas Ladder). Além dos princípios de sincronismo e discretização do tempo, as linguagens síncronas baseiam-se em memórias e iterações necessariamente finitas, e normalmente elas possuem ferramentas para compilação de seu código em alguma linguagem seqüencial tradicional de programação. As linguagens síncronas de programação podem ser imperativas ou declarativas: as linguagens imperativas (como Esterel [10]) apresentam código seqüencial semelhante ao visto em linguagens tradicionais, com a adição de estruturas que permitem a sincronização dos módulos do programa. Já as linguagens declarativas, como SIGNAL [33] e LUSTRE [13], descrevem os sistemas como conjuntos de equações que representam o fluxo de dados. Programadores tradicionais têm mais facilidade de utilizar as linguagens imperativas, enquanto as linguagens declarativas costumam ser a escolha de profissionais do ramo da automação. Atualmente, as linguagens síncronas têm encontrado aplicações industriais principalmente no projeto de sistemas embarcados de tempo-real [9], graças aos seus modelos matemáticos sólidos e à disponibilidade de ferramentas para projeto e verificação.

O uso de linguagens síncronas para a representação comportamental de protocolos de barramentos de comunicação é interessante para enfatizar a concorrência e o sincronismo vistos nestes protocolos. Dada a utilização destas linguagens em sistemas determinísticos e bem definidos, seu uso neste trabalho restringe-se a representações simplificadas dos protocolos para fácil compreensão e simulação, já que a especificação (e, conseqüentemente, a verificação) parametrizada não é possível neste tipo de linguagem. A linguagem LUSTRE é utilizada neste trabalho, devido ao seu modelo temporal simplificado e à disponibilidade de ferramentas para uso acadêmico.

4.3 A linguagem LUSTRE

LUSTRE é uma linguagem declarativa de programação síncrona, desenvolvida para a programação de sistemas de controle automático e processamento de sinais [13]. Existem ferramentas acadêmicas[1] e comerciais[2] para o desenvolvimento de LUSTRE, com casos bem-sucedidos de sua aplicação na indústria, como sua utilização na empresa Airbus para o desenvolvimento de softwares de calculadores de vôo.

De acordo com a base das linguagens síncronas, LUSTRE é causal e utiliza memória limitada. Assim, o valor de uma variável no instante t só pode depender de seu valor em instantes inferiores a t e de valores de outras variáveis em instantes não superiores a t; e existe um inteiro positivo n tal que nenhuma variável necessita de um número superior a n de valores anteriores de qualquer variável para calcular seu valor no instante seguinte. As variáveis em LUSTRE são vistas como seqüências de valores, com um valor para cada instante de tempo. LUSTRE possui os operadores aritméticos e booleanos elementares, além do condicional "if ... then ... else". Além disso, possui o operador *pre* para acessar valores anteriores de variáveis, o operador de seqüência "->" que permite atribuir valores iniciais a variáveis, o operador *when* que atribui um valor a uma variável se esta respeitar a condição de guarda, e o operador *current* que recupera o valor de uma variável, ou seu último valor definido, se esta não tiver um valor naquele período, o que pode ocorrer se o operador *when* é utilizado para definir seu valor. A semântica temporal de LUSTRE é bastante simples. Um relógio de base é utilizado para representar o tempo, e outros relógios podem ser derivados deste com o uso de variáveis auxiliares e a operação *when*.

A figura 12 mostra um exemplo de programa LUSTRE. Um programa pode conter uma ou mais funções (*nodes*), as quais possuem parâmetros de entrada, saída e variáveis internas. A função *sampler* recebe como entrada um sinal analógico (representado como booleano para simplificar o exemplo) e fornece um sinal amostrado como saída, contendo um contador como variável interna. Na linha 4, o contador é iniciado com o valor 0 e para os outros instantes seu valor passa a ser o seu antecessor acrescido de 1, ou 0 se o seu antecessor for 9. Na linha 5, o sinal amostrado tem o mesmo valor do sinal de entrada quando o contador estiver em 0, e não possui valor em outros instantes, retornando o valor não-definido *nil*. Se a saída desta função for aplicada na função zero_order_hold, o sinal original será reconstruído da mesma forma que um filtro de ordem zero o faz, mantendo o último valor lido pela função *sampler*, através da operação *current* realizada na linha 10.

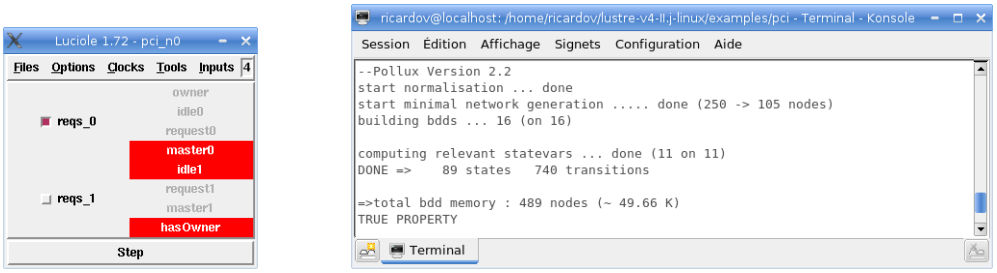
[1] Disponíveis em `http://www-verimag.imag.fr/~synchron/`
[2] SCADE: `http://www.esterel-technologies.com/products/scade-suite/`

```
1  node sampler(analog_signal: bool) returns (sampled_signal: bool)
2   let
3   var counter: int;
4    counter = 0 → if pre(counter) = 9 then 0 else pre(counter) + 1;
5    sampled_signal → analog_signal when counter = 0;
6   tel
7
8  node zero_order_hold (sampled_signal: bool) returns (held_signal: bool)
9   let
10   held_signal = current(sampled_signal);
11  tel
```

Figura 12: Um programa LUSTRE

4.4 Ferramentas LUSTRE

A distribuição acadêmica de LUSTRE possui uma série de ferramentas para compilação, verificação e simulação de seus programas. Seu compilador principal utiliza como entrada um arquivo LUSTRE (.lus) e o transforma num arquivo de código expandido (.ec), onde todos os vetores e recursões do código original são expandidos. A partir dos arquivos .ec, diversas ferramentas da distribuição podem ser usadas. Pode-se convertê-los em arquivos C ou Ada utilizando-se o compilador *ec2oc* para gerar arquivos de código-objeto e em seguida os compiladores *occ* ou *ocada* para compilar os arquivos de código-objeto. Ainda a partir dos arquivos de código expandido, também pode-se utilizar o simulador *luciole*, que é dotado de uma interface gráfica e simula a execução de uma função LUSTRE passo-a-passo. O usuário deve colocar os valores de entrada da função para cada ciclo, e o simulador mostra as saídas correspondentes. Outra ferramenta que faz uso dos arquivos de código expandido é o verificador *lesar*, que é chamado tendo uma função de saída booleana como parâmetro de entrada. Este verificador gera um autômato com todas as possíveis seqüências temporais do sistema e assume que a saída dela corresponde a uma propriedade (ou uma conjunção de propriedades) a ser verificada, assim, se a saída tiver o valor "TRUE" em todas as seqüências, a ferramenta interpretará que a propriedade é verdadeira. Uma limitação no uso desta ferramenta é a obrigação de utilizar somente variáveis booleanas na especificação, pois o autômato construído trabalha somente com valores booleanos. Na figura 13(a) pode-se ver que as variáveis de entrada são dadas pelo usuário, e quando este aciona o botão "step", as saídas são calculadas e exibidas na interface. Na figura 13(b), o funcionamento do verificador *lesar* pode ser visto: ele gera um autômato com as seqüências que podem ocorrer no sistema e utiliza este autômato para a verificação, retornando a resposta "True property" se a propriedade for verdadeira, ou um contra-exemplo se ela for falsa.



(a) O simulador *Luciole* (b) O verificador *Lesar*

Figura 13: Algumas ferramentas da linguagem LUSTRE

4.5 Conclusão

Neste capítulo, foram vistos os conceitos e aplicações de programação síncrona, assim como as características e os elementos básicos de uma linguagem de programação síncrona. A linguagem LUSTRE foi apresentada com mais detalhes.

A programação síncrona tem se mostrado uma alternativa viável para a modelagem de sistemas embarcados reativos complexos, já que, se estes puderem ter seu comportamento aproximado por tempos de cálculo e comunicação nulos, suas especificações podem ser bastante simplificadas e modularizadas com o uso de linguagens síncronas. As considerações necessárias para o uso das linguagens síncronas são válidas em muitos casos, o que é atestado pelas aplicações bem-sucedidas destas linguagens no meio industrial. Sistemas modelados com linguagens síncronas são determinísticos, concorrentes e podem ser analisados pelas ferramentas de verificação que cada linguagem possui.

A escolha da linguagem LUSTRE para ilustrar o comportamento de protocolos de barramento deve-se à sua simplicidade, já que tais protocolos não fazem uso de mecanismos temporais complexos. A utilização de linguagens síncronas para este estudo não resolve o problema de verificação parametrizada, mas auxilia na compreensão do funcionamento dos protocolos síncronos graças à simplicidade de representação e simulação destes protocolos em alto nível.

5 O Método Event-B

5.1 Introdução

O método Event-B é uma variação do método B, criado por Jean-Raymond Abrial [3] com o intuito de ser uma alternativa para o desenvolvimento das partes principais do ciclo de vida de um software. O método B permite a especificação de sistemas através de sucessivos refinamentos, obtendo-se uma arquitetura em diversos níveis que usualmente culminam numa especificação concreta do sistema, que pode então ser utilizada na geração de código em linguagens tradicionais de programação. Este código gerado tem a vantagem de já ter sido validado durante a especificação do sistema em B, já que uma especificação B válida possui provas matemáticas de consistência, tipagem e semântica. Cada refinamento da especificação respeita as propriedades de suas abstrações, desta forma, as implementações do sistema desempenham as funções descritas nas especificações abstratas.

O desenvolvimento incremental e a validação rigorosa permitidos pelo método B facilitaram sua difusão na indústria [16]. Entre os casos mais conhecidos de sua utilização estão o desenvolvimento da linha 14 do transporte metroviário de Paris, a concepção de sistemas automáticos de proteção em trens da companhia francesa de estradas de ferro SNCF, e o projeto de sistemas de direção automática para veículos do aeroporto de Paris-Roissy.

O sucesso da utilização do método B no desenvolvimento de softwares levou à criação de variações no método original para possibilitar seu uso em áreas diferentes. Alguns pesquisadores, incluindo o próprio Abrial, desenvolveram uma variação chamada Event-B [30] para que sistemas em geral (tais como sistemas embarcados) também possam ser concebidos em B.

5.2 Desenvolvimento Seguro de Sistemas

O método B (e suas derivações) foi batizado com o termo "método" por abranger todo um caminho de construção de softwares e sistemas em geral. Ele possui uma linguagem (conhecida como linguagem B) para especificação de sistemas, utiliza uma noção de refinamento que permite um desenvolvimento passo-a-passo e é dotado de uma semântica de validação formal que assegura a correção das especificações. O método também compreende ferramentas associadas que simplificam a especificação e a validação. No entanto, ao contrário do que se supõe da utilização do termo "método", não há um modo rígido de utilização de B no desenvolvimento de sistemas [15].

Apesar da ausência de um procedimento padronizado para desenvolver softwares, existem alguns passos fundamentais para a utilização do método B:

Especificação e validação abstrata Em um primeiro momento, deve-se especificar as funcionalidades e propriedades fundamentais do software utilizando a linguagem B. Neste nível, o comportamento esperado, não necessariamente determinístico, é descrito e as propriedades especificadas são verificadas para que a visão abstrata do software seja validada.

Refinamento da especificação abstrata Este passo consiste em gerar uma nova especificação a partir da anterior, introduzindo aspectos mais concretos ao modelo abstrato. Para que um refinamento seja válido, ele deve respeitar as propriedades do modelo que ele refina, além de desempenhar as mesmas funcionalidades. Este passo pode ser realizado várias vezes no desenvolvimento de um sistema, gerando vários níveis de refinamento.

Obtenção de um modelo de implementação O último nível de refinamento no desenvolvimento de um software é chamado de *implementação*. Ele deve corresponder ao nível de abstração visto em linguagens tradicionais de programação, permitindo assim a tradução da especificação B para uma linguagem tradicional de programação. Esta especificação deve ser determinística e as estruturas de dados nela usadas devem estar presentes na linguagem de programação na qual deseja-se realizar a tradução.

Nota-se que existem provas matemáticas necessárias em todos os passos. Em especificações mais complexas, realizar todas estas provas manualmente pode ser uma tarefa

demorada. Portanto, a utilização de ferramentas automáticas de geração e prova de teoremas é de fundamental importância para projetos utilizando o método B.

5.3 Fundamentos do Método B

5.3.1 Tipos de dados e notações

A linguagem utilizada no método B apóia-se na teoria dos conjuntos e na lógica de primeira ordem; os tipos de dados são vistos como conjuntos aos quais as variáveis pertencem. Os conjuntos pré-definidos no método B original definem os tipos booleanos, naturais, inteiros, e suas variantes finitas, que representam os números utilizáveis em máquina. Implementações devem obrigatoriamente utilizar os conjuntos matemáticos finitos, eliminando assim a possibilidade de *overflow*. Também há os conjuntos *CHAR* e *STRING*, onde o primeiro representa um conjunto de 255 caracteres e o segundo representa o conjunto de todas as seqüências de caracteres. Desenvolvedores podem criar seus próprios tipos, enumerando seus elementos num conjunto finito. Os conjuntos numéricos podem ter seus elementos comparados através dos comparadores matemáticos tradicionais, e modificados através de operações matemáticas triviais, tais como soma, multiplicação, sucessor e módulo de divisão. O método B também possui seqüências, formadas através de funções que mapeiam inteiros de 1 a n (onde n é o comprimento da seqüência) para elementos de um conjunto qualquer.

Os conjuntos em B podem ser construídos como conjuntos simples ou como produtos e conjuntos das partes de outros conjuntos. Os predicados de pertinência e inclusão são disponíveis para os conjuntos, assim como os operadores de união, intersecção e diferença. A união e a intersecção podem ser generalizadas ou quantificadas, sendo assim aplicadas em mais de dois conjuntos. Relações (conjuntos de pares) também podem ser utilizadas para a construção de conjuntos, possuindo numerosas operações. Se as relações formam funções, podem ser descritas diretamente como funções parciais, totais, sobrejetoras, injetoras ou bijetoras.

Os predicados do método B são construídos a partir dos operadores e quantificadores lógicos existentes na lógica de primeira ordem e mostrados na tabela 1.

Símbolo	Significado
$\neg$	Negação
$\vee$	Disjunção “ou”
$\wedge$	Conjunção “e”
$\Leftarrow$	Implicação
$\Leftrightarrow$	Equivalência
$\exists$	Quantificador Existencial
$\forall$	Quantificador Universal

Tabela 1: Operadores e quantificadores lógicos

5.3.2 Máquinas e Refinamentos

As máquinas são os componentes onde são escritas as especificações B. No método B clássico, as máquinas são divididas em três tipos: abstratas (que representam os sistemas em alto nível), refinamentos (que detalham a visão destes sistemas) e implementações, que fornecem as especificações concretas.

Uma máquina é composta de uma parte estática e uma parte dinâmica. Na parte estática, definem-se todas as estruturas que formam um estado do sistema representado, e na parte dinâmica, são descritos os mecanismos responsáveis pelas mudanças de estado.

Os principais elementos da parte estática de uma máquina são:

SETS Na cláusula *SETS* são especificados os conjuntos definidos pelo usuário. Estes podem ser abstratos ou enumerados.

CONSTANTS Lista de nomes das constantes da máquina.

PROPERTIES Axiomas referentes às constantes e conjuntos especificados nas cláusulas anteriores.

VARIABLES Esta cláusula contém os nomes das variáveis de estado da máquina.

INVARIANT Um predicado que especifica os tipos das variáveis e outras restrições destas. Normalmente, propriedades dinâmicas do sistema são expressas neste invariante.

A parte dinâmica de uma máquina é composta pelas operações que ela pode efetuar. Uma operação consiste num conjunto de substituições (atribuição de valores a variáveis da máquina), e, se necessário, no retorno de um parâmetro de saída. Operações podem ter

parâmetros de entrada e pré-condições de execução. Toda máquina possui uma operação pré-definida de inicialização das variáveis, que não possui parâmetros nem pré-condição.

Uma substituição pode ser descrita através de um predicado "antes-depois", que relaciona os valores das variáveis de estado antes e depois da substituição. A substituição $x := x+2$, por exemplo, tem como predicado antes-depois a expressão $x' = x+2$. A forma geral deste predicado, e de qualquer outro que envolva as variáveis x e x', é $Q(x, x')$.

A validação de uma máquina consiste na verificação dos seu invariante quando uma operação é executada: assume-se que o estado do sistema antes da execução da operação respeita o invariante, e deve-se provar que se a pré-condição for satisfeita, a execução da operação não levará a um estado onde o invariante seja invalidado. A verificação de invariantes é feita através da técnica de "pré-condição mais fraca": a substituição vista na operação é inserida no predicado do invariante e a expressão resultante é a condição necessária e suficiente para assegurar que a substituição não invalida o invariante. A equação 5.1 mostra a utilização desta técnica para a substituição $x := x + 2$ e o invariante $(x \geq 0)$. Este cálculo é feito automaticamente pelas ferramentas de provas de teoremas, que tentam realizar as provas procurando hipóteses que confirmem o predicado encontrado, neste caso, $x \geq -2$.

$$\begin{aligned} [x := x + 2](x \geq 0) \\ x + 2 \geq 0 \\ x \geq -2 \end{aligned} \tag{5.1}$$

No contexto deste trabalho, é importante ressaltar que as provas podem naturalmente conter referências a conjuntos abstratos, permitindo assim a realização de verificações parametrizadas sem a necessidade de esforços adicionais.

Numa definição informal, uma máquina refina outra se a utilização da máquina refinada no lugar da original for imperceptível ao usuário exterior. Os refinamentos são transitivos, assim, numa cadeia de refinamentos de máquinas, basta validar os refinamentos entre níveis adjacentes para provar que todos os níveis mais baixos refinam os níveis mais altos. Para que uma máquina refine outra, todas as suas operações devem ser refinadas. No refinamento, o espaço de variáveis pode mudar, desde que exista uma correspondência entre valores de variáveis refinadas e das variáveis originais. Do mesmo modo, uma substituição refina outra se, para cada atribuição de variável existente na

operação refinada, existir uma correspondência na operação original. Como exemplo, pode-se pensar numa operação que contenha a substituição $x :\in BOOL$ (a variável x assume um valor aleatório do conjunto de booleanos) e é refinada por uma operação que contenha a substituição $x := FALSE$ (x assume o valor booleano FALSE). Neste caso, o refinamento fez com que a substituição se torne determinística, o que é esperado, visto que os refinamentos vão na direção de obter uma especificação concreta.

5.4 B para sistemas

O método B orientado a eventos, conhecido como Event-B ou System-B, é uma adaptação do método B para sistemas a eventos discretos em geral, já que o método B original foi criado para o desenvolvimento de sistemas de software. Para tornar o método mais adequado para a concepção de sistemas, algumas mudanças foram feitas nas estruturas de máquinas vistas em B. Além disso, o próprio método Event-B apresenta variações de acordo com as ferramentas escolhidas para o desenvolvimento dos sistemas. Neste trabalho, o método Event-B é apresentado do modo como é utilizado com a ferramenta Rodin [41].

O desenvolvimento de sistemas em Event-B é bastante similar à proposta original do método B, com a diferença de que seu objetivo não costuma ser a geração de código em linguagens de programação. Com isso, as exigências de determinismo e variáveis similares a variáveis de programação são dispensadas em especificações Event-B.

5.4.1 Contextos Event-B

Em Event-B, as declarações de conjuntos, constantes, axiomas e propriedades referentes a estes foram separadas das máquinas e passam a fazer parte de um contexto. Contextos contêm as partes constantes dos sistemas, possuindo as seguintes cláusulas:

SETS Declarações dos nomes de conjuntos utilizados.

CONSTANTS Lista de nomes das constantes da máquina. Elementos de conjuntos enumerados são declarados como constantes.

AXIOMS Axiomas referentes às constantes e conjuntos especificados nas cláusulas anteriores. Cada constante deve ter seu tipo declarado, e conjuntos enumerados têm seus elementos declarados nesta cláusula. Axiomas que relacionam constantes e conjuntos também são declaradas aqui.

THEOREMS Teoremas são declarados para exprimir propriedades que devem ser provadas por meio dos axiomas.

CONTEXT synchronous_c0

SETS

$CONTROLLERS$
CTR_STATES

AXIOMS

$axm1 : CONTROLLERS \rightarrow CTR\_STATES \neq \varnothing$

END

Figura 14: Exemplo de contexto em Event-B

A figura 14 mostra um contexto que contém dois conjuntos e um axioma. Visto que não há nenhuma declaração de elementos para estes conjuntos, ambos são abstratos.

Event-B é um pouco mais restrito que o método B tradicional no tocante a tipos e estruturas de dados. Os tipos *CHAR* e *STRING*, assim como as seqüências, não estão pré-definidos e devem ser declarados manualmente se forem necessários. Abrial [3] descreve em seu livro a representação matemática de diversas estruturas de dados. Ainda assim, um grande número de operadores está presente na linguagem de Event-B. A tabela 2 mostra alguns operadores e símbolos utilizados neste trabalho.

Símbolo	Significado
$\leftrightarrow$	Relação
$\nrightarrow$	Função parcial
$\rightarrow$	Função total
;	Composição de funções
$\mapsto$	Mapeamento (geração de tuplas)
$\lhd\!\!+$	*Overriding* de uma relação
$\lhd$	Restrição sobre o domínio

Tabela 2: Alguns dos símbolos usados em Event-B

5.4.2 Máquinas Event-B

As máquinas de Event-B possuem variáveis e um invariante como uma máquina B tradicional e fazem referência a contextos através de uma cláusula chamada *SEES*. O invariante em Event-B é descrito na cláusula *INVARIANTS*: o invariante normalmente é dividido em vários invariantes ligados por conjunções, para facilitar a leitura e a validação

das partes separadas. As máquinas possuem uma cláusula *THEOREMS* onde teoremas podem ser declarados e provados com o uso dos invariantes. O uso dos teoremas tem como intenção facilitar a leitura, separando as provas intermediárias das provas realmente desejadas.

A diferença fundamental de uma máquina Event-B para uma máquina B tradicional está na parte dinâmica. Ao invés de operações, há a noção de eventos, compostos por substituições sem pré-condições ou parâmetros de entrada e saída. Esta representação simplificada é mais adequada para especificar sistemas reativos em geral, por outro lado, perde-se poder de expressão em modelos de software. Existem três tipos de substituições em Event-B:

BEGIN...END Substituição para eventos sem guarda, é composta somente de atribuições de valores a variáveis. Um exemplo de sua utilização é no evento de inicialização.

WHEN...THEN...END Substituição para eventos cuja guarda utiliza as variáveis da máquina.

ANY...WHERE...THEN...END Substituição para eventos cuja guarda utiliza variáveis locais. Se as variáveis locais (declaradas após o termo *ANY*) puderem ser instanciadas de modo a satisfazer o predicado dado após o termo *WHERE*, o evento pode ser disparado.

Estas substituições podem atribuir valores precisos ou aleatórios a variáveis, conforme o exposto na tabela 3. A figura 15 mostra exemplos de eventos com os três tipos de guardas. O evento *INITIALISATION* simplesmente atribui valores a variáveis, sem verificação de guarda. O evento *generic arbiter action* deve ser ativado de acordo com o valor de uma variável específica da máquina, portanto possui uma guarda do tipo *WHEN*. O evento *generic controller action* não é disparado sobre uma variável específica, assim, a guarda *ANY...WHERE* é utilizada para selecionar quaisquer variáveis C e ns que atendam às condições especificadas.

Expressão	Significado
$x := y$	x assume o valor y
$x :\in S$	x assume o valor de um elemento aleatório do conjunto S
$x : \|\| P$	x assume um valor que satisfaz o predicado P

Tabela 3: Atribuições de valores em Event-B

EVENTS

INITIALISATION

BEGIN
$ini\_fa : finishedArb := FALSE$
$ini\_state : stateCtr := \varnothing$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_prearbbusy : prearbbusy := FALSE$
$ini\_pgstate : pregstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
$ini\_gstate : gstateCtr := \varnothing$
END

EVENT generic_arbiter_action
REFINES generic_arbiter_action
WHEN
$arb\_ready : finishedArb = FALSE$
THEN
$arb\_done : finishedArb := TRUE$
$update\_status : arbbusy :\in BOOL$
END

EVENT generic_controller_action

ANY
C
ns
WHERE
$some\_ctrls : C \in \mathbb{P}(CONTROLLERS)$
$didnt\_act : C \cap dom(stateCtr) = \varnothing$
$new\_states : ns \in C \rightarrow CTR\_STATES$
THEN
$statechange : stateCtr := stateCtr \cup ns$
END
END

Figura 15: Exemplos de eventos em Event-B

Para validar um evento, deve-se provar que ele é factível (se sua guarda for satisfeita, ele pode ser disparado a partir de qualquer estado do sistema) e que ele respeita o invariante especificado. Seja M uma máquina com um conjunto v de variáveis, que faz referência a um contexto C que possui um conjunto s de conjuntos e um conjunto c de constantes. Assume-se que há um conjunto $P(s, c)$ de axiomas e um invariante $I(s, c, v)$ que regem estas variáveis e constantes. Seja E um evento de guarda $G(s, c, v)$ e predicado antes-depois $R(s, c, v, v')$. Para que o evento E seja factível, deve-se provar que v' existirá para estes invariantes, axiomas, variáveis, conjuntos e constantes. Intuitivamente, esta condição de factibilidade pode ser vista como a necessidade de que o disparo deste evento sempre leve a um estado possível no sistema. Formalmente, tem-se o predicado da equação 5.2, que será mencionado neste trabalho como a condição *FIS* (do inglês *feasibility*).

O passo seguinte de prova é verificar que o invariante é mantido: assumindo que a condição *FIS* é satisfeita, deve-se provar que o invariante se mantém válido para o novo valor de v. O predicado que representa a satisfação do invariante pode ser visto na equação 5.3. Esta condição de respeito ao invariante será mencionada como a condição *INV*.

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists v' \cdot R(s,c,v,v') \tag{5.2}$$

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \wedge R(s,c,v,v') \Rightarrow I(s,c,v') \tag{5.3}$$

Para maior compreensão destas equações, elas serão usadas para validar a máquina da figura 16. Visto que ela não utiliza um contexto, a condição *FIS* para o evento *evt1* torna-se $I(v) \wedge G(v) \Rightarrow \exists v' \cdot R(v,v')$. Aplicando-se o invariante da máquina e a guarda e o predicado antes-depois de *evt1*, tem-se o predicado

$$x \in BOOL \wedge y \in BOOL \wedge \neg(\neg x \wedge y) \wedge (x \wedge \neg y) \Rightarrow \exists x', y' \cdot (x' \wedge y')$$

que é trivialmente verdadeiro devido à tipagem das variáveis. A condição *INV* gera o predicado $I(v) \wedge G(v) \wedge R(v,v') \Rightarrow I(v')$, que neste caso é escrito como

$$x \in BOOL \wedge y \in BOOL \wedge \neg(\neg x \wedge y) \wedge (x \wedge \neg y) \wedge (x' \wedge y') \Rightarrow \neg(\neg x' \wedge y)$$

e pode ser facilmente verificado como verdadeiro. Deve-se notar que, embora o evento não indique explicitamente que $x' := TRUE$, esta ação pode ser deduzida, pois o valor de x é dado na guarda e não é modificado na ação.

5.4.3 Refinamentos em Event-B

Em Event-B, as máquinas podem ser refinadas e os contextos podem ser estendidos. Uma extensão de contexto preserva todos os elementos do contexto original e pode adicionar novos. O refinamento de máquinas é mais complexo: assume-se em um primeiro momento que os conjuntos de variáveis da máquina abstrata e da máquina refinada são disjuntos, e a correlação entre elas é feita através de um invariante chamado "invariante de colagem". Invariantes de colagem podem ser feitos somente entre níveis adjacentes de refinamento. Na plataforma Rodin, os invariantes de colagem entre variáveis que são refinadas para variáveis com o mesmo nome e tipo são implícitos. No caso de mudanças de variáveis, devem ser declarados predicados chamados *witnesses* nos refinamentos de eventos que utilizam variáveis que foram modificadas durante os refinamentos. Tais predicados representam a relação entre as variáveis dos dois níveis e devem ser condizentes com o invariante de colagem. Mais detalhes sobre *witnesses* são vistos nos estudos de caso do capítulo 8.

MACHINE m0

VARIABLES

x
y

INVARIANTS

$inv1 : x \in BOOL$
$inv2 : y \in BOOL$
$inv3 : \neg(x = FALSE \wedge y = TRUE)$

EVENTS

INITIALISATION

BEGIN
$act1 : x := TRUE$
$act2 : y := FALSE$
END

EVENT evt1
WHEN
$grd1 : x = TRUE$
$grd2 : y = FALSE$
THEN
$act1 : y := TRUE$
END

END

Figura 16: Um exemplo de máquina para verificação

Todos os eventos da máquina abstrata devem ser refinados. Seja M a mesma máquina especificada nas equações 5.2 e 5.3, e seja N um refinamento de M, com um conjunto w de variáveis e um invariante de colagem $J(s, c, v, w)$. N deve possuir no mínimo um evento F que refina o evento E da máquina M, e F possui uma guarda $H(s, c, w)$ e um predicado antes-depois $S(s, c, w, w')$. Para que o evento F seja válido, três condições devem ser satisfeitas:

- F deve ser factível. Do mesmo modo que é provada a factibilidade de E, deve-se verificar que no novo conjunto de variáveis também existirá um conjunto de valores que satisfaça o predicado antes-depois do evento. Esta condição é vista na equação 5.4.

- A guarda de F não pode ser menos restritiva que a guarda de E. Em outras palavras, não pode existir um conjunto de valores de variáveis que satisfaçam a guarda de F e que, quando aplicados na máquina M através do invariante de colagem, não satisfaçam a guarda de E. Esta condição é vista na equação 5.5.

- Se os eventos E e F forem executados sobre conjuntos de valores equivalentes do ponto de vista do invariante de colagem, os novos conjuntos de valores também

devem apresentar esta equivalência. Esta condição é vista na equação 5.6.

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w) \Rightarrow \exists w' \cdot S(s,c,w,w') \tag{5.4}$$

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w) \Rightarrow G(s,c,v) \tag{5.5}$$

$$\begin{aligned} P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w) \wedge S(s,c,w,w') \Rightarrow \\ \exists v.' \cdot (R(s,c,v,v') \wedge J(s,c,v',w')) \end{aligned} \tag{5.6}$$

Um evento pode refinar mais de um evento, desde que os eventos abstratos tenham as mesmas substituições e a guarda do evento refinado seja a disjunção das guardas dos eventos abstratos. Novos eventos podem ser adicionados em refinamentos de máquinas. Para isso, deve-se provar que os novos eventos refinam o evento implícito *skip* e que eles não divergem, isto é, não são disparados indefinidamente sem que algum evento original da máquina abstrata seja disparado. O evento *skip* não possui guarda e não altera nenhum valor de variáveis, assim, a equação 5.5 torna-se sempre verdadeira e a implicação da equação 5.6 é simplificada, já que $R(s,c,v,v')$ é verdadeiro quando $v' = v$.

A não-divergência é atestada se todos os eventos novos modificam alguma expressão de modo que ela sempre convirja para um valor em que somente eventos abstratos possam ser executados. Tal expressão é chamada de variante, e costuma ser descrita através de uma expressão numérica natural que tem seu valor decrementado por todos os eventos novos: deve-se provar que a expressão nunca deixará de ter um valor natural, indicando que um número finito de eventos novos é disparado e em seguida o sistema não poderá mais avançar por meio de eventos novos até que um evento original da máquina abstrata seja disparado.

Para compreensão destas condições, o exemplo da figura 16 será retomado. Um possível refinamento dele é visto no contexto da figura 17 e na máquina da figura 18. O contexto *c0* contém dois conjuntos enumerados de dois elementos, *DOOR_STATES* e *CAR_STATES*. A máquina *m1* possui três variáveis, e, observando-se seus invariantes, nota-se que as variáveis *door* e *car* fazem parte destes dois respectivos conjuntos e possuem correspondências com as variáveis x e y de *m0*. Para todos estes eventos, é trivial verificar a condição *FIS*, já que todas as variáveis recebem atribuições de valores per-

tencentes a seus tipos. A verificação formal da condição do refinamento da guarda para o evento *evt1* é mais trabalhosa, mas intuitivamente pode-se observar que a guarda do evento refinado é equivalente à do evento abstrato, apenas com valores e variáveis renomeados. Esta observação também permite que se verifique a validade do invariante de colagem.

Por fim, resta o novo evento *evt2* para verificar. A condição de refinar o evento *skip* pode ser vista na prática com a não-alteração de variáveis que já existiam no modelo abstrato: visto que *skip* não altera nenhum valor de variável na especificação abstrata, *evt2* também não o pode fazer. Como somente uma variável nova é alterada, esta condição é satisfeita. A condição de não-divergência pode ser observada neste caso através de uma comparação entre a guarda e a ação do evento, já que ele é o único novo evento. Visto que sua ação causa a não-satisfação da guarda (impedindo assim sua execução) é trivial concluir que este evento não pode ser executado duas vezes consecutivas. Para simplificar a verificação desta especificação, todas as provas foram feitas automaticamente no ambiente Rodin. Em sistemas de grande porte, onde as condições de verificação formam expressões complexas, a observação da especificação não é tão simples quanto nestes exemplos e as provas manuais podem exigir um longo tempo de resolução.

CONTEXT c0

SETS

$DOOR\_STATES$
CAR_STATES

CONSTANTS

$open$
$closed$
$moving$
$stopped$

AXIOMS

$axm1 : DOOR\_STATES = \{open, closed\}$
$axm2 : open \neq closed$
$axm3 : CAR\_STATES = \{moving, stopped\}$
$axm4 : moving \neq stopped$

END

Figura 17: O contexto utilizado para refinar a máquina *m0*

5.4.4 O ambiente Rodin

A plataforma de desenvolvimento *open-source* Rodin começou a ser desenvolvida em 2004 e consiste numa série de *plug-ins* para a plataforma Eclipse. Para a criação das

MACHINE m1

REFINES m0

SEES c0

VARIABLES

$door$
car
$lights$

INVARIANTS

$inv1 : door \in DOOR\_STATES$
$inv2 : car \in CAR\_STATES$
$inv3 : (door = closed \Leftrightarrow x = TRUE) \wedge (door = open \Leftrightarrow x = FALSE)$
$inv4 : (car = moving \Leftrightarrow y = TRUE) \wedge (car = stopped \Leftrightarrow y = FALSE)$
$inv5 : lights \in BOOL$

EVENTS

INITIALISATION

BEGIN
$act1 : door := closed$
$act2 : car := stopped$
$act3 : lights :\in BOOL$
END

EVENT evt1
REFINES evt1
WHEN
$grd1 : door = closed$
$grd2 : car = stopped$
THEN
$act1 : car := moving$
END

EVENT evt2
WHEN
$grd1 : car = moving$
$grd2 : lights = FALSE$
THEN
$act1 : lights := TRUE$
END

END

Figura 18: O refinamento da máquina *m0*

especificações em Event-B, a plataforma possui alguns assistentes que facilitam a criação de conjuntos enumerados, eventos e variáveis, além da funcionalidade de nomear todos os predicados da especificação, tais como axiomas, teoremas e guardas de eventos. A nomeação é útil para facilitar a referência a estes predicados durante o processo de validação da especificação. Outra facilidade disponível para a especificação de máquinas é a cópia automática do código de uma máquina abstrata para seu refinamento, já que o editor não permite a cópia de blocos de linhas de uma especificação para outra. Após o término da especificação, um *parser* verifica a sintaxe da especificação.

Após a verificação sintática, o próprio ambiente gera as obrigações de prova necessárias

para validar a especificação, de acordo com as equações vistas anteriormente. Diversos provadores de teoremas estão disponíveis no ambiente Rodin, e um deles (ML) é lançado automaticamente para tentar resolver as obrigações de prova. As provas que não forem concluídas serão sinalizadas na lista gerada pelo ambiente, e o usuário poderá utilizar outros provadores automáticos ou tentar finalizar a prova manualmente. Um *plug-in* adicional permite a exportação de especificações de máquinas e contextos para o formato *tex*. A interface do ambiente Rodin pode ser vista na figura 19.

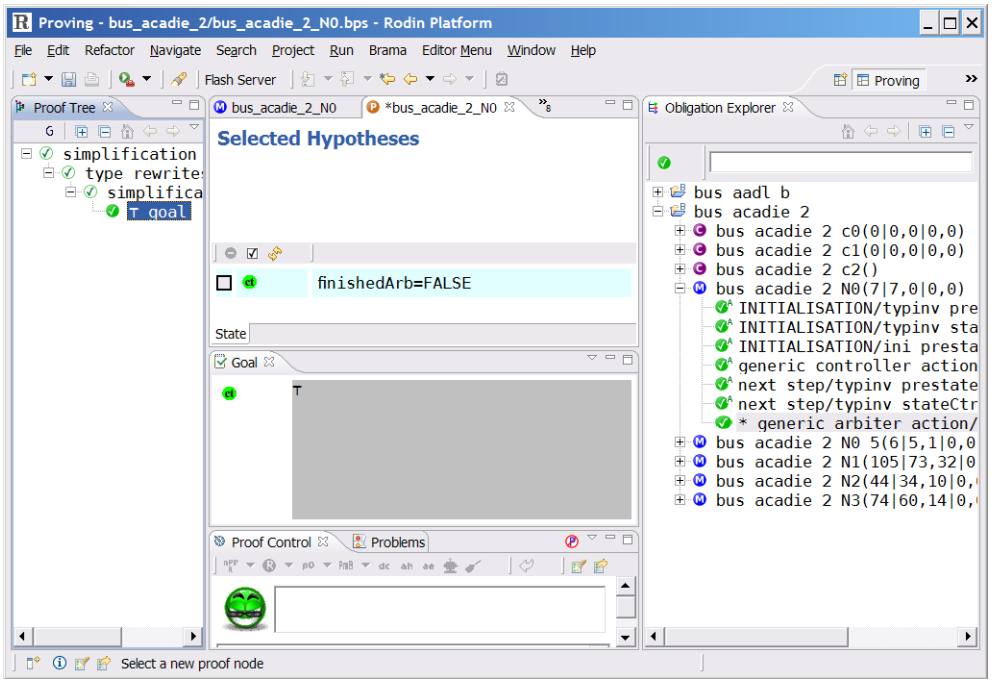


Figura 19: O ambiente de desenvolvimento Rodin

5.5 Conclusão

Este capítulo apresentou o método B e sua variante Event-B, utilizada neste trabalho. O método B é uma abordagem formal e rigorosa para o desenvolvimento de softwares, utilizando uma sólida base matemática e uma metodologia incremental de especificação. Os tipos de dados e as propriedades que se deseja provar são especificados com a utilização de um invariante, normalmente formado por uma conjunção de invariantes. Softwares desenvolvidos em B costumam ter uma máquina abstrata, vários refinamentos desta máquina e uma implementação que é utilizada para gerar código devidamente verificado pelo método.

O método Event-B caracteriza-se por perder a ênfase no desenvolvimento de sistemas de software até a geração de código, optando por uma representação mais simples e genérica de sistemas orientados a eventos. A verificação de uma especificação Event-

B ocorre em diversos níveis: primeiramente verifica-se a correta formação (expressões sintaxicamente corretas, variáveis corretamente inicializadas) e a tipagem de todos os predicados e variáveis, em seguida deve-se provar que todos os eventos abstratos são factíveis e respeitam o invariante. Nos refinamentos, deve-se verificar que os invariantes de colagem são respeitados, todos os eventos abstratos são refinados e todos os novos eventos refinam o evento implícito *skip* e não divergem. O ambiente de desenvolvimento Rodin é útil para a geração e resolução automática de todas as obrigações de prova necessárias para uma especificação, reduzindo o tempo necessário para validar as especificações Event-B.

Parte III

Abordagem Utilizada

6 *Introdução aos Estudos de Caso*

Este capítulo apresenta estudos de caso realizados com o intuito de criar uma abordagem de especificação e verificação de protocolos síncronos de barramento. Conforme o exposto no capítulo 2, o estudo é dividido em modelagem e verificação, tanto de arquitetura quanto de comportamento. O protocolo PCI é utilizado como exemplo para ambas as partes, e o protocolo AMBA AHB é utilizado na parte de comportamento. A utilização de apenas um dos protocolos para a modelagem de arquitetura deve-se à similaridade de representação dos diferentes barramentos numa arquitetura de sistema embarcado.

A modelagem de arquitetura, exposta no capítulo 7, é feita com AADL em dois níveis distintos de abstração: no nível *aadlpci0*, o barramento é inserido num sistema completo e suas características são dadas pelas propriedades disponíveis na linguagem, enquanto no nível *aadlpci1*, mais baixo, o componente AADL *system* é utilizado para representar cada elemento do barramento, e assim permitir uma representação mais detalhada de sua arquitetura. Os modelos AADL criados são baseados nas especificações de protocolos feitas pelo autor [22].

A utilização das propriedades AADL levou a uma contribuição adicional deste trabalho: visto que suas propriedades nem sempre têm sua semântica completamente verificada, foram criadas especificações em Event-B com o objetivo de modelar e verificar estas propriedades. A criação de um *framework* completo para representação da semântica AADL não faz parte do escopo deste trabalho, por esta razão foi criada uma especificação simplificada para descrever as partes essenciais da AADL para a representação da semântica de suas propriedades. A especificação é feita em quatro contextos Event-B, que descrevem quatro níveis de detalhamento da linguagem. A noção de componentes e suas categorias é vista no nível *aadl_c0*, subcomponentes e referências são descritos no nível *aadl_c1*, a semântica individual das propriedades é descrita no nível *aadl_c2* e a semântica de dependência de propriedades que se desejava representar é especificada no nível *aadl_c3*.

A modelagem do comportamento, descrita no capítulo 8 é feita com LUSTRE e Event-B. Inicialmente, a linguagem LUSTRE é utilizada para a elaboração de um modelo síncrono simples para o protocolo PCI, com a finalidade de possuir uma especificação de fácil compreensão e que ilustra com clareza os aspectos ligados à concorrência vista num protocolo de barramento.

O método Event-B é utilizado para a modelagem e verificação parametrizada do protocolo. Dada a orientação a refinamentos vista em Event-B, a especificação foi feita de modo incremental, a fim de permitir a criação de uma plataforma de base para a especificação de protocolos síncronos em Event-B. Esta plataforma de base é inspirada no conceito de *patterns*, bastante visto em projetos de softwares em geral e atualmente sendo investigado em Event-B . *Patterns* têm o objetivo de simplificar alguns aspectos da especificação do sistema, por meio de um conjunto de especificações que executam uma função que pode estar presente em vários tipos de sistema. Alguns trabalhos ilustram exemplos interessantes de *patterns*: Ball e Butler [8] criaram *patterns* para especificar mecanismos de tolerância a falhas em sistemas multiagente, Cansell *et al* [12] desenvolveram *patterns* para incluir aspectos temporais numa especificação Event-B, e Lliasov [34] criou *patterns* para diversos tipos de sistemas e desenvolveu um *plug-in* que opera em conjunto com a ferramenta Rodin para automatizar a inclusão de *patterns* numa especificação.

Enquanto *patterns* em Event-B costumam ser especificados com eventos e invariantes que são incluídos nos sistemas através de refinamentos, o modelo aqui apresentado serve como uma especificação abstrata para a especificação dos protocolos, já que a especificação de um protocolo é um refinamento desta base.

Nesta dissertação, utilizam-se quatro níveis para representar um protocolo: o nível 0 é utilizado para a expressão do invariante de alto nível, e os outros três são utilizados para descrever protocolos específicos com uma representação abstrata (nível 1), um refinamento para a fase de arbitração (nível 2) e um refinamento para a fase de transferência (nível 3). O interesse desta modelagem é concentrado na verificação de um invariante abstrato, assim, somente o protocolo PCI teve sua fase de transferência detalhada, já que esta é muito mais complexa que a fase de arbitragem e não tem influência no modelo de alto nível.

Visto que a especificação do protocolo vai se tornando bastante intrincada até mesmo com o desenvolvimento incremental permitido pelo método Event-B, algumas simplificações foram feitas durante a modelagem. A fase de arbitração de ambos os protocolos foi especificada utilizando as seguintes hipóteses:

- Uma requisição de um controlador não pode ser cancelada.
- O árbitro não pode interromper uma transferência em curso. Como será visto nos eventos, o fim de uma transferência é anunciado pelo mestre.
- Não será implementada uma política para escolha do mestre, pois especificações de protocolos normalmente deixam esta parte à escolha dos projetistas.

Além disso, a modelagem do protocolo PCI não inclui medidas concretas para os instantes em que nenhum controlador requer o controle do barramento – num protocolo PCI real, o árbitro possui um procedimento para controlar o barramento enquanto este está inativo. Esta simplificação não foi necessária na modelagem do protocolo AMBA pois o árbitro deste atribui automaticamente o controle ao "mestre padrão" em caso de inatividade.

A modelagem da fase de transferência do protocolo PCI também foi simplificada com a consideração de que não ocorrem erros durante a transferência, pois a especificação do funcionamento do protocolo na presença de erros possui uma complexidade que foge ao escopo deste trabalho. Assume- se também que os dados transmitidos na linha de barramento são números naturais, a fim de simplificar a especificação utilizando-se um tipo de dados já disponível em Event-B.

7 Modelagem e Análise da Arquitetura do Protocolo PCI

7.1 Especificando barramentos com AADL

7.1.1 Nível *aadlpci0* – O componente *bus* AADL

Conforme visto no capítulo 3, a linguagem AADL possui a categoria pré-definida *bus* para representar barramentos de comunicação. Os componentes da categoria *bus* possuem implicitamente uma interface de conexão que provê acesso do tipo barramento (*bus access*) aos componentes de hardware que o compartilham. O único tipo de interface externa que componentes do tipo *bus* podem possuir representa o acesso destes a outros barramentos (*requires bus access*). Assim, os barramentos conectam as partes do sistema através de conexões de acesso para elementos de hardware, e os componentes de software executados em diferentes partes do hardware utilizam o barramento implicitamente através de seu mapeamento para componentes de hardware, e também através da propriedade de mapeamento de conexões exposta nas figuras 7 e 8. A figura 20 mostra a arquitetura simplificada do hardware de um sistema computacional que utiliza o barramento PCI: o barramento principal *system_bus* é acessado pelos dispositivos *audio_controller* e *video_controller* e pelos barramentos *bridge* e *peripheral*, que possuem interfaces para realizar este acesso. Os barramentos auxiliares, por sua vez, são acessados por outros dispositivos e todas as conexões da figura são do tipo *bus access*. O código AADL desta especificação encontra-se no apêndice A.

As características de um componente *bus* são dadas pelas propriedades disponíveis na linguagem, assim, a descrição de barramentos com este componente são feitas conforme a figura 21. Na definição do tipo *pci*, são dadas propriedades gerais do barramento, e nas definições de implementações, o tamanho de mensagem transmitida no barramento é fornecido de acordo com cada implementação.

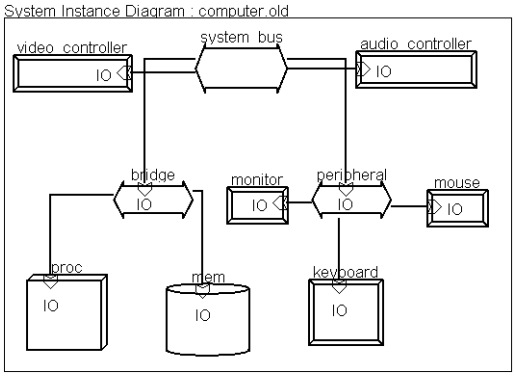


Figura 20: Componentes *bus* ligando elementos de hardware

```
1  bus pci
2    properties
3    Allowed_Connection_Protocol => Data_Connection;
4    Allowed_Access_Protocol => (Device_Access, Memory_Access);
5    Hardware_Source_Language => VHDL;
6  end pci;
7
8  bus implementation pci.b32bits
9    properties
10   Allowed_Message_Size => 1 B .. 4 B;
11 end pci.b32bits;
12
13 bus implementation pci.b64bits
14   properties
15   Allowed_Message_Size => 1 B .. 8 B;
16 end pci.b64bits;
```

Figura 21: Especificação AADL de um componente *bus*

As seguintes propriedades estão disponíveis no *standard* da AADL [50] para serem aplicadas em barramentos:

Allowed_Connection_Protocol Define se o barramento pode ser utilizado para ligar componentes de software que possuem conexões de dados, eventos ou ambos.

Allowed_Access_Protocol Define se o barramento pode ser utilizado para ligar dispositivos (*devices*), componentes de memória ou ambos.

Allowed_Message_Size Especifica os tamanhos mínimo e máximo de mensagens que podem ser transmitidas no barramento.

Transmission_Time Especifica um par de intervalos de tempo para traçar um modelo linear de tempos de transmissão.

Propagation_Delay Fornece o tempo decorrido entre o envio do primeiro *bit* de dados ao barramento e o recebimento deste pelo barramento.

Hardware_Description_Source_Text Especifica um conjunto de arquivos que contêm a descrição do hardware do barramento.

Hardware_Source_Language Especifica a linguagem utilizada para descrever o hardware do barramento.

Assign_Time, Assign_Byte_Time, Assign_Fixed_Time Estas três propriedades juntas especificam o tempo requerido para mover um bloco de N bytes no barramento. A propriedade *Assign_Time*, que representa o tempo total, é calculada através da soma de uma parcela fixa para todas as transferências com uma parcela proporcional ao número de bytes transmitidos: $(N * Assign\_Byte\_Time) + Assign\_Fixed\_Time$.

Propriedades adicionais podem ser incluídas em *property sets* criados pelos usuários. O *property set* SEI, criado pelo instituto de engenharia de software da universidade Carnegie Mellon [52] e já integrado ao OSATE, possui as seguintes propriedades que se aplicam a barramentos:

PowerCapacity, PowerBudget Estas propriedades definem, respectivamente, a capacidade fornecida ou consumida por um barramento, com valores dados em Watts.

BandWidthCapacity, BandWidthBudget Estas propriedades definem, respectivamente, a largura de banda fornecida ou consumida por um barramento, com valores dados em múltiplos de *bits* por segundo.

Três outras propriedades utilizam os barramentos como referência na descrição de outros componentes. A propriedade *Allowed Connection Binding Class* define as categorias de componentes que podem ser usados para realizar uma determinada conexão ou conjunto de conexões, já que componentes das categorias *processor* e *device* também podem ser vistos como dispositivos de conexão em certos casos. A propriedade *Allowed Connection Binding* fornece um conjunto de referências a componentes que podem realizar uma conexão, e a propriedade *Actual Connection Binding* define o componente que realmente realiza a conexão. Estas propriedades podem ser utilizadas em especificações incrementais de sistemas, onde somente a implementação concreta de um componente utiliza referências concretas a outros componentes.

7.1.2 Nível *aadlpci1* – O barramento detalhado

A representação anterior de barramentos de comunicação é interessante para a arquitetura de um sistema como um todo, mas é limitada para descrever a arquitetura interna

do barramento, pois não se pode representar as linhas que o compõem ou outros elementos específicos de um protocolo. Para representar a arquitetura de um barramento em um nível mais concreto, optou-se por utilizar componentes da categoria *system* para representar o barramento, o árbitro e os dispositivos que utilizam o barramento. Esta solução é aprovada por uma nova versão (ainda não oficial) do *standard* AADL [51]. Através desta representação, pode-se descrever as linhas do barramento e também as permissões de acesso em cada linha, conforme mostra a figura 22. Visto que o árbitro tem acesso a todas as linhas, mas só pode enviar dados às linhas *GNT*, seu acesso é descrito como restrito à leitura nas outras. As definições de tipos de dados fornecidas pelo Anexo Comportamental são utilizadas para especificar que as linhas do barramento são modeladas como variáveis booleanas no componente.

```
1  system pci_arbiter
2    features
3      frame: requires data access behavior::boolean {
4        Required_Access => access read_only;  };
5      req_1: requires data access behavior::boolean {
6        Required_Access => access read_only; };
7      req_2: requires data access behavior::boolean {
8        Required_Access => access read_only; };
9      gnt_1: requires data access behavior::boolean {
10       Required_Access => access read_write; };
11     gnt_2: requires data access behavior::boolean {
12       Required_Access => access read_write; };
13     irdy: requires data access behavior::boolean {
14       Required_Access => access read_only; };
15     trdy: requires data access behavior::boolean {
16       Required_Access => access read_only; };
17     devsel: requires data access behavior::boolean {
18       Required_Access => access read_only; };
19     ad: requires data access behavior::integer {
20       Required_Access => access read_only; };
21     command: requires data access data_types::command {
22       Required_Access => access read_only; };
23 end pci_arbiter;
```

Figura 22: Um árbitro PCI descrito com um *system* AADL

Deste mesmo modo, podem ser descritas as outras partes do sistema. Os dispositivos conectados ao barramento têm direito de escrita em todas as linhas, excetuando-se as linhas *GNT*, e não têm direito de leitura sobre as linhas *REQ* e *GNT* dos outros dispositivos. As especificações de dispositivos são genéricas, dispositivos concretos podem ser especificados estendendo-se o componente genérico.

7.2 Análise Semântica das Propriedades AADL utilizando Event-B

Durante a utilização de propriedades AADL para a especificação da arquitetura do protocolo PCI, pôde-se notar que o *parser* disponível no OSATE verifica a sintaxe de todas, mas o mesmo nem sempre acontece com a semântica destas. Enquanto as propriedades de acesso são verificadas para que o acesso requerido seja compatível com o acesso fornecido (um erro é gerado caso um componente forneça acesso apenas para leitura e outro tente um acesso de leitura e escrita), não há nenhuma verificação em certas propriedades que também dependem de outras propriedades, como no caso das propriedades de mapeamento *Allowed Connection Binding Class*, *Allowed Connection Binding* e *Actual Connection Binding*. Neste contexto, decidiu-se especificar formalmente alguns aspectos da semântica AADL com Event-B para possibilitar a criação de uma base adequada para expressão e verificação da semântica das propriedades referentes aos barramentos e também a outros componentes.

Conforme o que foi visto no capítulo 3, as propriedades contêm três informações básicas: o componente onde ela é declarada (implícito numa especificação AADL), o valor atribuído a ela, e o nome do componente que terá este valor de propriedade associado a ele. Este componente pode ser o próprio componente em que a propriedade é declarada (caso em que a cláusula *applies to* é dispensada), ou em qualquer membro da hierarquia de componentes contida no componente onde é declarada a propriedade. Neste caso, o nome de componente dado é composto pelos nomes de todos os componentes percorridos ao longo da hierarquia. A figura 23 mostra um exemplo do uso de propriedades AADL numa hierarquia de componentes, nas linhas 32-34 pode-se ver como propriedades são atribuídas a componentes em níveis hierárquicos mais distantes. A linha 33 mostra a atribuição de uma propriedade ao próprio componente onde ela é declarada, portanto, sem a cláusula *applies to*.

Para fazer o melhor uso possível da metodologia de Event-B, a parte da semântica AADL necessária para a expressão de propriedades é especificada em vários níveis, cada um especificado com um contexto Event-B. Primeiramente, um contexto Event-B *aadl_c0* é criado para definir as noções de categorias, tipos e implementações de componentes. O nível seguinte (*aadl_c1*) possui um contexto onde o mecanismo de subcomponentes é descrito de modo a tornar possível a obtenção de hierarquias de componentes em forma de árvore. No terceiro nível de abstração (*aadl_c2*), as propriedades AADL desejadas são especificadas e é feita a descrição do mecanismo de verificação individual de cada

```
1  system s end s;
2
3  system s1
4    features
5      i: in event port;
6      o: out event port;
7  end s1;
8
9  system s2 end s2;
10
11 system implementation s2.i
12   subcomponents
13     b1: bus b;
14 end s2.i;
15
16 bus b end b;
17
18 system implementation s1.i
19   subcomponents
20     son: system s2.i;
21 end s1.i;
22
23 system implementation s.i
24   subcomponents
25     sub1: system s1.i;
26     sub2: system s1.i;
27     connector: system s2.i;
28   connections
29     conn1: event port sub1.o -> sub2.i;
30     conn2: event port sub2.o -> sub1.i;
31   properties
32     Actual_Connection_Binding => reference sub2.son.b1 applies to sub1;
33     Actual_Connection_Binding => reference sub2.son.b1; --applies to self
34     Actual_Connection_Binding => reference sub2.son.b1 applies to conn1;
35 end s.i;
```

Figura 23: Um exemplo de uso de propriedades AADL numa hierarquia

propriedade, e no nível *aadl_c3*, são descritas as dependências entre as propriedades. Todos os níveis são feitos utilizando-se apenas os contextos Event-B, já que a especificação AADL não sofre mudanças enquanto é verificada.

Nestes quatro contextos, somente quatro provas (num total de 18 geradas) precisaram ser feitas com um provador de teoremas mais potente, e sem qualquer outro tipo de interferência do usuário. Isto se deve ao fato de que somente duas propriedades (expressas em teoremas) precisaram ser verificadas. As verificações feitas em axiomas resumem-se a assegurar a tipagem correta das constantes e a boa formação das expressões, o que normalmente é feito de forma automática por qualquer ferramenta de prova.

7.2.1 Nível *aadl_c0*: Noções básicas de componentes AADL

Conforme visto no capítulo 3, os componentes AADL fazem parte de categorias pré-definidas. Estas categorias são definidas em Event-B através de um conjunto enumerado, cujos elementos são declarados como constantes e axiomas atestam que os elementos são

diferentes entre si. O ambiente Rodin gera automaticamente as constantes e axiomas necessários através de um assistente para a criação de conjuntos enumerados. Parte da especificação do conjunto de categorias pode ser vista na figura 24.

CONTEXT aadl_c0

SETS

$CATEGORIES$

CONSTANTS

$system$
$processor$
$device$
$memory$
bus
$process$
$thread\_group$
$thread$
$subprogram$
$data$

AXIOMS

$axm1 : CATEGORIES = \{system, processor, device, memory, bus, process, thread\_group, thread, subprogram, data\}$
$axm2 : system \neq processor$
$axm3 : system \neq device$
...
$axm45 : thread \neq data$
$axm46 : subprogram \neq data$

END

Figura 24: Categorias de componentes AADL

Neste nível também é criado um conjunto de componentes, que representa todos os tipos e implementações de componentes vistos numa especificação AADL. O conjunto de componentes é formado pela união dos conjuntos disjuntos *c_types* e *c_impls*, que representam os conjuntos de tipos e implementações da especificação fornecida. A relação de implementação é descrita através da função total *typeof*, que relaciona um tipo para cada implementação. Também é criada a função total *categoryof*, que mapeia cada componente a uma categoria. Visto que uma implementação deve pertencer à mesma categoria de componentes que o tipo que a originou, um axioma deve ser criado para declarar que a composição das funções *categoryof* e *typeof* numa implementação de componente deve gerar o mesmo resultado que a aplicação direta da função *typeof*. Todas estas definições são vistas na figura 25.

CONTEXT aadl_c0

SETS

$COMPONENTS$

CONSTANTS

c_types
c_impls
$typeof$
$categoryof$

AXIOMS

$axm47 : categoryof \in COMPONENTS \rightarrow CATEGORIES$
$axm48 : typeof \in c\_impls \rightarrow c\_types$
$axm49 : COMPONENTS = c\_impls \cup c\_types$
$axm50 : c\_impls \cap c\_types = \varnothing$
$axm51 : c\_impls \lhd typeof; categoryof = c\_impls \lhd categoryof$

END

Figura 25: Definições básicas de componentes AADL

7.2.2 Nível *aadl_c1*: Definição da semântica de subcomponentes

Neste nível, os conceitos de subcomponentes e de hierarquia de componentes são inseridos na especificação. Eles são necessários para que as propriedades possam ser aplicadas em qualquer elemento interior ao componente em que uma propriedade é declarada, assim, é importante compreender como os componentes são instanciados em AADL.

Uma especificação AADL pode ser vista como uma "floresta" de componentes: alguns deles não são subcomponentes de nenhum outro, então a floresta começa com a criação de uma instância para cada componente que não seja subcomponente de outro. A partir daí, seus subcomponentes são instanciados como "filhos" em cada árvore, percorrendo toda a hierarquia até que sejam encontrados todos os componentes "folha", que não possuem subcomponentes. Partindo-se do princípio de que uma especificação AADL é finita, tem-se a garantia de que este processo é finito se a hierarquia de componentes realmente for uma floresta: deve-se provar que não há hierarquias cíclicas de componentes. Esta floresta pode ser transformada numa árvore com a inclusão de uma origem comum a todos os componentes-raiz.

Neste trabalho, define-se como "árvore de componentes" uma estrutura em forma de árvore que mostra as relações hierárquicas entre os componentes. Define-se também "árvore de instâncias" como uma estrutura em forma de árvore que representa como seria um sistema formado a partir da instanciação de todos os componentes que não são subcomponentes de outros.

A figura 29 mostra as árvores de componentes e instâncias formadas pela especificação AADL da figura 23. Enquanto a árvore de instâncias mostra toda a especificação AADL, a árvore de componentes restringe-se a mostrar as relações hierárquicas entre os componentes.

Neste nível, dois novos conjuntos abstratos são criados: o conjunto *NAMES* contém todos os nomes de referência utilizados numa especificação qualquer, e o conjunto *REFERENCES* representa todas as referências a componentes numa instanciação da especificação. Para facilitar a representação, foi criada uma referência especial chamada *root*, que servirá para unir todas as árvores numa só que terá esta referência como raiz. O conjunto de nomes também tem um elemento especial, chamado *self*, que representa um nome abstrato para todos os componentes que originalmente eram raízes das árvores de componentes e, por não serem subcomponentes, não tinham nenhuma instância nomeada. A relação entre componentes e subcomponentes foi modelada como uma função que liga uma implementação de componente a um nome e um componente, já que a relação de subcomponente em AADL não leva diretamente a uma instância, e sim a um conjunto de instâncias que terão o mesmo nome e instanciarão o mesmo componente. A cardinalidade deste conjunto é dada pela quantidade de instâncias da implementação que contém este par. Uma referência completa deve possuir, além de um nome e um componente, uma outra referência que representa o componente que a instanciou.

A figura 26 ilustra os conjuntos e axiomas aqui descritos. Do axioma *axm1* ao *axm6*, são definidas as constantes e funções. O axioma *axm7* define que somente implementações podem ser imagens da função *parentof*, pois somente implementações podem conter subcomponentes. O axioma *axm8* define que os descendentes do componente root são aqueles componentes que não figuram como subcomponentes de outros.

A partir da definição das referências oriundas da raiz da árvore de instâncias, pode-se incluir seus subcomponentes como descendentes na árvore. A figura 27 mostra os axiomas *axm9* e *axm11* que definem o mecanismo de inclusão de novas referências na árvore. O axioma *axm10* garante que não existem duas referências com o mesmo nome, mesmo componente e que são subcomponentes da mesma referência. O axioma *axm12* descreve as restrições impostas pela linguagem quanto às categorias de subcomponentes permitidas para cada categoria de componente.

Para tornar possível a especificação de caminhos ao longo de uma hierarquia, a definição de sequência do método B tradicional vista em [3] foi adaptada para representar uma sequência de nomes de componentes. Entretanto, a implementação de um método

CONTEXT aadl_c1
REFINES aadl_c0

SETS

$NAMES$
$REFERENCES$

CONSTANTS

$root$
$self$
$subcomponentsof$
...

AXIOMS

$axm1 : self \in NAMES$
$axm2 : subcomponentsof \in c\_impls \rightarrow (NAMES \nrightarrow COMPONENTS)$
$axm3 : root \in REFERENCES$
$axm4 : nameof \in REFERENCES \rightarrow NAMES$
$axm5 : componentof \in REFERENCES \setminus \{root\} \rightarrow COMPONENTS$
$axm6 : parentof \in REFERENCES \setminus \{root\} \rightarrow REFERENCES$
$axm7 : \forall r \cdot (r \neq root \wedge parentof(r) \neq root \Rightarrow componentof(parentof(r)) \in c\_impls)$
$axm8 : \forall c \cdot (\forall n \cdot (n \in NAMES \wedge \{n \mapsto c\} \notin ran(subcomponentsof) \Rightarrow$
$(\exists r \cdot (r \in REFERENCES \setminus \{root\} \wedge parentof(r) = root \wedge$
$nameof(r) = n \wedge n = self \wedge componentof(r) = c))))$

Figura 26: Definições de subcomponentes AADL

$axm9 : \forall r1, r2 \cdot (r2 \in (REFERENCES \setminus \{root\}) \wedge r1 \in (REFERENCES \setminus \{root\}) \wedge r2 = parentof(r1) \Rightarrow (nameof(r1) \mapsto componentof(r1) \in subcomponentsof(componentof(r2))))$
$axm10 : \forall r1, r2 \cdot (r2 \in (REFERENCES \setminus \{root\}) \wedge r1 \in (REFERENCES \setminus \{root\}) \wedge nameof(r1) = nameof(r2) \wedge parentof(r1) = parentof(r2) \Rightarrow r1 = r2)$
$axm11 : \forall r1, n, c \cdot (r1 \in (REFERENCES \setminus \{root\}) \wedge componentof(r1) \in c\_impls \wedge n \mapsto c \in subcomponentsof(componentof(r1)) \Rightarrow (\exists r2 \cdot (r2 \neq root \wedge nameof(r2) = n \wedge componentof(r2) = c \wedge parentof(r2) = r1)))$
$axm12 : \forall n, c2, c1 \cdot ((c1 \in c\_impls \wedge n \mapsto c2 \in subcomponentsof(c1) \wedge categoryof(c1) = data) \Rightarrow categoryof(c2) = data \wedge categoryof(c1) \neq subprogram \wedge categoryof(c1) \neq bus \wedge categoryof(c1) \neq device \wedge (categoryof(c1) = thread \Rightarrow categoryof(c2) = data) \wedge (categoryof(c1) = thread\_group \Rightarrow categoryof(c2) = data \vee categoryof(c2) = thread \vee categoryof(c2) = thread\_group) \wedge (categoryof(c1) = process \Rightarrow categoryof(c2) = thread \vee categoryof(c2) = data \vee categoryof(c2) = thread\_group) \wedge (categoryof(c1) = processor \Rightarrow categoryof(c2) = memory) \wedge (categoryof(c1) = memory \Rightarrow categoryof(c2) = memory) \wedge (categoryof(c1) = system \Rightarrow categoryof(c2) = system \vee categoryof(c2) = data \vee categoryof(c2) = process \vee categoryof(c2) = processor \vee categoryof(c2) = memory \vee categoryof(c2) = bus \vee categoryof(c2) = device))$

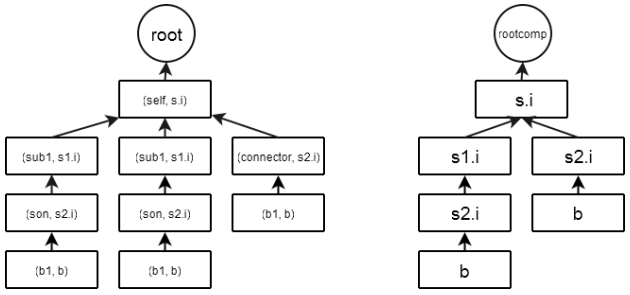
Figura 27: Mais definições de subcomponentes AADL

para criar, modificar e comparar caminhos não é trivial, já que as propriedades podem ser especificadas em vários componentes diferentes e ainda assim apontar para uma mesma instância, assim, a criação um algoritmo que compare valores de propriedades dadas em componentes diferentes (mas que apontem para a mesma instância) é uma tarefa de complexidade muito maior do que sua importância no escopo deste trabalho. Portanto, o uso de sequências neste trabalho é feito de forma abstrata: o axioma *axm13*, visto na figura 28, define um caminho de modo análogo à definição de sequência. O axioma *axm14* define que a árvore de subcomponentes de um componente é um conjunto de caminhos, *axm15* define que o primeiro elemento de um caminho de subcomponentes é o próprio componente que os contém, e *axm16* define a função *pathleaf*, que fornece o componente (que pode ser um tipo ou uma implementação) do último elemento de um caminho.

$axm13 : path = (\bigcup n \cdot (n \in \mathbb{N}) | (1 \mathinner{..} n) \rightarrow NAMES)$
$axm14 : subtree \in COMPONENTS \rightarrow \mathbb{P}(path)$
$axm15 : \forall p, name, num \cdot ((p \in path \wedge num \mapsto name \in p \wedge num = 1) \Rightarrow name = self)$
$axm16 : pathleaf \in path \rightarrow COMPONENTS$

Figura 28: Definições de caminhos para a hierarquia

Por fim, deve-se verificar que a hierarquia de componentes realmente forma uma árvore, algo que não é feito no *parser* do OSATE. A ausência de ciclos na árvore de componentes é verificada utilizando-se o método apresentado por Damchoom et al. [18] para verificar a ausência de ciclos em sistemas arborescentes de arquivos. A figura 30 mostra nos axiomas *axm17* a *axm20* como a árvore de componentes é feita. Os axiomas *axm21* a *axm25* descrevem a função "fechamento transitivo" (a menor relação transitiva que contém uma outra relação) para a função *parentcompof*, que é a adaptação da função *parentof* para implementações de componentes. Somente implementações de componentes são usadas para fazer a árvore, pois tipos de componentes não têm subcomponentes e assim não podem formar ciclos. O axioma *axm26* atesta a ausência de ciclos, já que nenhum componente é relacionado a si mesmo através da função fechamento transitivo. O teorema *thm2* descreve a conclusão desejada com a verificação de ausência de ciclos: se há um número finito de componentes e de nomes (isto é, se há um número finito de subcomponentes), o conjunto de referências formado a partir destes também será finito.



(a) Árvore de instâncias

(b) Árvore de componentes

Figura 29: Árvores de instâncias e componentes

7.2.3 Nível *aadl_c2*: Especificação da semântica individual de propriedades

Neste nível, pode-se declarar as propriedades desejadas. Um fator complicador na AADL é a falta de rigor na tipagem destas propriedades: algumas delas podem se aplicar tanto a componentes quanto a conectores, o que viola a tipagem rigorosa estabelecida em Event-B. Por esta razão, escolheu-se o conjunto de propriedades referentes ao mape-

$axm17 : rootcomp \in c\_impls$
$axm18 : parentcompof \in c\_impls \setminus \{rootcomp\} \rightarrow c\_impls$
$axm19 : \forall c, d \cdot (c \in c\_impls \setminus \{rootcomp\} \land d \in c\_impls \setminus \{rootcomp\} \land (d \in ran(subcomponentsof(c)) \Leftrightarrow parentcompof(d) = c))$
$axm20 : \forall d \cdot (d \in c\_impls \setminus \{rootcomp\} \land (\neg(\exists c \cdot (c \in c\_impls \setminus \{rootcomp\} \land (d \in ran(subcomponentsof(c)))) \Leftrightarrow parentcompof(d) = rootcomp)))$
$axm21 : comprel = c\_impls \leftrightarrow c\_impls$
$axm22 : tclcomp \in comprel \rightarrow comprel$
$axm23 : \forall c \cdot (c \in comprel \Rightarrow c \subseteq tclcomp(c))$
$axm24 : \forall c \cdot (c \in comprel \Rightarrow c; tclcomp(c) \subseteq tclcomp(c))$
$axm25 : \forall c, t \cdot (c \in comprel \land c \subseteq t \land c; t \subseteq t \Rightarrow tclcomp(c) \subseteq t)$
$axm26 : tclcomp(parentcompof) \cap id(c\_impls) = \varnothing$

THEOREMS

$thm2 : finite(COMPONENTS) \land finite(NAMES) \Rightarrow finite(REFERENCES)$

END

Figura 30: Verificação de ciclos na hierarquia de componentes

amento de software para componentes de memória, já que estas propriedades envolvem apenas componentes. A figura 31 mostra a semântica da propriedade *Allowed Memory Binding Class* com a definição da sintaxe no axioma *axm1*. O valor fornecido deve ser um subconjunto do conjunto $system, processor, memory$ e deve ser aplicado num caminho cujo componente final deve pertencer ao conjunto especificado no axioma *axm2*. Outras propriedades são descritas com o mesmo tipo de raciocínio.

CONTEXT aadl_c2

REFINES aadl_c1

CONSTANTS

$Allowed\_Memory\_Binding\_Class$
$Allowed\_Memory\_Binding$
$Actual\_Memory\_Binding$

AXIOMS

$axm1 : Allowed\_Memory\_Binding\_Class \in COMPONENTS \leftrightarrow (\mathbb{P}(\{system, processor, memory\}) \times path)$
$axm2 : \forall cat, p, c \cdot (c \mapsto (cat \mapsto p) \in Allowed\_Memory\_Binding\_Class \Rightarrow p \in subtree(c) \land categoryof(pathleaf(p)) \in \{thread, thread\_group, process, system, device, subprogram, processor\})$

Figura 31: Semântica individual de uma propriedade

7.2.4 Nível *aadl_c3*: Especificação de dependências entre propriedades

Após a definição da semântica de cada propriedade AADL individualmente, pode-se finalmente descrever as dependências entre propriedades. No caso das propriedades de mapeamento para memória, sabe-se que a lista de componentes especificada na propriedade

Allowed Memory Binding deve conter somente componentes das categorias especificadas na propriedade *Allowed Memory Binding Class*, caso ambas sejam aplicadas ao mesmo componente. Esta propriedade é dada pelo axioma *axm1* da figura 32. O axioma *axm2* declara que o componente especificado na propriedade *Actual Memory Binding* deve estar contido na lista fornecida na propriedade *Allowed Memory Binding* e o axioma *axm3* declara que o componente especificado na propriedade *Actual Memory Binding* deve pertencer a uma das categorias especificadas na propriedade *Allowed Memory Binding Class*. Os três axiomas são necessários pois a declaração de cada uma destas propriedades é facultativa, assim, todas devem estar relacionadas duas a duas.

CONTEXT aadl_c3

REFINES aadl_c2

AXIOMS

$axm1 : \forall trgt, c, listref, cats \cdot ((c \mapsto (cats \mapsto trgt) \in Allowed\_Memory\_Binding\_Class \land c \mapsto (listref \mapsto trgt) \in Allowed\_Memory\_Binding) \Rightarrow (pathleaf; categoryof)[listref] \subseteq cats)$

$axm2 : \forall trgt, c, ref, cats \cdot ((c \mapsto (cats \mapsto trgt) \in Allowed\_Memory\_Binding\_Class \land c \mapsto (ref \mapsto trgt) \in Actual\_Memory\_Binding) \Rightarrow (pathleaf; categoryof)[\{ref\}] \subseteq cats)$

$axm3 : \forall trgt, c, ref, listref \cdot (c \mapsto (listref \mapsto trgt) \in Allowed\_Memory\_Binding \land c \mapsto (ref \mapsto trgt) \in Actual\_Memory\_Binding) \Rightarrow (ref \in listref)$

END

Figura 32: Dependências entre propriedades

7.3 Conclusão

Este capítulo apresentou a modelagem da arquitetura de um protocolo de barramento, utilizando o barramento PCI como exemplo. A linguagem AADL foi utilizada para especificar os componentes típicos de um sistema com barramento em dois níveis: o nível *aadlpci0* retrata a visão de um barramento como um componente inserido numa arquitetura de sistema embarcado e o nível *aadlpci1* mostra com mais detalhes a estrutura do barramento e a interface utilizada para conectá-lo ao árbitro e aos controladores que o acessam.

Para que a análise de arquitetura seja mais apurada, é importante que a semântica das propriedades dos componentes AADL seja formalizada. Com este fim, foi proposto um modelo simplificado da semântica AADL em Event-B e um conjunto de propriedades dependentes teve suas dependências especificadas formalmente neste modelo.

8 *Modelagem e Verificação do Comportamento*

8.1 Modelagem e Verificação com LUSTRE

Apesar das limitações da linguagem LUSTRE na modelagem e verificação de sistemas parametrizados, sua utilização para especificar o comportamento de alto nível de sistemas síncronos auxilia na compreensão do modelo, já que o sincronismo inerente à linguagem simplifica a representação de tais protocolos. Uma especificação LUSTRE foi criada para representar o comportamento de alto nível de um protocolo síncrono e visualizá-lo com o simulador *luciole*. O sistema aqui especificado é composto por um árbitro e dois controladores que disputam o acesso ao barramento.

A especificação é composta por quatro funções: uma função principal, uma subfunção de arbitração, uma subfunção para cálculo de estados de controladores e uma função utilizada para verificação. Conforme já visto, todas as funções utilizam parâmetros booleanos para permitir a utilização do verificador *lesar*.

A função principal é vista na figura 33. Ela recebe como entrada um vetor com duas variáveis que simbolizam solicitações internas dos dispositivos, que serão usadas para a mudança de estado. Suas saídas representam os três possíveis estados (ocioso, requerendo controle e mestre) para cada um dos dois controladores modelados, além das variáveis de arbitração *owner*, que define o dispositivo que está controlando o barramento, e *hasOwner*, que é utilizada para sinalizar se o barramento está ocupado ou livre. Nas linhas 4 e 5, as variáveis de arbitração são inicializadas para valores que mantêm o barramento livre e são calculadas nos períodos seguintes através da função de arbitração, chamada *arbitration*. Nas linhas 6-11, as variáveis dos controladores são inicializadas de modo a manter ambos os controladores no estado ocioso, e para os períodos seguintes, são calculadas com o uso da função *calculate_states*.

A função de arbitração (figura 34) recebe os estados de ambos os controladores e cal-

```
1  node hi_lvl_protocol(reqs: bool^2) returns(owner, idle0, request0, master0,
2   idle1, request1, master1, hasOwner: bool);
3   let
4    (owner, hasOwner) = arbitration(idle0, request0, master0,
5     idle1, request1, master1);
6    (idle0, request0, master0) = (true, false, false) →
7     calculateState(false, owner, pre(idle0), pre(request0),
8                           pre(master0), reqs[0], hasOwner);
9    (idle1, request1, master1) = (true, false, false) →
10    calculateState(true, owner, pre(idle1),
11           pre(request1), pre(master1), reqs[1], hasOwner);
12  tel
```

Figura 33: A função principal do protocolo

cula os valores das variáveis de arbitração. Nas linhas 5-7, o árbitro verifica as requisições feitas pelos dispositivos, para escolher a mais prioritária. Nas linhas 8-15, o dispositivo mestre é escolhido. No primeiro período o árbitro não seleciona nenhum mestre, nos outros ele escolhe o dispositivo mais prioritário que tiver feito uma requisição, se o barramento estiver livre, ou mantêm o mestre atual caso este continue no estado de mestre. Pode-se observar que todas as decisões são feitas sobre valores anteriores das variáveis, já que um sinal só é lido pelos dispositivos no período seguinte ao que ele foi transmitido.

```
1  node arbitration(idle0, request0, master0, idle1, request1, master1: bool)
2   returns (owner, hasOwner: bool);
3   var firstreq: int;
4   let
5    firstreq = -1 → if pre(request0) then 0
6                    else if pre(request1) then 1
7                    else -1;
8    (owner, hasOwner) = (false, false) →
9      if pre(hasOwner) and ((pre(owner)= false and not pre(idle0))
10     or (pre(owner) = true and not pre(idle1)))
11      then (pre(owner), pre(hasOwner))
12      else if firstreq = -1 then (false, false)
13      else if firstreq = 0 then (false, true)
14      else (true, true);
15  tel
```

Figura 34: A função de arbitração

A função de tomada de decisão por parte dos controladores é bastante simples e pode ser vista na figura 35. Estando no estado ocioso, um dispositivo pode passar ao estado de requisição se tiver recebido um evento interno para fazê-lo. A partir do estado de requisição, um controlador pode manter-se no mesmo estado ou passar ao estado de mestre, se receber o acesso pela variável *owner*. Por fim, a partir do estado de mestre, o dispositivo pode manter-se no mesmo estado ou passar ao estado de ocioso quando o evento interno de requisição for retirado.

A função de verificação tem a finalidade de provar que as propriedades desejadas no sistema são satisfeitas. Aqui, deseja-se que a exclusão mútua seja respeitada (isto é, só

```
1  node calculateState(index, owner, preidle, prereq, premaster,
2                                  internalreq, hasOwner: bool)
3   returns (newidle, newreq, newmaster: bool)
4   let
5    (newidle, newreq, newmaster) = if preidle and not pre(internalreq)
6                                                  then (true, false, false)
7              else if preidle and pre(internalreq) then (false, true, false)
8              else if prereq and pre(hasOwner) and
9                  ((pre(owner) = false and index = false)
10                 or (pre(owner) = true and index = true))
11                 then (false, false, true)
12             else if prereq and
13                  ((pre(owner) = false and index = true)
14                   or (pre(owner) = true and index = false)
15                   or not pre(hasOwner))
16                   then (false, true, false)
17             else if premaster and not pre(internalreq)
18                              then (true, false, false)
19             else (false, false, true);
20  tel
```

Figura 35: A função de decisão dos controladores

um dispositivo pode estar no estado de mestre em um instante qualquer) e que os estados de cada controlador sejam válidos, com uma e somente uma das três variáveis de estado booleanas tendo o valor verdadeiro. Conforme a figura 36 mostra, esta função recebe os eventos internos de requisição como entrada e fornece um valor booleano "ok" como saída. O verificador validará o sistema se este valor sempre for verdadeiro, assim, ele deve corresponder a um predicado que represente as propriedades desejadas. O mecanismo da função de verificação consiste em executar a função principal do protocolo, e com os valores de saída desta, verificar se as propriedades de estados coerentes (linhas 7-14) e exclusão mútua (linha 15) são respeitadas. A conjunção destas duas variáveis é transmitida como parâmetro de retorno na linha 16. A execução do verificador Lesar nesta função gera um autômato com todos os estados alcançáveis do sistema e verifica se em todos eles o parâmetro de saída é verdadeiro, o que ocorre neste caso.

```
1  node verif (reqs: bool^2) returns(ok: bool);
2   var owner, idle0, request0, master0, idle1, request1,
3           master1, hasOwner, validstates, mutex: bool;
4   let
5    (owner, idle0, request0, master0,
6    idle1, request1, master1, hasOwner) = pci_n0(reqs);
7    validstates =
8         ((idle0 and not request0 and not master0)
9         or (not idle0 and request0 and not master0)
10        or (not idle0 and not request0 and master0))
11                  and
12        ((idle1 and not request1 and not master1)
13        or (not idle1 and request1 and not master1)
14        or (not idle1 and not request1 and master1))
15     mutex = not (master0 and master1);
16     ok =  validstates and mutex;
17  tel
```

Figura 36: A função de verificação

Apesar da falta de parametrização impedir que a verificação seja suficiente para o escopo deste trabalho, pode-se constatar que o modelo obtido é bastante simples e conciso. A compreensão do funcionamento de um protocolo num ambiente síncrono é importante para que um ambiente similar seja especificado em formalismos como Event-B, que não possuem este sincronismo inerente.

8.2 Modelagem e Verificação com Event-B

8.2.1 Um modelo básico para protocolos síncronos

Para especificar o comportamento de protocolos síncronos de barramento em Event-B, deve-se raciocinar em termos de partes estáticas e dinâmicas do sistema. Deste modo, é natural utilizar contextos, variáveis e invariantes para definir as variáveis e propriedades do protocolo, e representar as mudanças de estado de cada parte do sistema com eventos. Entretanto, percebe-se rapidamente que a especificação de eventos não pode ser feita de modo direto, já que dois eventos não podem ocorrer simultaneamente. Por esta razão, optou-se pela criação de especificações Event-B que permitam representar o mecanismo de sincronização visto fisicamente nos protocolos síncronos, onde os dispositivos (controladores e árbitro) tomam decisões em instantes não necessariamente iguais, mas lêem os dados enviados por outros no início do período seguinte. Estas especificações de base podem então ser refinadas para descrever diferentes protocolos síncronos.

O modelo básico criado é composto por um contexto chamado *synchronous_c0*, onde são descrito os conjuntos necessários para a especificação, e duas máquinas *synchronous_m0* e synchronous_m1. A máquina *synchronous_m0* mostra a visão mais abstrata possível de um protocolo, com um processo de decisão síncrono e abstrato. A máquina *synchronous_m1* refina *synchronous_m0* através da criação de eventos que representam o processo assíncrono de tomada de decisão por parte de cada dispositivo conectado no barramento.

A especificação do contexto *synchronous_c0* pode ser vista na figura 37. Nele são declarados os conjuntos abstratos *CONTROLLERS* e *CTR_STATES*, que representam os controladores e seus possíveis estados. O conjunto de controladores deve ser sempre abstrato para garantir a parametrização do modelo, e o conjunto de estados deve ser enumerado quando um protocolo específico for escolhido para refinar as especificações de base. Um axioma é dado para atestar que os conjuntos são não-nulos, hipótese evidentemente necessária para validar qualquer protocolo.

CONTEXT synchronous_c0

SETS

$CONTROLLERS$
CTR_STATES

AXIOMS

$axm1 : CONTROLLERS \rightarrow CTR\_STATES \neq \varnothing$

END

Figura 37: Conjuntos do modelo básico

Para descrever um protocolo síncrono da forma mais abstrata possível, optou-se por criar uma primeira máquina com apenas duas variáveis de interesse: o estado do barramento *arbbusy* e o estado do conjunto de controladores *gstateCtr*. Visto que um barramento pode estar livre ou ocupado, seu estado pode ser guardado numa variável booleana, enquanto o estado dos controladores deve ser uma função total que liga cada controlador a um estado.

Entretanto, somente estas variáveis não são suficientes, pois é necessário armazenar os valores anteriores delas, já que estes são usados como dados de entrada no processo de decisão de cada dispositivo. Assim, foram criadas as variáveis *prearbbusy* e *pregstateCtr* para armazenar os estados do período anterior. Para que o sistema seja sincronizado, supõe-se que todos os dispositivos agem antes do ciclo de relógio em que a leitura de dados (e consequente sincronização) é feita por todos os dispositivos. Assim, também é necessário criar a variável booleana *finishedArb* que indique que o árbitro já realizou uma ação no período. Uma variável similar para os controladores não é necessária, pois seu valor está implícito na definição de estados de controladores: a variável é uma função parcial que contém os estados dos controladores que já realizaram uma ação no período, assim, quando todos já tomaram uma ação, basta verificar que a função se tornou total. A declaração destas variáveis e seus tipos encontra-se na figura 38.

As mudanças de estado desta especificação são dadas por quatro eventos, cujo código na linguagem de Event-B é dado na figura 39:

INITIALISATION Evento obrigatório de inicialização de todas as variáveis do sistema. As variáveis *prearbbusy* e *arbbusy* são iniciadas com o valor booleano falso para simbolizar um barramento livre, a variável *pregstateCtr* é inicializada como uma função total aleatória, pois ela sempre deve assumir um valor de função total; em caso contrário, teria-se uma situação onde o sistema avançou um ciclo sem que

MACHINE synchronous_m0

SEES synchronous_c0

VARIABLES

$finishedArb$
$gstateCtr$
$pregstateCtr$
$arbbusy$
$prearbbusy$

INVARIANTS

$typinv\_fa : finishedArb \in BOOL$
$typinv\_gstate : gstateCtr \in CONTROLLERS \nrightarrow CTR\_STATES$
$typinv\_arbbusy : arbbusy \in BOOL$
$typinv\_pgstate : pregstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
$typinv\_parbbusy : prearbbusy \in BOOL$

Figura 38: Parte estática da máquina *synchronous_m0*

todos os controladores tivessem agido, o que contraria a hipótese de sincronização feita. A variável *gstateCtr* é inicializada como uma função vazia, para simbolizar que nenhum dispositivo tomou uma ação no período, e a variável *finishedArb* é inicializada com o valor falso para indicar que o árbitro ainda não realizou uma ação no período.

controllers_sync Evento que representa tomadas de ações de todos os controladores simultaneamente. Ele pode ser disparado quando a variável *gstateCtr* é o conjunto vazio, transformando-a em uma função total aleatória.

generic_arbiter_action Evento que representa uma tomada de ação por parte do árbitro. Ele pode ser disparado quando a variável *finishedArb* é falsa, e sua execução atribui o valor verdadeiro a esta variável e fornece um valor aleatório à variável *arbbusy*.

next_step Evento de sincronização do sistema, que pode ser disparado quando o árbitro e os controladores já realizaram suas ações. Ele atribui os valores das variáveis de estado às suas auxiliares "pre" (representando assim o avanço de um ciclo) e reinicializa as variáveis *finishedArb* e *gstateCtr*.

A separação das ações dos controladores é feita na máquina *synchronous_m1*, que refina *synchronous_m0*. Este refinamento utiliza as mesmas variáveis do nível abstrato e adiciona uma nova variável, chamada *stateCtr*, que possui o mesmo tipo e a mesma função de *gstateCtr*, com a diferença de que sua passagem de função vazia para função total pode ser feita em vários passos, como será visto nos novos eventos deste nível. As adições na parte estática da nova máquina podem ser vistas na figura 40. Os eventos vistos em

EVENTS

INITIALISATION

BEGIN
$ini\_parbbusy : prearbbusy := FALSE$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_pgstate : pregstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
$ini\_gstate : gstateCtr := \varnothing$
$ini\_fa : finishedArb := FALSE$
END

EVENT next_step
WHEN
$arb\_done : finishedArb = TRUE$
$ctr\_sync : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
THEN
$update\_states : pregstateCtr := gstateCtr$
$update\_arb : prearbbusy := arbbusy$
$reset\_gstate : gstateCtr := \varnothing$
$reset\_fa : finishedArb := FALSE$
END

EVENT generic_arbiter_action
WHEN
$arb\_rdy : finishedArb = FALSE$
THEN
$arb\_done : finishedArb := TRUE$
$action\_taken : arbbusy :\in BOOL$
END

EVENT controllers_sync
WHEN
$not\_sync : gstateCtr = \varnothing$
THEN
$ctr\_sync : gstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
END

Figura 39: Eventos da máquina *synchronous_m0*

synchronous_m0 permanecem, com algumas modificações, já que o evento de inicialização deve possuir uma ação a mais para inicializar a variável *stateCtr* com o conjunto vazio, o evento *next_step* possui uma ação adicional para reiniciar esta mesma variável e o evento *controllers_sync* possui agora uma guarda adicional para verificar se a variável *stateCtr* é uma função total, e uma ação adicional para atribuir à variável *gstateCtr* os valores de *stateCtr*. Assim, a variável global de estado continua sendo modificada em apenas um passo.

Um novo evento, visto na figura 41, foi criado para representar ações genéricas tomadas por um número qualquer de controladores. Este evento atribui à variável *stateCtr* um conjunto de pares formados por controladores e estados, descrevendo de forma abstrata ações que estes controladores tomaram. Evidentemente, somente controladores que ainda não fazem parte do domínio de *stateCtr* podem fazer parte do conjunto de controladores que disparam este evento.

A partir destas especificações, pode-se descrever protocolos síncronos de barramento

MACHINE synchronous_m1

REFINES synchronous_m0

SEES synchronous_c0

VARIABLES

...
stateCtr

INVARIANTS

$typinv\_stateCtr : stateCtr \in CONTROLLERS \nrightarrow CTR\_STATES$

Figura 40: Parte estática da máquina *synchronous_m1*

EVENTS

EVENT generic_controller_action
ANY
C
ns
WHERE
$some\_ctrls : C \in \mathbb{P}(CONTROLLERS)$
$didnt\_act : C \cap dom(stateCtr) = \varnothing$
$new\_states : ns \in C \rightarrow CTR\_STATES$
THEN
$statechange : stateCtr := stateCtr \cup ns$
END

Figura 41: O novo evento da máquina *synchronous_m1*

utilizando-se eventos para representar ações de grupos de controladores e do árbitro. Para protocolos com arbitração distribuída, pode-se usar estas especificações, mas elas devem ser adaptadas para que o dispositivo de arbitração seja retirado e suas variáveis sejam modificadas pelos controladores.

8.2.2 Especificações de protocolos utilizando o modelo básico

Para exemplificar o uso do método proposto, foram feitas as especificações dos protocolos PCI Local Bus e AMBA AHB. O protocolo PCI foi modelado nos níveis de comportamento abstrato (níveis 0 e 1), refinamento da arbitração (nível 2) e refinamento da transferência (nível 3), e o protocolo AMBA foi modelado nos níveis 0, 1 e 2. Nos dois protocolos, deseja-se verificar a propriedade de exclusão mútua: apenas um controlador pode enviar dados ao barramento durante um período.

Ambos os protocolos possuem o nível 0 composto por um contexto e uma máquina, Visto que os protocolos PCI e AMBA podem ser modelados com os mesmos estados abstratos, o nível 0 é idêntico para ambos. Estes estados são definidos no contexto da figura 42, e assim o conjunto *CTR_STATES* passa a ter os elementos *idle*, *request* e *master*, que

representam, respectivamente, os estados em que um controlador está ocioso, aguardando permissão do árbitro ou controlando o barramento (mestre). Com estes estados, é possível expressar a propriedade desejada na forma de um invariante: deseja-se que somente um controlador esteja no estado de mestre.

CONTEXT pci_amba_c0

REFINES synchronous_c0

CONSTANTS

idle
request
master

AXIOMS

$axm1 : CTR\_STATES = \{idle, request, master\}$
$axm2 : idle \neq request$
$axm3 : idle \neq master$
$axm4 : request \neq master$

END

Figura 42: Contexto comum aos protocolos PCI e AMBA

A especificação do invariante é feita na máquina *pci_amba_m0*. Esta máquina conta com as mesmas variáveis vistas anteriormente e adiciona uma nova, chamada *owner*, que é utilizada pelo árbitro para indicar o dispositivo que tem o controle do barramento. Do mesmo modo que as variáveis globais de estado, a variável *owner* precisa da uma variável auxiliar *preowner* para armazenar seu valor anterior. Com estas variáveis, a propriedade de exclusão mútua pode ser definida pelo invariante *singlemaster* visto na figura 43. Este invariante atesta que o conjunto de controladores que encontram-se no estado mestre deve estar contido no conjunto formado pela variável *owner*, que é unitário de acordo com a definição da variável. Deve-se enfatizar que a variável *preowner*, e não *owner*, deve ser utilizada na definição do invariante, pois os controladores lêem as informações no barramento no período seguinte ao que elas foram transmitidas; os invariantes devem ser especificados com cuidado para que tenham um significado correto. Além disso, deve-se levar em conta limitações da ferramenta: o invariante de exclusão mútua pode ser expresso como $card(stateCtr^{-1}[\{master\}]) \leq 1$ mas os provadores disponíveis não têm a capacidade de lidar com a operação de cardinalidade *card*, por isso deve-se evitar utilizá-la.

Os eventos vistos nesta máquina são os mesmos vistos anteriormente, modificados para interagir com as novas variáveis. Ambas devem ser inicializadas no evento *INITIALISATION*, *preowner* é atualizada no evento *next_step* e *owner* modificação no evento *generic arbiter action*). No ponto de vista da modelagem, estas modificações parecem su-

MACHINE pci_amba_m0

REFINES synchronous_m1

SEES pci_amba_c0

VARIABLES

...
owner
preowner

INVARIANTS

$typinv\_owner : owner \in CONTROLLERS$
$typinv\_preowner : preowner \in CONTROLLERS$
$singlemaster : stateCtr^{-1}[\{master\}] \subseteq \{preowner\}$

Figura 43: Especificação do invariante de exclusão mútua

ficientes, porém a utilização das ferramentas de verificação da plataforma Rodin mostra que a especificação não pode ser validada, pois não se pode provar que o evento *generic controller action* respeita o invariante de exclusão mútua. Por este motivo, ele também precisa ser modificado, com a adição da guarda *correct_master* vista na figura 44. Esta guarda garante que algum dos controladores do conjunto selecionado passará ao estado de mestre somente se ele for o valor da variável *preowner*. Conforme o que foi visto no capítulo 5, todos os eventos que refinarem este evento genérico também deverão refinar a guarda *correct_master*, assim, eles preservarão a propriedade de exclusão mútua verificada neste nível.

EVENTS

EVENT generic_controller_action
REFINES generic_controller_action
ANY
C
ns
WHERE
$some\_ctrls : C \in \mathbb{P}(CONTROLLERS)$
$didnt\_act : C \cap dom(stateCtr) = \varnothing$
$new\_states : ns \in C \rightarrow CTR\_STATES$
$correct\_master : ns^{-1}[\{master\}] \subseteq \{preowner\}$
THEN
$statechange : stateCtr := stateCtr \cup ns$
END

Figura 44: Modificação no evento *generic_controller_action*

8.2.3 Especificação do protocolo PCI

8.2.3.1 Nível 1 – Protocolo Abstrato

Após a especificação do invariante no modelo genérico de protocolo, pode-se especificar individualmente cada protocolo. Em um primeiro momento, deseja-se apenas modelar as

mudanças de estado em cada controlador. Tais mudanças ocorrem durante a fase de arbitração do protocolo, assim, os eventos de transferência de dados só serão modelados em um nível mais próximo da implementação.

Para a especificação das mudanças de estado no protocolo PCI, não é necessário criar outras variáveis. A nova máquina criada utiliza o contexto visto anteriormente e sua parte estática permanece a mesma, com exceção de novas propriedades que deseja-se provar neste nível. O protocolo foi especificado de modo que, na ausência de um controlador que requeira o controle do barramento, este permanece inativo e sem algum controlador que o controle, então deseja-se provar que se o barramento estiver ocupado, somente o dispositivo autorizado pelo árbitro estará no estado de mestre, e se o barramento estiver livre, nenhum dispositivo estará no estado de mestre. Estas duas propriedades são expressas pelos teoremas da figura 45.

MACHINE pci_m1

REFINES pci_amba_m0

SEES pci_amba_c0

THEOREMS

$singlemastertheorem : dom(stateCtr) = CONTROLLERS \land prearbbusy = TRUE \Rightarrow stateCtr^{-1}[\{master\}] \subseteq \{preowner\}$

$nomastertheorem : dom(stateCtr) = CONTROLLERS \land prearbbusy = FALSE \Rightarrow stateCtr^{-1}[\{master\}] = \varnothing$

Figura 45: Propriedades desejadas no comportamento abstrato do protocolo PCI

O evento genérico dos controladores visto na máquina *pci_amba_m0* é refinado para dar origem a quatro outros eventos. Três deles representam as mudanças de estados vistas na figura 46 e um descreve a situação em que o estado de um controlador não muda. Em todos estes eventos, somente o estado de um controlador é alterado, portanto faz-se necessária a criação de *witnesses* para associar as variáveis do evento genérico de *pci_amba_m0* às novas variáveis dos eventos desta máquina. A figura 47 mostra um destes novos eventos. O evento *req2master* tem como objetivo colocar em estado de mestre um controlador que esteja aguardando para utilizar o barramento e tenha recebido a autorização do árbitro. A guarda *is_req* denota o estado que o componente deve estar para disparar o evento, enquanto a guarda *is_owner* mostra a condição de autorização do árbitro. Esta condição pode ser vista também como o refinamento da guarda *correct_master* do evento genérico de controladores, já que representa uma condição ainda mais restritiva de disparo. As duas *witnesses* declaradas atestam que o conjunto C fornecido no evento genérico agora é representado por um conjunto formado pelo elemento c, e que o conjunto de pares ns é representado pelo par $\{c \mapsto master\}$. O evento *master2idle*

possui forma similar e pode ser disparado quando um controlador está em estado de mestre mas o árbitro retirou sua autorização para utilizar o barramento. O evento *idle2req* é modelado como não-determinístico, pois as requisições dos controladores não podem ser previstas. Assim, ele pode disparar sempre que existirem controladores ociosos. O evento *defaultCtrlEvent* pode ocorrer sempre que existirem controladores ociosos ou que não possam disparar os eventos *req2master* ou *master2idle*. Com isso, garante-se que sempre é possível disparar um destes eventos para cada controlador durante um período.

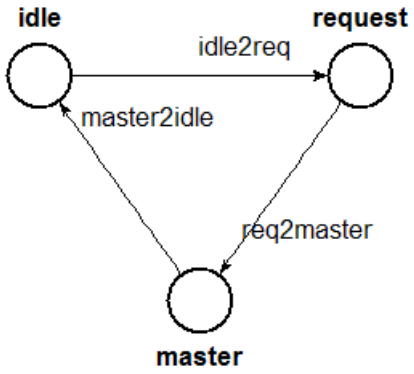


Figura 46: Estados de um controlador no modelo abstrato do PCI

EVENTS

EVENT req2master
REFINES generic_controller_action
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_req : pregstateCtr(c) = request$
$is\_owner : prearbbusy = TRUE \wedge preowner = c$
WITNESSES
$C : C = \{c\}$
$ns : ns = \{c \mapsto master\}$
THEN
$going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$
END

Figura 47: Um evento de controlador PCI na especificação *pci_m1*

O árbitro também passa a possuir quatro eventos neste refinamento: o evento *idle2busy* atribui o controle do barramento livre a um controlador, o evento *busy2busy* altera o controlador que tem acesso ao barramento, o evento *busy2idle* retira a autorização do controlador em estado de mestre e deixa o barramento ocioso, e o evento *defaultArbEvent* mantém os valores do período anterior nas variáveis de arbitragem. O evento *busy2busy*, descrito na figura 48, pode alterar a variável *owner* sempre que houver ao menos um controlador à espera para utilizar o barramento. Se houver mais de um, o novo mestre é escolhido de forma aleatória. Tanto o instante em que este evento realmente é disparado

(o evento *defaultArbEvent* também pode disparar com a mesma guarda) quanto o novo mestre são deixados de forma não-determinista pois somente em níveis mais concretos que estas questões passam a ter importância.

EVENTS

EVENT busy2busy
REFINES generic_arbiter_action
ANY
$c2$
WHERE
$arb\_ready : finishedArb = FALSE$
$old\_owner : prearbbusy = TRUE$
$a\_req : c2 \in pregstateCtr^{-1}[\{request\}]$
THEN
$arb\_done : finishedArb := TRUE$
$going\_busy : arbbusy := TRUE$
$newowner : owner := c2$

Figura 48: Um evento do árbitro na especificação *pci_m1*

Após a especificação destes novos eventos, deve-se validar a máquina. Nem sempre as obrigações de prova são triviais, e as ferramentas automáticas de prova também possuem limitações em suas táticas de prova, assim, muitas vezes deve-se provar invariantes auxiliares que devem então ser utilizados como hipóteses para realizar as obrigações de prova mais complexas. Os invariantes auxiliares criados para finalizar as obrigações de prova dos teoremas desta máquina encontram-se disponíveis no apêndice B.

8.2.3.2 Nível 2 – Refinamento da Arbitração

Com a máquina *pci_m1* modelada e verificada, pode-se realizar um novo refinamento para mostrar o funcionamento concreto da arbitração do protocolo PCI.

O mecanismo concreto de arbitração é modelado com a inclusão das variáveis booleanas *REQ* e *GNT*, assim como as auxiliares *preREQ* e *preGNT*. As variáveis abstratas de arbitração são mantidas pois elas não podem ser traduzidas diretamente para estas novas variáveis. Um novo evento, chamado *end_master*, é adicionado para representar o instante em que o mestre retira o sinal de sua linha *REQ*, para avisar o árbitro que irá finalizar o uso do barramento. Este evento pode ser visto na figura 49. Ele é um refinamento do evento "default" de controlador, já que o controlador que o dispara não tem seu estado alterado.

Todos os eventos, exceto *controllers_sync*, devem ser levemente modificados para que utilizem também as novas variáveis. A figura 50 mostra os refinamentos para os eventos *req2master* e *busy2busy*. No evento *req2master*, duas condições de guarda são adicionadas para indicar os valores que as variáveis concretas devem ter para que o controlador esteja

MACHINE pci_m2

REFINES pci_m1

SEES pci_amba_c0

EVENTS

EVENT end_master
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$is\_master : pregstateCtr(c) = master$
$has\_req : preREQ(c) = TRUE$
$didnt\_act : c \notin dom(stateCtr)$
$is\_owner : preowner = c \land prearbbusy = TRUE$
$concrete\_owner : preGNT(c) = TRUE$
THEN
$stop\_req : REQ := REQ \Leftarrow \{c \mapsto FALSE\}$
$keep\_states : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
END

Figura 49: O novo evento da especificação *pci_m2*

na situação descrita por suas variáveis abstratas de estado. No evento *busy2busy*, as mudanças são mais significativas: além da junção de duas condições (*concrete old owner* e *concrete new req*) que representam de forma concreta a guarda abstrata, há a criação da condição extra de guarda *concrete old not req*, que torna este evento determinístico, já que ele só será disparado se o mestre retirar seu sinal *REQ*.

Após a inclusão das novas variáveis nos eventos, deve-se verificar que elas possuem o comportamento esperado. Para isto, foram criados dois invariantes, mostrados na figura 51 que relacionam as variantes concretas e abstratas de arbitração. O invariante *nomaster_concrete* mostra as equivalências entre a variável concreta *GNT* e as variáveis abstratas *owner* e *arbbusy*, deve-se notar que a relação muda de acordo com o estado do barramento, dificultando a criação de um invariante de colagem adequado. Ainda assim, posto que estes invariantes puderam ser provados, as variáveis concretas são válidas.

8.2.3.3 Nível 3 – Refinamento da Transferência

Por fim, pode-se detalhar a fase de transferência do protocolo PCI. Um contexto adicional deve ser criado para auxiliar na representação de novas linhas do barramento. Este contexto, chamado *pci_c1* e mostrado na figura 52, define o conjunto enumerado *COMMANDS*, que contém dois elementos que representam as operações básicas de leitura e escrita. Os axiomas *axm3* e *axm4* são declarados para representar hipóteses vistas em sistemas reais, necessárias para algumas obrigações de prova.

MACHINE pci_m2

REFINES pci_m1

SEES pci_amba_c0

EVENTS

EVENT req2master
REFINES req2master
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_req : pregstateCtr(c) = request$
$is\_owner : prearbbusy = TRUE \land preowner = c$
$is\_req\_concrete : preREQ(c) = TRUE$
$is\_owner\_concrete : preGNT(c) = TRUE$
THEN
$going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$
END

EVENT busy2busy
REFINES busy2busy
ANY
$c1$
$c2$
WHERE
$arb\_ready : finishedArb = FALSE$
$old\_owner : prearbbusy = TRUE \land preowner = c1$
$a\_req : c2 \in pregstateCtr^{-1}[\{request\}]$
$concrete\_old\_owner : preGNT(c1) = TRUE$
$concrete\_old\_not\_req : preREQ(c1) = FALSE$
$concrete\_new\_req : preREQ(c2) = TRUE$
THEN
$arb\_done : finishedArb := TRUE$
$going\_busy : arbbusy := TRUE$
$newowner : owner := c2$
$concrete\_newowner : GNT := (CONTROLLERS \times \{FALSE\}) \lhd \{c2 \mapsto TRUE\}$
END

Figura 50: Exemplos de eventos refinados na especificação *pci_m2*

MACHINE pci_m2

REFINES pci_m1

SEES pci_amba_c0

INVARIANTS

$nomaster\_concrete : (arbbusy = FALSE \Leftrightarrow GNT^{-1}[\{TRUE\}] = \varnothing) \land (prearbbusy = FALSE \Leftrightarrow preGNT^{-1}[\{TRUE\}] = \varnothing)$
$singlemaster\_concrete : (arbbusy = TRUE \Leftrightarrow GNT^{-1}[\{TRUE\}] = \{owner\}) \land (prearbbusy = TRUE \Leftrightarrow preGNT^{-1}[\{TRUE\}] = \{preowner\})$

Figura 51: Invariantes verificados para validar as novas variáveis

A máquina *pci_m3*, utilizada para descrever a especificação com detalhamento da fase de transferência, possui várias novas variáveis (figura 53 que correspondem às principais linhas do barramento PCI. As variáveis *DEVSEL*, *IRDY*, *TRDY* e *FRAME* correspondem às linhas homônimas explicadas no capítulo 2, sendo representadas com valores booleanos. A variável *command* representa a linha *C/BE*, e pertence ao tipo *COMMANDS*.

CONTEXT pci_c1

REFINES pci_amba_c0

SETS

$COMMANDS$

CONSTANTS

$read$
$write$

AXIOMS

$axm1 : COMMANDS = \{read, write\}$
$axm2 : read \neq write$
$axm3 : CONTROLLERS \rightarrow \mathbb{N} \neq \varnothing$
$axm4 : \forall c \cdot (c \in CONTROLLERS \Rightarrow CONTROLLERS \setminus \{c\} \neq \varnothing)$

END

Figura 52: Contexto utilizado na máquina *pci_m3*

As variáveis *busdata* e *address* representam a linha multiplexada *AD*, elas devem ser separadas por serem de tipos diferentes. A variável *hasAddress* é utilizada para indicar se a transferência encontra-se em fase de endereço ou dados, pois durante a fase de dados, a variável *address* não tem significado físico. A variável *buffers* representa os buffers de entrada de cada controlador, e a variável *slave* representa um conjunto de booleanos que indica quais são os controladores que estão sendo utilizados como escravos durante uma transferência.

MACHINE pci_m3

REFINES pci_m2

SEES pci_c1

VARIABLES

$DEVSEL$
$IRDY$
$TRDY$
$command$
$busdata$
$buffers$
$address$
$slave$
$FRAME$
$preDEVSEL$
...

Figura 53: Novas variáveis da máquina *pci_m3*

Neste nível, os eventos que representam ações do árbitro permanecem os mesmos e os eventos de ações dos controladores são refinados para ilustrar os mecanismos de transferência. O evento *req2master* é refinado pelos eventos *req2write* e *req2read*, que

descrevem o início das operações básicas de escrita e leitura, respectivamente. A figura 54 mostra o evento *req2write*. Além da mudança de estado, outras quatro ações são adicionadas: a ação *start_writing* representa o posicionamento do comando de escrita na linha *C/BE*, a ação *call_slave* descreve a utilização da linha *AD* para transmissão do endereço do escravo, a ação *frame_on* aciona a linha *FRAME* para sinalizar o início de uma transferência e a ação *addressphase* atribui o valor verdadeiro à variável *hasAddress* para sinalizar que a transferência está em fase de endereço.

EVENT req2write
REFINES req2master
  ANY
    c
  WHERE
    $a\_ctrl : c \in CONTROLLERS$
    $didnt\_act : c \notin dom(stateCtr)$
    $is\_req : pregstateCtr(c) = request$
    $is\_owner : prearbbusy = TRUE \land preowner = c$
    $is\_req\_concrete : preREQ(c) = TRUE$
    $is\_owner\_concrete : preGNT(c) = TRUE$
  THEN
    $going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$
    $start\_writing : command := write$
    $call\_slave : address :\in (CONTROLLERS \setminus \{c\})$
    $frame\_on : FRAME := TRUE$
    $addressphase : hasAddress := TRUE$
  END

Figura 54: O evento concreto *req2write*

Outros eventos são adicionados para ilustrar as diferentes etapas da fase de dados das transferências. Todos refinam o evento *default*, pois o dispositivo mestre que pode dispará-los não muda seu estado. Um destes eventos é visto na figura 55. Este evento representa um período normal da operação de escrita, onde o mestre envia dados e mantém suas variáveis de controle com os mesmos valores, pois ainda tem mais dados a enviar. Além das condições vistas no evento *default*, a guarda possui uma condição *is_writer* para confirmar que o mestre encontra-se numa operação de escrita, e a condição *a slave or start*, que verifica que a transferência encontra-se no primeiro bloco de dados (condição $preIRDY = FALSE$) ou que o escravo já está pronto para receber o bloco seguinte de dados (condição $preDEVSEL = TRUE \land preTRDY = TRUE$), no caso de já terem sido enviados outros blocos anteriormente. Suas ações incluem a manutenção das variáveis de estado e *REQ*, a escrita de um valor no barramento, a asserção da linha *IRDY* para sinalizar que um dado foi enviado, e a atribuição do valor falso à variável *hasAddress* para sinalizar que a transferência encontra-se em fase de dados.

O evento *master2idle* também deve ser dividido em dois, para representar as ações tomadas pelo mestre em seu último período de transferência, já que esta pode ser de escrita ou leitura.

EVENT normal_master_write
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_master : pregstateCtr(c) = master$
$is\_owner : prearbbusy = TRUE \wedge preowner = c$
$has\_request : preREQ(c) = TRUE$
$is\_writer : precommand = write$
$a\_slave\_or\_start : (preDEVSEL = TRUE \wedge preTRDY = TRUE) \vee preIRDY = FALSE$
$concrete\_owner : preGNT(c) = TRUE$
THEN
$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
$write\_data : busdata :\in \mathbb{N}$
$irdy : IRDY := TRUE$
$dataphase : hasAddress := FALSE$
END

Figura 55: O evento concreto *normal master write*

A especificação da transferência é completa através da adição de eventos para representar o comportamento de um controlador escravo. O evento *normal slave write*, utilizado na fase de dados de uma operação de escrita, é visto na figura 56. A sua guarda verifica que ele encontra-se como escravo de uma transferência (condição *is_slave*), que há uma transferência em curso (*transfer_going*), que o mestre está preparado e enviando dados (*master_ready*), além das condições de guarda do evento abstrato. Se a guarda é satisfeita, o escravo armazena as informações da linha *AD* em seu *buffer* e sinaliza que a recepção foi bem-sucedida através da linha *TRDY*. Seu sinal *REQ* e seu estado permanecem inalterados.

EVENT normal_slave_write
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$is\_slave : preslave(c) = TRUE$
$transfer\_going : preFRAME = TRUE$
$master\_ready : preIRDY = TRUE$
$didnt\_act : c \notin dom(stateCtr)$
$write\_transfer : precommand = write$
$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$
$notgnt : preGNT(c) = FALSE$
THEN
$save\_data : buffers(c) := prebusdata$
$trdy : TRDY := TRUE$
$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
END

Figura 56: O evento concreto *normal slave write*

8.2.4 O Protocolo AMBA

8.2.4.1 Nível 1 – Protocolo Abstrato

O comportamento do protocolo AMBA em um nível abstrato é similar ao comportamento do PCI, com um conjunto de eventos bastante similar devido aos estados abstratos idênticos. Na especificação abstrata, a única mudança refere-se ao procedimento utilizado na ausência de um dispositivo que esteja esperando o barramento. No AMBA, sempre deve haver um controlador em estado de mestre, assim, na falta de requisições, atribui-se o controle do barramento a um controlador "default", então a máquina de estados de um controlador deve conter uma transição a mais, para ligar o estado *idle* ao estado *master*, conforme a figura 57.

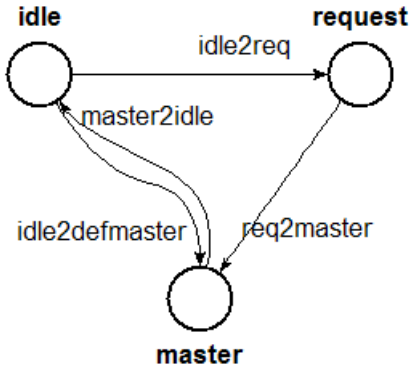


Figura 57: Estados abstratos de um controlador AMBA

No nível abstrato, o controlador que será o mestre do barramento quando este estiver ocioso é escolhido aleatoriamente, assim, o evento *busy2idle* disparado pelo árbitro permanece o mesmo. O comportamento do controlador mestre *default* será como o de um mestre normal, mas a variável *arbbusy* será mantida com o valor falso, para indicar que nenhuma transferência está ocorrendo e que o barramento encontra-se disponível. O evento criado para representar esta passagem direta de um controlador do estado ocioso para mestre pode ser visto na figura 58.

8.2.4.2 Nível 2 – Refinamento da Arbitração

No nível 2, a necessidade de manter um dispositivo mestre tornou a modelagem mais simples, pois sempre há um dispositivo mestre, assim, o sinal *HGRANT* sempre estará verdadeiro para um dispositivo e pode-se traçar uma correspondência direta entre este sinal e a variável abstrata *owner*. A correspondência, formalizada através de um invariante de colagem, é vista na figura 59, nos invariantes *inv2* e *inv4*: a variável *HGRANT*

EVENT idle2defmaster
REFINES generic_controller_action
ANY
c
WHERE
$grd1 : c \in CONTROLLERS$
$grd2 : c \notin dom(stateCtr)$
$grd3 : pregstateCtr(c) = idle$
$grd4 : preowner = c$
WITNESSES
$C : C = \{c\}$
$ns : ns = \{c \mapsto master\}$
THEN
$act1 : stateCtr := stateCtr \cup \{c \mapsto master\}$
END

Figura 58: Um controlador ocioso passando diretamente a mestre

deve mapear todos os componentes ao valor falso, exceto para o componente correspondente à variável abstrata *owner*. Uma correspondência análoga é feita para a variável auxiliar *preHGRANT*. Em meio às variáveis introduzidas neste nível, nota-se a criação de duas variáveis com o prefixo *prepre*, ambas têm a função de armazenar valores de um instante de tempo anterior ao referenciado pelas variáveis *pre*. Este método de armazenamento de valores anteriores em sistemas síncronos é inspirado na semântica de LUSTRE, onde a memória necessária para a execução de um programa depende da quantidade de operadores *pre* aninhados.

MACHINE amba_n2

REFINES amba_n1

SEES amba_c1

VARIABLES

...
$HGRANT$
$preHGRANT$
$prepreHGRANT$
$preprearbbusy$
$HMASTER$
$preHMASTER$

INVARIANTS

$inv1 : HGRANT \in CONTROLLERS \rightarrow BOOL$
$inv2 : HGRANT = (CONTROLLERS \times \{FALSE\}) \lhd\!\!+ \{owner \mapsto TRUE\}$
$inv3 : preHGRANT \in CONTROLLERS \rightarrow BOOL$
$inv4 : preHGRANT = (CONTROLLERS \times \{FALSE\}) \lhd\!\!+ \{preowner \mapsto TRUE\}$

Figura 59: Invariantes de colagem no refinamento do AMBA

Devido à troca de variáveis feita neste refinamento, a inicialização das variáveis é ligeiramente diferente à vista em outras especificações. Devem ser dadas *witnesses* que tornem explícita a correspondência entre a inicialização das variáveis abstratas e concretas, quando as variáveis abstratas são inicializadas de modo não-determinístico e suas

equivalentes concretas apresentam valores concretos. Assim, como pode-se ver na figura 60, a inicialização começa com correspondências para as variáveis $owner'$ e $preowner'$. O valor *DefCtr* é definido como uma constante no conjunto dos controladores, representando o mestre "default" do sistema.

EVENTS

INITIALISATION

BEGIN
WITNESSES
$owner' : owner' = DefCtr$
$preowner' : preowner' = DefCtr$
$ini\_fa : finishedArb := FALSE$
$ini\_state : stateCtr := \varnothing$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_HGRANT : HGRANT := (CONTROLLERS \times \{FALSE\}) \lhd \{DefCtr \mapsto TRUE\}$
$act2 : gstateCtr := \varnothing$
$act4 : HMASTER := DefCtr$
...
END

Figura 60: Inicialização no refinamento do AMBA

Os outros eventos são refinados através da substituição das variáveis abstratas pelas concretas, com a validação do refinamento sendo efetuada através da verificação do invariante de colagem. Um novo evento, visto na figura 61 é incluído para representar o comportamento da variável *HMASTER*, este evento precisa de valores dos dois períodos anteriores ao corrente para verificar se o período atual é o segundo de uma transferência. A disjunção vista na guarda *grd2* é utilizada para os casos de ociosidade ou de uma transferência real.

EVENT authtransferArb REFINES samestateArb
ANY
c
WHERE
$grd1 : finishedArb = FALSE$
$grd2 : ((prearbbusy = FALSE \land pregstateCtr^{-1}[\{request\}] = \varnothing) \lor (prearbbusy = TRUE)) \land (preprearbbusy = FALSE \lor prepreHGRANT \neq preHGRANT)$
$grd3 : preHGRANT(c) = TRUE$
THEN
$act1 : finishedArb := TRUE$
$act2 : arbbusy := prearbbusy$
$act3 : HGRANT := preHGRANT$
$act4 : HMASTER := c$
END

Figura 61: O evento de arbitração que configura a variável *HMASTER*

8.3 Balanço das Provas Realizadas

Conforme a discussão do capítulo anterior, os provadores de teoremas disponíveis na ferramenta Rodin ainda não têm a capacidade de concluir automaticamente todas

as provas necessárias para validar uma especificação Event-B. Devido a esta limitação, é importante verificar a quantidade de provas que podem ser feitas automaticamente, e assim ter uma noção do tempo necessário para validar um modelo de protocolo de barramento.

A tabela 4 mostra o balanço geral de provas. Constata-se que cerca de 60% das provas foram realizadas de maneira automática, e que nas máquinas onde as propriedades abstratas são inseridas nos protocolos (*pci_m1, amba_m1, amba_m2*) a proporção de obrigações de prova manuais é particularmente alta. Isto se deve principalmente ao fato do provador "default", lançado automaticamente, não ser capaz de realizar algumas provas elementares, tornando suficiente a troca do mecanismo de prova para a validação. Nas máquinas de base, somente invariantes de tipo são criados, e o provador "default" pode finalizar as provas automaticamente. Na máquina *pci_amba_m0*, é necessário um provador de teoremas mais potente para provar o invariante de exclusão mútua sem a interferência do usuário. Na máquina *pci_m1*, a grande quantidade de invariantes auxiliares criados para a prova dos teoremas gerou muitas provas que só puderam ser concluídas com um provador mais forte, e em alguns casos, foi necessário realizar parte da prova manualmente. Nesta especificação, os invariantes e teoremas não apresentam grande complexidade, assim, cada prova feita parcialmente com interferência do usuário levava no máximo uma dezena de minutos. Na máquina *pci_m2*, as provas realizadas com interferência do usuário também deveram-se à introdução de novos invariantes, e apresentaram complexidade similar às da máquina *pci_m1*. A especificação *pci_m3* contém 14 provas que não puderam ser feitas automaticamente, porém todas são referentes a refinamentos de guardas ou verificações de "boa definição" de predicados, onde o provador "default" não conseguiu provar automaticamente que as expressões são bem-formadas e sem erros de tipagem. Cada uma destas provas requer apenas a troca do provador de teoremas e poucos cliques para ser finalizada. De modo análogo, a especificação *amba_m1* possui praticamente metade das provas feitas com a interferência do usuário, mas todas puderam ser feitas em poucos cliques. As provas manuais da máquina *amba_m2* também foram feitas rapidamente mas é necessário o bom uso das *witnesses*, pois as *witnesses* que o ambiente Rodin gera automaticamente (quando o usuário não as insere) nem sempre efetuam a correta ligação entre as variáveis abstratas e concretas. Assim, constata-se que as especificações aqui apresentadas não apresentam provas manuais demasiadamente intrincadas, mas é necessário que o usuário tenha o conhecimento básico das equações utilizadas nas provas para que estas sejam finalizadas rapidamente quando alguma interferência manual é necessária.

Especificação	Provas Automáticas	Provas Manuais
$synchronous\_m0$	8	0
$synchronous\_m1$	4	0
pci_amba_m0	5	1
pci_m1	75	33
pci_m2	34	10
pci_m3	60	14
$amba\_m1$	14	13
$amba\_m2$	37	23
$Total$	237	94

Tabela 4: Provas realizadas nas especificações de protocolos

8.4 Conclusão

Este capítulo apresentou os modelos criados em LUSTRE e Event-B para estudar e validar o comportamento de protocolos de barramento. LUSTRE foi utilizada para a criação de um modelo comportamental síncrono de fácil compreensão, devido à sua simplicidade, e o método Event-B foi utilizado para exprimir o comportamento detalhado dos protocolos.

A especificação abstrata de protocolo criada em LUSTRE tem utilidade reduzida para fins de verificação, dada a ausência de parametrização em LUSTRE. Apesar disto, a modelagem é interessante para que os aspectos de sincronização sejam bem compreendidos. O interesse deste modelo síncrono pode ser visto na criação de um modelo síncrono de base num ambiente assíncrono como Event-B: algumas das adaptações necessárias para representar este sincronismo foram baseadas na sintaxe e semântica LUSTRE.

A modelagem detalhada com Event-B permitiu a realização de uma verificação formal parametrizada e incremental, bem como a especificação de protocolos de barramento em diversos níveis de abstração. Além de facilitar a compreensão dos protocolos, o método permitiu a criação de um modelo de base genérico para protocolos e a validação da propriedade de exclusão mútua num nível abstrato. A noção de refinamento em Event-B permite a preservação desta propriedade em níveis mais concretos.

Parte IV

Conclusões

9 Conclusões

Este trabalho apresentou uma abordagem para especificação e validação de protocolos síncronos de barramentos de comunicação enfatizando seus aspectos de arquitetura e comportamento. Deste modo, pôde-se descrever os protocolos tanto no nível de hardware quanto no nível de software, através da atribuição de propriedades de hardware em alto nível, do mapeamento de componentes de hardware para executar os módulos do software, e da descrição da arquitetura e do comportamento do software para que este seja verificado de forma parametrizada. Visto que tanto a arquitetura quanto o comportamento dos barramentos de comunicação aqui estudados foram especificados e validados de forma satisfatória com as linguagens escolhidas, os objetivos deste trabalho foram alcançados, provendo ao projeto TOPCASED uma abordagem de projeto e validação de protocolos síncronos.

A modelagem da arquitetura de sistemas com barramento foi realizada com a linguagem AADL, permitindo que elementos de hardware e software fossem representados com componentes que se assemelham à sua forma real (como processadores, memórias e processos de software), facilitando a especificação e compreensão dos sistemas. Visto que a semântica de algumas propriedades AADL ainda não está completamente formalizada nas ferramentas atualmente disponíveis, um modelo simples para expressão de dependências e restrições de propriedades AADL foi criado com o método Event-B, conhecido por seu rigor na especificação e verificação de propriedades.

O estudo e utilização da linguagem LUSTRE numa especificação comportamental abstrata foi de interesse para a compreensão do funcionamento de protocolos (e sistemas em geral) síncronos. Este trabalho estuda protocolos síncronos de barramento com arbitração centralizada dada sua ampla aplicação em sistemas embarcados.

O emprego do método Event-B para a modelagem e verificação parametrizada do comportamento dos protocolos trouxe diversas vantagens: a necessidade de realizar uma verificação parametrizada não implicou qualquer limitação para a modelagem, as provas

formais necessárias foram geradas e parcialmente provadas automaticamente, e a possibilidade de especificar os protocolos através de refinamentos sucessivos permitiu que as propriedades desejadas fossem expressas num nível abstrato de especificação e mantidas nos modelos concretos, mais complexos. O trabalho adicional necessário para representar protocolos síncronos num ambiente assíncrono foi bastante reduzido através da criação do modelo abstrato visto na seção 8.2.1, que realiza a sincronização dos eventos individuais de cada controlador e do árbitro. Neste sentido, a principal contribuição deste trabalho consiste neste modelo abstrato que é genérico o suficiente para ser reutilizado em outras especificações de protocolos síncronos com arbitração centralizada. Outros sistemas síncronos, tais como circuitos eletrônicos, podem ter seu comportamento especificado e verificado em Event-B de modo similar: as máquinas abstratas serão diferentes mas possivelmente vão conter eventos semelhantes ao evento sincronizador *controllers_sync* visto no capítulo 8.

Como limitações deste trabalho, verifica-se que os resultados aqui obtidos podem ainda ser enriquecidos com um estudo mais aprofundado de um protocolo de barramento em um sistema a ser desenvolvido e que necessita de análises de arquitetura e comportamento. O poder de expressão da AADL poderia ser usado para analisar, por exemplo, se um mapeamento especificado garante que todos os barramentos vão operar dentro de seus limites de largura de banda. O comportamento validado em Event-B pode ser ainda mais refinado, para incluir aspectos próximos à implementação, tais como a política de arbitração. O estudo de políticas de arbitração combinado com a especificação de requisitos e prioridades para cada controlador pode culminar com novas propriedades a serem verificadas.

Trabalhos futuros na área de arquiteturas de barramentos podem abordar o tema de parametrização de componentes nas especificações AADL e também na criação de um *framework* completo para formalização da semântica das propriedades. Um fator adicional para motivar a parametrização de componentes AADL é a utilização do Anexo Comportamental para a descrição (e possível verificação) do comportamento dos componentes.

Na área de verificação formal, um tema interessante para trabalhos futuros é a busca por maior automatização nas provas de teoremas, seja com provadores mais potentes, seja com a criação de caminhos de prova típicos para propriedades, como a exclusão mútua. Assim, poderia-se utilizar táticas pré-definidas de provas para reduzir a interação do usuário com o provador.

Parte V

Apêndices

APÊNDICE A – Especificações AADL

A.1 Exemplo de *flow*

```
system big
end big;

system implementation big.i
 subcomponents
  first: system flowsource.i;
  second: system flowpath.i;
  third: system flowsink.i;
 connections
  source2path: event port first.exit -> second.entry;
  path2sink: event port second.exit -> third.entry;
 flows
  event_passage: end to end flow first.start ->
  source2path -> second.going -> path2sink ->
  third.finish;
end big.i;

system flowsource
 features
  exit: out event port;
 flows
  start: flow source exit;
end flowsource;

system implementation flowsource.i
 flows
  start: flow source exit;
end flowsource.i;

system flowsink
 features
  entry: in event port;
 flows
  finish: flow sink entry;
end flowsink;

system implementation flowsink.i
 flows
  finish: flow sink entry;
```

```
end flowsink.i;

system flowpath
 features
  entry: in event port;
  exit: out event port;
 flows
  going: flow path entry -> exit;
end flowpath;

system implementation flowpath.i
 subcomponents
  subsubsystem: system subsubsystem.i;
 connections
  tosubentry: event port entry -> subsubsystem.entry;
  fromsubexit: event port subsubsystem.exit -> exit;
 flows
  going: flow path entry -> tosubentry ->
  subsubsystem.going -> fromsubexit -> exit;
end flowpath.i;

system subsubsystem
 features
  entry: in event port;
  exit: out event port;
 flows
  going: flow path entry -> exit;
end subsubsystem;

system implementation subsubsystem.i
 flows
  going: flow path entry -> exit;
end subsubsystem.i;
```

A.2 Arquitetura do Barramento PCI

A.2.1 Nível *aadlpci0*

```
bus pci
  properties
  Allowed_Connection_Protocol => Data_Connection;
  Allowed_Access_Protocol => (Device_Access, Memory_Access);
  Hardware_Source_Language => VHDL;
end pci;

bus implementation pci.b32bits
  properties
  Allowed_Message_Size => 1 B .. 4 B;
end pci.b32bits;

bus implementation pci.b64bits
  properties
  Allowed_Message_Size => 1 B .. 8 B;
```

```
end pci.b64bits;

system computer
end computer;

system implementation computer.old
  subcomponents
    video_controller: device generic;
    audio_controller: device generic;
    system_bus: bus pci.b32bits;
    bridge: bus generic_bus;
    peripheral: bus generic_bus;
    proc: processor generic_proc;
    mem: memory generic_mem;
    monitor: device generic;
    keyboard: device generic;
    mouse: device generic;

  connections
    bus access system_bus -> video_controller.IO;
    bus access system_bus -> audio_controller.IO;
    bus access system_bus -> bridge.IO;
    bus access system_bus -> peripheral.IO;
    bus access bridge -> proc.IO;
    bus access bridge -> mem.IO;
    bus access peripheral -> monitor.IO;
    bus access peripheral -> keyboard.IO;
    bus access peripheral -> mouse.IO;
end computer.old;

device generic
  features
    IO: requires bus access;
end generic;

bus generic_bus
  features
    IO: requires bus access pci.b32bits;
end generic_bus;

processor generic_proc
  features
    IO: requires bus access generic_bus;
end generic_proc;

memory generic_mem
  features
    IO: requires bus access generic_bus;
end generic_mem;
```

A.2.2 Nível *aadlpci1*

```
system pci_arbiter
  features
```

```
    frame: requires data access behavior::boolean {
      Required_Access => access read_only;  };
    req_1: requires data access behavior::boolean {
      Required_Access => access read_only; };
    req_2: requires data access behavior::boolean {
      Required_Access => access read_only; };
    gnt_1: requires data access behavior::boolean {
      Required_Access => access read_write; };
    gnt_2: requires data access behavior::boolean {
      Required_Access => access read_write; };
    irdy: requires data access behavior::boolean {
      Required_Access => access read_only; };
    trdy: requires data access behavior::boolean {
      Required_Access => access read_only; };
    devsel: requires data access behavior::boolean {
      Required_Access => access read_only; };
    ad: requires data access behavior::integer {
      Required_Access => access read_only; };
    command: requires data access data_types::command {
      Required_Access => access read_only; };
end pci_arbiter;

system pci_bus
  features
    frame: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    req_1: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    req_2: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    gnt_1: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    gnt_2: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    irdy: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    trdy: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    devsel: provides data access behavior::boolean {
      Provided_Access => access read_write;  };
    ad: provides data access behavior::integer {
      Provided_Access => access read_write;  };
    command: provides data access data_types::command {
      Provided_Access => access read_write;  };
end pci_bus;

system pci_device
  features
    frame: requires data access behavior::boolean {
      Required_Access => access read_write; };
    req: requires data access behavior::boolean {
      Required_Access => access read_write; };
    gnt: requires data access behavior::boolean {
      Required_Access => access read_only; };
```

```
    irdy: requires data access behavior::boolean {
      Required_Access => access read_write; };
    trdy: requires data access behavior::boolean {
      Required_Access => access read_write; };
    devsel: requires data access behavior::boolean {
      Required_Access => access read_write; };
    ad: requires data access behavior::integer {
      Required_Access => access read_write; };
    command: requires data access data_types::command {
      Required_Access => access read_write;  };
end pci_device;
```

APÊNDICE B – Especificações Event-B

B.1 Base para Protocolos Síncronos

B.1.1 Contexto genérico

CONTEXT synchronous_c0

SETS

$CONTROLLERS$
CTR_STATES

AXIOMS

$axm1 : CONTROLLERS \rightarrow CTR\_STATES \neq \varnothing$

END

B.1.2 Máquina Síncrona

MACHINE synchronous_m0

SEES synchronous_c0

VARIABLES

$finishedArb$
$gstateCtr$
$pregstateCtr$
$arbbusy$
$prearbbusy$

INVARIANTS

$typinv\_fa : finishedArb \in BOOL$
$typinv\_gstate : gstateCtr \in CONTROLLERS \nrightarrow CTR\_STATES$
$typinv\_arbbusy : arbbusy \in BOOL$

$typinv\_pgstate : pregstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
$typinv\_parbbusy : prearbbusy \in BOOL$

EVENTS

INITIALISATION

BEGIN
$ini\_parbbusy : prearbbusy := FALSE$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_pgstate : pregstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
$ini\_gstate : gstateCtr := \varnothing$
$ini\_fa : finishedArb := FALSE$
END

EVENT next_step
WHEN
$arb\_done : finishedArb = TRUE$
$ctr\_sync : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
THEN
$update\_states : pregstateCtr := gstateCtr$
$update\_arb : prearbbusy := arbbusy$
$reset\_gstate : gstateCtr := \varnothing$
$reset\_fa : finishedArb := FALSE$
END

EVENT generic_arbiter_action
WHEN
$arb\_rdy : finishedArb = FALSE$
THEN
$arb\_done : finishedArb := TRUE$
$action\_taken : arbbusy :\in BOOL$
END

EVENT controllers_sync
WHEN
$not\_sync : gstateCtr = \varnothing$
THEN
$ctr\_sync : gstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
END

END

B.1.3 Refinamento para Eventos Assíncronos

MACHINE synchronous_m1

REFINES synchronous_m0

SEES synchronous_c0

VARIABLES

$finishedArb$
$stateCtr$
$arbbusy$
$prearbbusy$
$pregstateCtr$
$gstateCtr$

INVARIANTS

$typinv\_stateCtr : stateCtr \in CONTROLLERS \pfun CTR\_STATES$

EVENTS

INITIALISATION

BEGIN
$ini\_fa : finishedArb := FALSE$
$ini\_state : stateCtr := \varnothing$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_prearbbusy : prearbbusy := FALSE$
$ini\_pgstate : pregstateCtr :\in CONTROLLERS \to CTR\_STATES$
$ini\_gstate : gstateCtr := \varnothing$
END

EVENT generic_controller_action

ANY
C
ns
WHERE
$some\_ctrls : C \in \mathbb{P}(CONTROLLERS)$
$didnt\_act : C \cap dom(stateCtr) = \varnothing$
$new\_states : ns \in C \to CTR\_STATES$
THEN
$statechange : stateCtr := stateCtr \cup ns$
END

EVENT next_step
REFINES next_step
WHEN
$actions\_taken : gstateCtr \in CONTROLLERS \to CTR\_STATES$
$arb\_done : finishedArb = TRUE$
THEN
$arb\_ready : finishedArb := FALSE$
$update\_states : pregstateCtr := gstateCtr$
$update\_arbiter : prearbbusy := arbbusy$
$reset\_state : stateCtr := \varnothing$
$reset\_gstate : gstateCtr := \varnothing$
END

EVENT generic_arbiter_action
REFINES generic_arbiter_action

WHEN

$arb\_ready : finishedArb = FALSE$

THEN

$arb\_done : finishedArb := TRUE$

$update\_status : arbbusy :\in BOOL$

END

EVENT controllers_sync

REFINES controllers_sync

WHEN

$ctr\_done : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

$not\_sync : gstateCtr = \varnothing$

THEN

$ctr\_sync : gstateCtr := stateCtr$

END

END

B.1.4 Contexto com os estados abstratos do protocolo

CONTEXT pci_amba_c0

REFINES synchronous_c0

CONSTANTS

$idle$

$request$

$master$

AXIOMS

$axm1 : CTR\_STATES = \{idle, request, master\}$

$axm2 : idle \neq request$

$axm3 : idle \neq master$

$axm4 : request \neq master$

END

B.1.5 Nível 0 – Especificação do Invariante

MACHINE pci_amba_m0

REFINES synchronous_m1

SEES pci_amba_c0

VARIABLES

$finishedArb$
$stateCtr$
$arbbusy$
$prearbbusy$
$owner$
$preowner$
$gstateCtr$
$pregstateCtr$

INVARIANTS

$typinv\_owner : owner \in CONTROLLERS$
$typinv\_preowner : preowner \in CONTROLLERS$
$singlemaster : stateCtr^{-1}[\{master\}] \subseteq \{preowner\}$

EVENTS

INITIALISATION

BEGIN
$ini\_fa : finishedArb := FALSE$
$ini\_state : stateCtr := \varnothing$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_prearbbusy : prearbbusy := FALSE$
$ini\_owner : owner :\in CONTROLLERS$
$ini\_preowner : preowner :\in CONTROLLERS$
$ini\_pgstate : pregstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
$ini\_gstate : gstateCtr := \varnothing$
END

EVENT generic_controller_action

REFINES generic_controller_action
ANY
C
ns
WHERE
$some\_ctrls : C \in \mathbb{P}(CONTROLLERS)$
$didnt\_act : C \cap dom(stateCtr) = \varnothing$
$new\_states : ns \in C \rightarrow CTR\_STATES$
$correct\_master : ns^{-1}[\{master\}] \subseteq \{preowner\}$
THEN
$statechange : stateCtr := stateCtr \cup ns$
END

EVENT next_step
REFINES next_step
WHEN
$actions\_taken : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
$arb\_done : finishedArb = TRUE$
THEN
$arb\_ready : finishedArb := FALSE$

$update\_states : pregstateCtr := gstateCtr$
$update\_arbiter : prearbbusy := arbbusy$
$reset\_states : stateCtr := \varnothing$
$update\_owner : preowner := owner$
$reset\_gstate : gstateCtr := \varnothing$
END

EVENT generic_arbiter_action
REFINES generic_arbiter_action
WHEN
$arb\_ready : finishedArb = FALSE$
THEN
$arb\_done : finishedArb := TRUE$
$update\_status : arbbusy :\in BOOL$
$update\_owner : owner :\in CONTROLLERS$
END

EVENT controllers_sync
REFINES controllers_sync
WHEN
$ctr\_done : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
$not\_sync : gstateCtr = \varnothing$
THEN
$ctr\_sync : gstateCtr := stateCtr$
END

END

B.2 O Protocolo PCI

B.2.1 Nível 1 – Especificação Abstrata

MACHINE pci_m1

REFINES pci_amba_m0

SEES pci_amba_c0

VARIABLES

$prearbbusy$
$preowner$
$stateCtr$
$owner$
$arbbusy$
$finishedArb$
$pregstateCtr$
$gstateCtr$

INVARIANTS

$typinv\_preho : prearbbusy \in BOOL$
$typinv\_po : preowner \in CONTROLLERS$
$typinv\_owner : owner \in CONTROLLERS$
$typinv\_hasowner : arbbusy \in BOOL$
$singlemaster : prearbbusy = TRUE \land preowner \in dom(stateCtr) \Rightarrow (stateCtr^{-1}[\{master\}]) = \{preowner\}$
$nomaster : prearbbusy = FALSE \Rightarrow (stateCtr^{-1}[\{master\}]) = \varnothing$
$singlemaster2 : (prearbbusy = TRUE \land preowner \notin dom(stateCtr)) \Rightarrow (stateCtr^{-1}[\{master\}]) = \varnothing$
$inv1 : prearbbusy = TRUE \Rightarrow pregstateCtr(preowner) \in \{request, master\}$
$inv2 : arbbusy = TRUE \land finishedArb = TRUE \Rightarrow pregstateCtr(owner) \in \{request, master\}$
$inv3 : \forall c \cdot (c \in CONTROLLERS \land c \in dom(stateCtr) \land pregstateCtr(c) = idle \Rightarrow stateCtr(c) \in \{idle, request\})$
$inv4 : \forall c \cdot (c \in CONTROLLERS \land c \in dom(stateCtr) \land pregstateCtr(c) = request \Rightarrow stateCtr(c) \in \{master, request\})$
$inv5 : \forall c \cdot (c \in CONTROLLERS \land c \in dom(stateCtr) \land pregstateCtr(c) = master \Rightarrow stateCtr(c) \in \{idle, master\})$
$inv6 : stateCtr^{-1}[\{master\}] \subseteq \{preowner\}$
$inv7 : arbbusy = TRUE \land prearbbusy = TRUE \land owner \neq preowner \Rightarrow pregstateCtr(owner) = request$
$inv8 : arbbusy = TRUE \land prearbbusy = TRUE \land finishedArb = TRUE \land owner = preowner \Rightarrow pregstateCtr(owner) \in \{request, master\}$
$inv9 : arbbusy = TRUE \land prearbbusy = FALSE \Rightarrow pregstateCtr(owner) = request$
$inv10 : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES \Rightarrow gstateCtr = stateCtr$

THEOREMS

$singlemastertheorem : dom(stateCtr) = CONTROLLERS \land prearbbusy = TRUE \Rightarrow stateCtr^{-1}[\{master\}] \subseteq \{preowner\}$
$nomastertheorem : dom(stateCtr) = CONTROLLERS \land prearbbusy = FALSE \Rightarrow stateCtr^{-1}[\{master\}] = \varnothing$

EVENTS

INITIALISATION

BEGIN
$ini\_pregstateCtr : pregstateCtr := CONTROLLERS \times \{idle\}$
$ini\_stateCtr : stateCtr := \varnothing$
$ini\_owner : owner :\in CONTROLLERS$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_fa : finishedArb := FALSE$
$ini\_preho : prearbbusy := FALSE$
$ini\_po : preowner :\in CONTROLLERS$
$act1 : gstateCtr := \varnothing$
END

EVENT idle2req

REFINES generic_controller_action
ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_idle : pregstateCtr(c) = idle$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto request\}$

THEN

$going\_req : stateCtr := stateCtr \cup \{c \mapsto request\}$

END

EVENT req2master

REFINES generic_controller_action

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_req : pregstateCtr(c) = request$

$is\_owner : prearbbusy = TRUE \land preowner = c$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto master\}$

THEN

$going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$

END

EVENT master2idle

REFINES generic_controller_action

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_master : pregstateCtr(c) = master$

$is\_not\_owner : \neg(prearbbusy = TRUE \land preowner = c)$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto idle\}$

THEN

$going\_idle : stateCtr := stateCtr \cup \{c \mapsto idle\}$

END

EVENT idle2busy

REFINES generic_arbiter_action

ANY

c

WHERE

$arb\_ready : finishedArb = FALSE$

$is\_idle : prearbbusy = FALSE$

$a\_req : c \in pregstateCtr^{-1}[\{request\}]$

THEN

$arb\_done : finishedArb := TRUE$

$going\_busy : arbbusy := TRUE$

$newowner : owner := c$

END

EVENT busy2busy

REFINES generic_arbiter_action

ANY

$c2$

WHERE

$arb\_ready : finishedArb = FALSE$

$old\_owner : prearbbusy = TRUE$

$a\_req : c2 \in pregstateCtr^{-1}[\{request\}]$

THEN

$arb\_done : finishedArb := TRUE$

$going\_busy : arbbusy := TRUE$

$newowner : owner := c2$

END

EVENT busy2idle

REFINES generic_arbiter_action

WHEN

$arb\_ready : finishedArb = FALSE$

$oldowner : prearbbusy = TRUE$

$no\_reqs : pregstateCtr^{-1}[\{request\}] = \varnothing$

THEN

$arb\_done : finishedArb := TRUE$

$going\_idle : arbbusy := FALSE$

$dontcare : owner :\in CONTROLLERS$

END

EVENT defaultArbEvent

REFINES generic_arbiter_action

WHEN

$defaultGuard : (prearbbusy = FALSE \wedge pregstateCtr^{-1}[\{request\}] = \varnothing)$
$\vee (prearbbusy = TRUE)$

$arb\_ready : finishedArb = FALSE$

THEN

$owner\_unchanged : owner := preowner$

$arbbusy\_unchanged : arbbusy := prearbbusy$

$arb\_done : finishedArb := TRUE$

END

EVENT defaultCtrlEvent

REFINES generic_controller_action

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$defaultGuard : pregstateCtr(c) = idle \vee$
$(pregstateCtr(c) = request \wedge \neg(prearbbusy = TRUE \wedge preowner = c)) \vee$
$(pregstateCtr(c) = master \wedge prearbbusy = TRUE \wedge preowner = c)$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto pregstateCtr(c)\}$

THEN

$samestate : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

END

EVENT clock_cycle

REFINES next_step

WHEN

$arb\_ok : finishedArb = TRUE$

$ctrl\_ok : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

THEN

$update\_owner : preowner := owner$

$update\_hasowner : prearbbusy := arbbusy$

$update\_states : pregstateCtr := gstateCtr$

$reset\_arb : finishedArb := FALSE$

$reset\_states : stateCtr := \varnothing$

$reset\_gstate : gstateCtr := \varnothing$

END

EVENT controllers_sync

REFINES controllers_sync

WHEN

$ctr\_done : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

$not\_sync : gstateCtr = \varnothing$

THEN

$ctr\_sync : gstateCtr := stateCtr$

END

END

B.2.2 Nível 2 – Especificação concreta da arbitração

MACHINE pci_m2

REFINES pci_m1

SEES pci_amba_c0

VARIABLES

$gstateCtr$

$pregstateCtr$

$prearbbusy$

$preowner$

$stateCtr$

$owner$
$arbbusy$
$finishedArb$
REQ
GNT
$preREQ$
$preGNT$

INVARIANTS

$typinv\_req : REQ \in CONTROLLERS \rightarrow BOOL$
$typinv\_gnt : GNT \in CONTROLLERS \rightarrow BOOL$
$typinv\_prereq : preREQ \in CONTROLLERS \rightarrow BOOL$
$typinv\_pregnt : preGNT \in CONTROLLERS \rightarrow BOOL$
$nomaster\_concrete : (arbbusy = FALSE \Leftrightarrow GNT^{-1}[\{TRUE\}] = \varnothing) \wedge (prearbbusy = FALSE \Leftrightarrow preGNT^{-1}[\{TRUE\}] = \varnothing)$
$singlemaster\_concrete : (arbbusy = TRUE \Leftrightarrow GNT^{-1}[\{TRUE\}] = \{owner\}) \wedge (prearbbusy = TRUE \Leftrightarrow preGNT^{-1}[\{TRUE\}] = \{preowner\})$

EVENTS

INITIALISATION

BEGIN
$ini_gstate : gstateCtr := \varnothing$
$ini\_pgstate : pregstateCtr := CONTROLLERS \times \{idle\}$
$ini\_state : stateCtr := \varnothing$
$ini\_owner : owner :\in CONTROLLERS$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_aw : finishedArb := FALSE$
$ini\_preab : prearbbusy := FALSE$
$ini\_po : preowner :\in CONTROLLERS$
$ini\_req : REQ := CONTROLLERS \times \{FALSE\}$
$ini\_gnt : GNT := CONTROLLERS \times \{FALSE\}$
$ini\_prereq : preREQ := CONTROLLERS \times \{FALSE\}$
$ini\_pregnt : preGNT := CONTROLLERS \times \{FALSE\}$
END

EVENT idle2req

REFINES idle2req
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_idle : pregstateCtr(c) = idle$
$is\_idle\_concrete : preREQ(c) = FALSE$
THEN
$going\_req : stateCtr := stateCtr \cup \{c \mapsto request\}$
$going\_req\_concrete : REQ := REQ \leftarrow \{c \mapsto TRUE\}$
END

EVENT req2master

REFINES req2master

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_req : pregstateCtr(c) = request$

$is\_owner : prearbbusy = TRUE \land preowner = c$

$is\_req\_concrete : preREQ(c) = TRUE$

$is\_owner\_concrete : preGNT(c) = TRUE$

THEN

$going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$

END

EVENT end_master

REFINES defaultCtrlEvent

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$is\_master : pregstateCtr(c) = master$

$has\_req : preREQ(c) = TRUE$

$didnt\_act : c \notin dom(stateCtr)$

$is\_owner : preowner = c \land prearbbusy = TRUE$

$concrete\_owner : preGNT(c) = TRUE$

THEN

$stop\_req : REQ := REQ \lhd\!\!+ \{c \mapsto FALSE\}$

$keep\_states : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

END

EVENT master2idle

REFINES master2idle

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_master : pregstateCtr(c) = master$

$is\_not\_owner : \neg(prearbbusy = TRUE \land preowner = c)$

$is\_not\_owner\_concrete : preGNT(c) = FALSE$

THEN

$going\_idle : stateCtr := stateCtr \cup \{c \mapsto idle\}$

END

EVENT idle2busy

REFINES idle2busy

ANY

c

WHERE

$arb\_ready : finishedArb = FALSE$

$is\_idle : prearbbusy = FALSE$

$a\_req : c \in pregstateCtr^{-1}[\{request\}]$

$concrete\_req : preREQ(c) = TRUE$

$concrete\_idle\_bus : preGNT^{-1}[\{TRUE\}] = \varnothing$

THEN

$arb\_done : finishedArb := TRUE$

$going\_busy : arbbusy := TRUE$

$newowner : owner := c$

$owner\_concrete : GNT := (CONTROLLERS \times \{FALSE\}) \lhd \{c \mapsto TRUE\}$

END

EVENT busy2busy

REFINES busy2busy

ANY

$c1$

$c2$

WHERE

$arb\_ready : finishedArb = FALSE$

$old\_owner : prearbbusy = TRUE \land preowner = c1$

$a\_req : c2 \in pregstateCtr^{-1}[\{request\}]$

$concrete\_old\_owner : preGNT(c1) = TRUE$

$concrete\_old\_not\_req : preREQ(c1) = FALSE$

$concrete\_new\_req : preREQ(c2) = TRUE$

THEN

$arb\_done : finishedArb := TRUE$

$going\_busy : arbbusy := TRUE$

$newowner : owner := c2$

$concrete\_newowner : GNT := (CONTROLLERS \times \{FALSE\}) \lhd \{c2 \mapsto TRUE\}$

END

EVENT busy2idle

REFINES busy2idle

WHEN

$arb\_ready : finishedArb = FALSE$

$oldowner : prearbbusy = TRUE$

$no\_reqs : pregstateCtr^{-1}[\{request\}] = \varnothing$

$owner\_not\_req : preREQ(preowner) = FALSE$

$oldowner\_concrete : preGNT(preowner) = TRUE$

THEN

$arb\_done : finishedArb := TRUE$

$going\_idle : arbbusy := FALSE$

$dontcare : owner :\in CONTROLLERS$

$concrete\_noowner : GNT := (CONTROLLERS \times \{FALSE\})$

END

EVENT defaultArbEvent

REFINES defaultArbEvent

WHEN

$defaultGuard : (prearbbusy = FALSE \land pregstateCtr^{-1}[\{request\}] = \varnothing \land preREQ^{-1}[\{TRUE\}] = \varnothing) \lor (prearbbusy = TRUE \land preREQ(owner) = TRUE)$

$arb\_ready : finishedArb = FALSE$

THEN

$owner\_unchanged : owner := preowner$

$arbbusy\_unchanged : arbbusy := prearbbusy$

$arb\_done : finishedArb := TRUE$

$concrete\_unchanged : GNT := preGNT$

END

EVENT defaultCtrlEvent

REFINES defaultCtrlEvent

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$defaultGuard : (preREQ(c) = FALSE \wedge preGNT(c) = FALSE \wedge pregstateCtr(c) = idle) \vee (preREQ(c) = TRUE \wedge preGNT(c) = FALSE \wedge pregstateCtr(c) = request \wedge \neg(prearbbusy = TRUE \wedge preowner = c)) \vee (pregstateCtr(c) = master \wedge prearbbusy = TRUE \wedge preowner = c \wedge preREQ(c) = TRUE \wedge preGNT(c) = TRUE)$

THEN

$samestate : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

$samereq : REQ := REQ$

END

EVENT clock_cycle

REFINES clock_cycle

WHEN

$arb\_ok : finishedArb = TRUE$

$ctrl\_ok : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

THEN

$update\_owner : preowner := owner$

$update\_arbbusy : prearbbusy := arbbusy$

$update\_states : pregstateCtr := gstateCtr$

$reset\_arb : finishedArb := FALSE$

$update\_REQ : preREQ := REQ$

$update\_GNT : preGNT := GNT$

$reset\_states : stateCtr := \varnothing$

$reset\_gstate : gstateCtr := \varnothing$

END

EVENT controllers_sync

REFINES controllers_sync

WHEN

$grd1 : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

$grd2 : gstateCtr = \varnothing$

THEN

$act1 : gstateCtr := stateCtr$

END

END

B.2.3 Contexto da especificação concreta

CONTEXT pci_c1

REFINES pci_amba_c0

SETS

$COMMANDS$

CONSTANTS

$read$
$write$

AXIOMS

$axm1 : COMMANDS = \{read, write\}$
$axm2 : read \neq write$
$axm3 : CONTROLLERS \rightarrow \mathbb{N} \neq \varnothing$
$axm4 : \forall c \cdot (c \in CONTROLLERS \Rightarrow CONTROLLERS \setminus \{c\} \neq \varnothing)$

END

B.2.4 Nível 3 – Especificação concreta da transferência

MACHINE pci_m3

REFINES pci_m2

SEES pci_c1

VARIABLES

$gstateCtr$
$pregstateCtr$
$prearbbusy$
$preowner$
$stateCtr$
$owner$
$arbbusy$
$finishedArb$
REQ
GNT
$preREQ$
$preGNT$
$DEVSEL$
$IRDY$
$TRDY$
$preDEVSEL$

$preIRDY$
$preTRDY$
$command$
$precommand$
$busdata$
$prebusdata$
$buffers$
$prebuffers$
$FRAME$
$preFRAME$
$address$
$preaddress$
$hasAddress$
$prehasAddress$
$slave$
$preslave$

INVARIANTS

$typinv\_devsel : DEVSEL \in BOOL$
$typinv\_irdy : IRDY \in BOOL$
$typinv\_trdy : TRDY \in BOOL$
$typinv\_predevsel : preDEVSEL \in BOOL$
$typinv\_preirdy : preIRDY \in BOOL$
$typinv\_pretrdy : preTRDY \in BOOL$
$typinv\_command : command \in COMMANDS$
$typinv\_precommand : precommand \in COMMANDS$
$typinv\_busdata : busdata \in \mathbb{N}$
$typinv\_prebusdata : prebusdata \in \mathbb{N}$
$typinv\_buffers : buffers \in CONTROLLERS \rightarrow \mathbb{N}$
$typinv\_prebuffers : prebuffers \in CONTROLLERS \rightarrow \mathbb{N}$
$typinv\_frame : FRAME \in BOOL$
$typinv\_preframe : preFRAME \in BOOL$
$typinv\_address : address \in CONTROLLERS$
$typinv\_preaddress : preaddress \in CONTROLLERS$
$typinv\_hasaddress : hasAddress \in BOOL$
$typinv\_preha : prehasAddress \in BOOL$
$typinv\_slave : slave \in CONTROLLERS \rightarrow BOOL$
$typinv\_preslave : preslave \in CONTROLLERS \rightarrow BOOL$

EVENTS

INITIALISATION

BEGIN

$act1 : gstateCtr := \varnothing$
$ini\_pregstateCtr : pregstateCtr := CONTROLLERS \times \{idle\}$
$ini\_stateCtr : stateCtr := \varnothing$
$ini\_owner : owner :\in CONTROLLERS$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_aw : finishedArb := FALSE$
$ini\_preab : prearbbusy := FALSE$
$ini\_po : preowner :\in CONTROLLERS$
$ini\_req : REQ := CONTROLLERS \times \{FALSE\}$

$ini\_gnt : GNT := CONTROLLERS \times \{FALSE\}$
$ini\_prereq : preREQ := CONTROLLERS \times \{FALSE\}$
$ini\_pregnt : preGNT := CONTROLLERS \times \{FALSE\}$
$ini\_devsel : DEVSEL := FALSE$
$ini\_irdy : IRDY := FALSE$
$ini\_trdy : TRDY := FALSE$
$ini\_predevsel : preDEVSEL := FALSE$
$ini\_preirdy : preIRDY := FALSE$
$ini\_pretrdy : preTRDY := FALSE$
$ini\_command : command :\in COMMANDS$
$ini\_precommand : precommand :\in COMMANDS$
$ini\_busdata : busdata :\in \mathbb{N}$
$ini\_prebusdata : prebusdata :\in \mathbb{N}$
$ini\_buffers : buffers :\in CONTROLLERS \rightarrow \mathbb{N}$
$ini\_prebuffers : prebuffers :\in CONTROLLERS \rightarrow \mathbb{N}$
$iniframe : FRAME := FALSE$
$ini\_preframe : preFRAME := FALSE$
$ini\_address : address :\in CONTROLLERS$
$ini\_preaddress : preaddress :\in CONTROLLERS$
$ini\_hasaddress : hasAddress := FALSE$
$ini\_preha : prehasAddress := FALSE$
$ini\_slave : slave := CONTROLLERS \times \{FALSE\}$
$ini\_preslave : preslave := CONTROLLERS \times \{FALSE\}$

END

EVENT idle2req

REFINES idle2req

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_idle : pregstateCtr(c) = idle$
$is\_idle\_concrete : preREQ(c) = FALSE$

THEN

$going\_req : stateCtr := stateCtr \cup \{c \mapsto request\}$
$going\_req\_concrete : REQ := REQ \leftarrow \{c \mapsto TRUE\}$

END

EVENT req2write

REFINES req2master

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_req : pregstateCtr(c) = request$
$is\_owner : prearbbusy = TRUE \wedge preowner = c$
$is\_req\_concrete : preREQ(c) = TRUE$
$is\_owner\_concrete : preGNT(c) = TRUE$

THEN

$going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$

$start\_writing : command := write$
$call\_slave : address :\in (CONTROLLERS \setminus \{c\})$
$frame\_on : FRAME := TRUE$
$addressphase : hasAddress := TRUE$
END

EVENT req2read
REFINES req2master
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_req : pregstateCtr(c) = request$
$is\_owner : prearbbusy = TRUE \land preowner = c$
$is\_req\_concrete : preREQ(c) = TRUE$
$is\_owner\_concrete : preGNT(c) = TRUE$
THEN
$going\_master : stateCtr := stateCtr \cup \{c \mapsto master\}$
$start\_reading : command := read$
$call\_slave : address :\in (CONTROLLERS \setminus \{c\})$
$frame\_on : FRAME := TRUE$
$addressphase : hasAddress := TRUE$
END

EVENT normal_master_write
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_master : pregstateCtr(c) = master$
$is\_owner : prearbbusy = TRUE \land preowner = c$
$has\_request : preREQ(c) = TRUE$
$is\_writer : precommand = write$
$a\_slave\_or\_start : (preDEVSEL = TRUE \land preTRDY = TRUE) \lor preIRDY = FALSE$
$concrete\_owner : preGNT(c) = TRUE$
THEN
$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
$write\_data : busdata :\in \mathbb{N}$
$irdy : IRDY := TRUE$
$dataphase : hasAddress := FALSE$
END

EVENT master_read_turnaround
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$is\_master : pregstateCtr(c) = master$
$is\_owner : prearbbusy = TRUE \land preowner = c$

$has\_request : preREQ(c) = TRUE$
$is\_reader : precommand = read$
$starting : preDEVSEL = FALSE$
$concrete\_owner : preGNT(c) = TRUE$
$didnt\_act : c \notin dom(stateCtr)$

THEN

$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
$irdy : IRDY := TRUE$
$dataphase : hasAddress := FALSE$

END

EVENT normal_master_read
REFINES defaultCtrlEvent

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$is\_master : pregstateCtr(c) = master$
$is\_owner : prearbbusy = TRUE \land preowner = c$
$has\_request : preREQ(c) = TRUE$
$is\_reader : precommand = read$
$concrete\_owner : preGNT(c) = TRUE$
$trdy : preTRDY = TRUE$

THEN

$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
$irdy : IRDY := TRUE$
$save\_data : buffers(c) := prebusdata$

END

EVENT next_to_last_write
REFINES end_master

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$
$is\_master : pregstateCtr(c) = master$
$has\_req : preREQ(c) = TRUE$
$didnt\_act : c \notin dom(stateCtr)$
$is\_owner : preowner = c \land prearbbusy = TRUE$
$concrete\_owner : preGNT(c) = TRUE$
$is\_writer : precommand = write$
$trdy : preTRDY = TRUE$

THEN

$stop\_req : REQ := REQ \lhd\!\!\!\!- \{c \mapsto FALSE\}$
$keep\_stateCtrs : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$write\_data : busdata :\in \mathbb{N}$
$irdy : IRDY := TRUE$

END

EVENT next_to_last_read
REFINES end_master

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$is\_master : pregstateCtr(c) = master$

$has\_req : preREQ(c) = TRUE$

$didnt\_act : c \notin dom(stateCtr)$

$is\_owner : preowner = c \wedge prearbbusy = TRUE$

$concrete\_owner : preGNT(c) = TRUE$

$trdy : preTRDY = TRUE$

THEN

$stop\_req : REQ := REQ \lhd\!\!\!- \{c \mapsto FALSE\}$

$keep\_stateCtrs : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

$irdy : IRDY := TRUE$

$save\_data : buffers(c) := prebusdata$

END

EVENT write2idle

REFINES master2idle

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_master : pregstateCtr(c) = master$

$is\_not\_owner : \neg(prearbbusy = TRUE \wedge preowner = c)$

$is\_not\_owner\_concrete : preGNT(c) = FALSE$

$trdy : preTRDY = TRUE$

THEN

$going\_idle : stateCtr := stateCtr \cup \{c \mapsto idle\}$

$write\_data : busdata :\in \mathbb{N}$

$irdy : IRDY := TRUE$

$end\_of\_transfer : FRAME := FALSE$

END

EVENT read2idle

REFINES master2idle

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$is\_master : pregstateCtr(c) = master$

$is\_not\_owner : \neg(prearbbusy = TRUE \wedge preowner = c)$

$is\_not\_owner\_concrete : preGNT(c) = FALSE$

$is\_reader : precommand = read$

THEN

$going\_idle : stateCtr := stateCtr \cup \{c \mapsto idle\}$

$save\_data : buffers(c) := prebusdata$

$irdy : IRDY := TRUE$

$end\_of\_transfer : FRAME := FALSE$

END

EVENT idle2busy

REFINES idle2busy

ANY

c

WHERE

$didnt\_act : finishedArb = FALSE$

$is\_idle : prearbbusy = FALSE$

$a\_req : c \in pregstateCtr^{-1}[\{request\}]$

$concrete\_req : preREQ(c) = TRUE$

$concrete\_idle\_bus : preGNT^{-1}[\{TRUE\}] = \varnothing$

THEN

$action\_taken : finishedArb := TRUE$

$arbbusy : arbbusy := TRUE$

$newowner : owner := c$

$owner\_concrete : GNT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{c \mapsto TRUE\}$

END

EVENT busy2busy

REFINES busy2busy

ANY

$c1$

$c2$

WHERE

$didnt\_act : finishedArb = FALSE$

$old\_owner : prearbbusy = TRUE \land preowner = c1$

$a\_req : c2 \in pregstateCtr^{-1}[\{request\}]$

$concrete\_old\_owner : preGNT(c1) = TRUE$

$concrete\_old\_not\_req : preREQ(c1) = FALSE$

$concrete\_new\_req : preREQ(c2) = TRUE$

THEN

$action\_taken : finishedArb := TRUE$

$arbbusy : arbbusy := TRUE$

$newowner : owner := c2$

$concrete\_newowner : GNT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{c2 \mapsto TRUE\}$

END

EVENT busy2idle

REFINES busy2idle

WHEN

$didnt\_act : finishedArb = FALSE$

$oldowner : prearbbusy = TRUE$

$no\_reqs : pregstateCtr^{-1}[\{request\}] = \varnothing$

$owner\_not\_req : preREQ(preowner) = FALSE$

$oldowner\_concrete : preGNT(preowner) = TRUE$

THEN

$action\_taken : finishedArb := TRUE$

$noowner : arbbusy := FALSE$

$dontcare : owner :\in CONTROLLERS$

$concrete\_noowner : GNT := (CONTROLLERS \times \{FALSE\})$

END

EVENT defaultArbEvent

REFINES defaultArbEvent

WHEN

$defaultGuard$: $(prearbbusy = FALSE \wedge pregstateCtr^{-1}[\{request\}] = \varnothing \wedge preREQ^{-1}[\{TRUE\}] = \varnothing) \vee (prearbbusy = TRUE \wedge preREQ(owner) = TRUE)$

$didnt\_act$: $finishedArb = FALSE$

THEN

$owner\_unchanged$: $owner := preowner$

$arbbusy\_unchanged$: $arbbusy := prearbbusy$

$action\_taken$: $finishedArb := TRUE$

$concrete\_unchanged$: $GNT := preGNT$

END

EVENT defaultCtrlEvent

REFINES defaultCtrlEvent

ANY

c

WHERE

a_ctrl : $c \in CONTROLLERS$

$didnt\_act$: $c \notin dom(stateCtr)$

$idleorreq$: $(preREQ(c) = FALSE \wedge preGNT(c) = FALSE \wedge pregstateCtr(c) = idle) \vee (preREQ(c) = TRUE \wedge preGNT(c) = FALSE \wedge pregstateCtr(c) = request \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$

$notcalled$: $\neg(prehasAddress = TRUE \wedge preaddress = c)$

THEN

$samestateCtr$: $stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

$samereq$: $REQ := REQ$

END

EVENT clock_cycle

REFINES clock_cycle

WHEN

arb_ok : $finishedArb = TRUE$

$ctrl\_ok$: $gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

THEN

$update\_owner$: $preowner := owner$

$update\_arbbusy$: $prearbbusy := arbbusy$

$update\_stateCtrs$: $pregstateCtr := gstateCtr$

$reset\_arb$: $finishedArb := FALSE$

$update\_REQ$: $preREQ := REQ$

$update\_GNT$: $preGNT := GNT$

$update\_IRDY$: $preIRDY := IRDY$

$update\_TRDY$: $preTRDY := TRDY$

$update\_address$: $preaddress := address$

$update\_hasAddress$: $prehasAddress := hasAddress$

$update\_DEVSEL$: $preDEVSEL := DEVSEL$

$update\_buffers$: $prebuffers := buffers$

$update\_busdata$: $prebusdata := busdata$

$update\_FRAME$: $preFRAME := FRAME$

$update\_command$: $precommand := command$

$update\_slave$: $preslave := slave$

$reset\_states$: $stateCtr := \varnothing$

$act1$: $gstateCtr := \varnothing$

END

EVENT inactive2slavewrite
REFINES defaultCtrlEvent
ANY
c
WHERE
$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$
$a\_ctrl : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$addressphase : prehasAddress = TRUE$
$is\_called : preaddress = c$
$write\_transfer : precommand = write$
$not\_gnt : preGNT(c) = FALSE$
$transfer\_started : preFRAME = TRUE$
THEN
$going\_slave : slave(c) := TRUE$
$selected : DEVSEL := TRUE$
$ready : TRDY := TRUE$
$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
END

EVENT inactive2slaveread
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$
$didnt\_act : c \notin dom(stateCtr)$
$addressphase : prehasAddress = TRUE$
$is\_called : preaddress = c$
$read\_transfer : precommand = read$
$not\_gnt : preGNT(c) = FALSE$
$transfer\_started : preFRAME = TRUE$
THEN
$going\_slave : slave(c) := TRUE$
$selected : DEVSEL := TRUE$
$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
$turnaround : TRDY := FALSE$
END

EVENT normal_slave_write
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$is\_slave : preslave(c) = TRUE$
$transfer\_going : preFRAME = TRUE$
$master\_ready : preIRDY = TRUE$

$didnt\_act : c \notin dom(stateCtr)$

$write\_transfer : precommand = write$

$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$

$notgnt : preGNT(c) = FALSE$

THEN

$save\_data : buffers(c) := prebusdata$

$trdy : TRDY := TRUE$

$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

$samereq : REQ := REQ$

END

EVENT normal_slave_read

REFINES defaultCtrlEvent

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$notgnt : preGNT(c) = FALSE$

$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$

$didnt\_act : c \notin dom(stateCtr)$

$is\_slave : preslave(c) = TRUE$

$transfer\_going : preFRAME = TRUE$

$read\_transfer : precommand = read$

$master\_ready\_or\_start : (preDEVSEL = TRUE \wedge preTRDY = FALSE) \vee preIRDY = TRUE$

THEN

$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

$samereq : REQ := REQ$

$ready : TRDY := TRUE$

$write\_data : busdata :\in \mathbb{N}$

END

EVENT endslavewrite

REFINES defaultCtrlEvent

ANY

c

WHERE

$a\_ctrl : c \in CONTROLLERS$

$is\_slave : preslave(c) = TRUE$

$trasfer\_end : preFRAME = FALSE$

$master\_ready : preIRDY = TRUE$

$didnt\_act : c \notin dom(stateCtr)$

$write\_transfer : precommand = write$

$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$

$notgnt : preGNT(c) = FALSE$

THEN

$save\_data : buffers(c) := prebusdata$

$trdy : TRDY := TRUE$

$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

$samereq : REQ := REQ$

$not\_slave\_anymore : slave(c) := FALSE$

END

EVENT endslaveread
REFINES defaultCtrlEvent
ANY
c
WHERE
$a\_ctrl : c \in CONTROLLERS$
$notgnt : preGNT(c) = FALSE$
$is\_slave : preslave(c) = TRUE$
$didnt\_act : c \notin dom(stateCtr)$
$inactive : (pregstateCtr(c) = idle \wedge preREQ(c) = FALSE) \vee (pregstateCtr(c) = request \wedge preREQ(c) = TRUE \wedge \neg(prearbbusy = TRUE \wedge preowner = c))$
$trasfer\_end : preFRAME = FALSE$
$read\_transfer : precommand = read$
$master\_ready\_or\_start : (preDEVSEL = TRUE \wedge preTRDY = FALSE) \vee preIRDY = TRUE$
THEN
$write\_data : busdata :\in \mathbb{N}$
$not\_slave\_anymore : slave(c) := FALSE$
$samestateCtr : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
$samereq : REQ := REQ$
$ready : TRDY := TRUE$
END

EVENT controllers_sync
REFINES controllers_sync
WHEN
$grd1 : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
$grd2 : gstateCtr = \varnothing$
THEN
$act1 : gstateCtr := stateCtr$
END

END

B.3 O Protocolo AMBA

B.3.1 Nível 1 – Especificação abstrata do protocolo

MACHINE amba_m1

REFINES pci_amba_m0

SEES pci_amba_c0

VARIABLES

$finishedArb$
$stateCtr$
$arbbusy$

$prearbbusy$
$owner$
$preowner$
$gstateCtr$
$pregstateCtr$

EVENTS

INITIALISATION

BEGIN
$ini\_fa : finishedArb := FALSE$
$ini\_state : stateCtr := \varnothing$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_prearbbusy : prearbbusy := FALSE$
$ini\_owner : owner :\in CONTROLLERS$
$ini\_preowner : preowner :\in CONTROLLERS$
$act1 : pregstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
$act2 : gstateCtr := \varnothing$
END

EVENT samestateCtr

REFINES generic_controller_action
ANY
c
WHERE
$some\_ctrls : c \in CONTROLLERS$
$didnt\_act : c \notin dom(stateCtr)$
$grd1 : (pregstateCtr(c) = idle \wedge preowner \neq c) \vee (pregstateCtr(c) = request \wedge \neg(prearbbusy = TRUE \wedge preowner \neq c)) \vee (pregstateCtr(c) = master \wedge preowner = c)$
WITNESSES
$C : C = \{c\}$
$ns : ns = \{c \mapsto pregstateCtr(c)\}$
THEN
$statechange : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$
END

EVENT next_step
REFINES next_step
WHEN
$actions\_taken : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$
$arb\_done : finishedArb = TRUE$
THEN
$arb\_ready : finishedArb := FALSE$
$update\_states : pregstateCtr := gstateCtr$
$update\_arbiter : prearbbusy := arbbusy$
$reset\_states : stateCtr := \varnothing$
$update\_owner : preowner := owner$
$act1 : gstateCtr := \varnothing$
END

EVENT samestateArb
REFINES generic_arbiter_action

WHEN

$arb\_ready : finishedArb = FALSE$

$grd1 : (prearbbusy = FALSE \land pregstateCtr^{-1}[\{request\}] = \varnothing) \lor (prearbbusy = TRUE)$

THEN

$arb\_done : finishedArb := TRUE$

$update\_status : arbbusy := prearbbusy$

$update\_owner : owner := preowner$

END

EVENT controllers_sync

REFINES controllers_sync

WHEN

$grd1 : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

$grd2 : gstateCtr = \varnothing$

THEN

$act1 : gstateCtr := stateCtr$

END

EVENT idle2req

REFINES generic_controller_action

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : c \notin dom(stateCtr)$

$grd3 : pregstateCtr(c) = idle$

$grd4 : preowner \neq c$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto request\}$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto request\}$

END

EVENT idle2defmaster

REFINES generic_controller_action

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : c \notin dom(stateCtr)$

$grd3 : pregstateCtr(c) = idle$

$grd4 : preowner = c$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto master\}$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto master\}$

END

EVENT req2master

REFINES generic_controller_action

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : c \notin dom(stateCtr)$

$grd3 : pregstateCtr(c) = request$

$grd4 : preowner = c$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto master\}$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto master\}$

END

EVENT master2idle

REFINES generic_controller_action

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : pregstateCtr(c) = master$

$grd3 : preowner \neq c$

$grd4 : c \notin dom(stateCtr)$

WITNESSES

$C : C = \{c\}$

$ns : ns = \{c \mapsto idle\}$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto idle\}$

END

EVENT idle2busy

REFINES generic_arbiter_action

ANY

c

WHERE

$grd1 : c \in pregstateCtr^{-1}[\{request\}]$

$grd2 : finishedArb = FALSE$

$grd3 : arbbusy = FALSE$

THEN

$act1 : finishedArb := TRUE$

$act2 : arbbusy := TRUE$

$act3 : owner := c$

END

EVENT busy2busy

REFINES generic_arbiter_action

ANY

c

WHERE

$grd1 : c \in pregstateCtr^{-1}[\{request\}]$

$grd2 : finishedArb = FALSE$

$grd3 : prearbbusy = TRUE$

THEN

$act1 : arbbusy := TRUE$

$act2 : finishedArb := TRUE$

$act3 : owner := c$

END

EVENT busy2idle
REFINES generic_arbiter_action
WHEN
$grd1 : arbbusy = TRUE$
$grd2 : pregstateCtr^{-1}[\{request\}] = \varnothing$
$grd3 : finishedArb = FALSE$
THEN
$act1 : finishedArb := TRUE$
$act2 : arbbusy := FALSE$
$act3 : owner :\in CONTROLLERS$
END

END

B.3.2 Contexto para arbitração concreta

CONTEXT amba_c1

REFINES pci_amba_c0

CONSTANTS

$DefCtr$

AXIOMS

$axm1 : DefCtr \in CONTROLLERS$

END

B.3.3 Nível 2 – Especificação concreta da arbitração

MACHINE amba_n2

REFINES amba_n1

SEES amba_c1

VARIABLES

$finishedArb$
$stateCtr$
$arbbusy$
$prearbbusy$
$gstateCtr$

$pregstateCtr$
$HGRANT$
$preHGRANT$
$prepreHGRANT$
$preprearbbusy$
$HMASTER$
$preHMASTER$

INVARIANTS

$inv1 : HGRANT \in CONTROLLERS \rightarrow BOOL$
$inv2 : HGRANT = (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{owner \mapsto TRUE\}$
$inv3 : preHGRANT \in CONTROLLERS \rightarrow BOOL$
$inv4 : preHGRANT = (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{preowner \mapsto TRUE\}$
$inv5 : prepreHGRANT \in CONTROLLERS \rightarrow BOOL$
$inv6 : \forall c \cdot (preHGRANT(c) = TRUE \Leftrightarrow preowner = c)$
$inv7 : \forall c \cdot (HGRANT(c) = TRUE \Leftrightarrow owner = c)$
$inv8 : \forall c \cdot (preHGRANT(c) = FALSE \Leftrightarrow preowner \neq c)$
$inv9 : \forall c \cdot (HGRANT(c) = FALSE \Leftrightarrow owner \neq c)$
$inv10 : prearbbusy = FALSE \Rightarrow preowner = DefCtr$
$inv11 : arbbusy = FALSE \Rightarrow owner = DefCtr$
$inv12 : preprearbbusy \in BOOL$
$inv13 : HMASTER \in CONTROLLERS$
$inv14 : preHMASTER \in CONTROLLERS$

EVENTS

INITIALISATION

BEGIN
WITNESSES
$owner' : owner' = DefCtr$
$preowner' : preowner' = DefCtr$
$ini\_fa : finishedArb := FALSE$
$ini\_state : stateCtr := \varnothing$
$ini\_arbbusy : arbbusy := FALSE$
$ini\_prearbbusy : prearbbusy := FALSE$
$ini\_HGRANT : HGRANT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{DefCtr \mapsto TRUE\}$
$ini\_preHGRANT : preHGRANT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{DefCtr \mapsto TRUE\}$
$act1 : pregstateCtr :\in CONTROLLERS \rightarrow CTR\_STATES$
$act2 : gstateCtr := \varnothing$
$ini\_prepreHGRANT : prepreHGRANT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{DefCtr \mapsto TRUE\}$
$act3 : preprearbbusy := FALSE$
$act4 : HMASTER := DefCtr$
$act5 : preHMASTER := DefCtr$
END

EVENT samestateCtr

REFINES samestateCtr
ANY
c
WHERE
$some\_ctrls : c \in CONTROLLERS$

$didnt\_act : c \notin dom(stateCtr)$

$grd1 : (pregstateCtr(c) = idle \land preHGRANT(c) = FALSE) \lor (pregstateCtr(c) = request \land \neg(prearbbusy = TRUE \land preHGRANT(c) = FALSE)) \lor (pregstateCtr(c) = master \land preHGRANT(c) = TRUE)$

THEN

$statechange : stateCtr := stateCtr \cup \{c \mapsto pregstateCtr(c)\}$

END

EVENT next_step

REFINES next_step

WHEN

$actions\_taken : gstateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

$arb\_done : finishedArb = TRUE$

THEN

$arb\_ready : finishedArb := FALSE$

$update\_states : pregstateCtr := gstateCtr$

$update\_arbiter : prearbbusy := arbbusy$

$act3 : preprearbbusy := prearbbusy$

$reset\_states : stateCtr := \varnothing$

$update\_owner : preHGRANT := HGRANT$

$act1 : gstateCtr := \varnothing$

$act2 : prepreHGRANT := preHGRANT$

$act4 : preHMASTER := HMASTER$

END

EVENT samestateArb

REFINES samestateArb

WHEN

$arb\_ready : finishedArb = FALSE$

$grd1 : ((prearbbusy = FALSE \land pregstateCtr^{-1}[\{request\}] = \varnothing) \lor (prearbbusy = TRUE)) \land \neg(preprearbbusy = FALSE \lor prepreHGRANT \neq preHGRANT)$

THEN

$arb\_done : finishedArb := TRUE$

$update\_status : arbbusy := prearbbusy$

$update\_owner : HGRANT := preHGRANT$

END

EVENT authtransferArb

REFINES samestateArb

ANY

c

WHERE

$grd1 : finishedArb = FALSE$

$grd2 : ((prearbbusy = FALSE \land pregstateCtr^{-1}[\{request\}] = \varnothing) \lor (prearbbusy = TRUE)) \land (preprearbbusy = FALSE \lor prepreHGRANT \neq preHGRANT)$

$grd3 : preHGRANT(c) = TRUE$

THEN

$act1 : finishedArb := TRUE$

$act2 : arbbusy := prearbbusy$

$act3 : HGRANT := preHGRANT$

$act4 : HMASTER := c$

END

EVENT controllers_sync

REFINES controllers_sync

WHEN

$grd1 : stateCtr \in CONTROLLERS \rightarrow CTR\_STATES$

$grd2 : gstateCtr = \varnothing$

THEN

$act1 : gstateCtr := stateCtr$

END

EVENT idle2req

REFINES idle2req

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : c \notin dom(stateCtr)$

$grd3 : pregstateCtr(c) = idle$

$grd4 : preHGRANT(c) = FALSE$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto request\}$

END

EVENT idle2defmaster

REFINES idle2defmaster

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : c \notin dom(stateCtr)$

$grd3 : pregstateCtr(c) = idle$

$grd4 : preHGRANT(c) = TRUE$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto master\}$

END

EVENT req2master

REFINES req2master

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : c \notin dom(stateCtr)$

$grd3 : pregstateCtr(c) = request$

$grd4 : preHGRANT(c) = TRUE$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto master\}$

END

EVENT master2idle

REFINES master2idle

ANY

c

WHERE

$grd1 : c \in CONTROLLERS$

$grd2 : pregstateCtr(c) = master$

$grd3 : preHGRANT(c) = FALSE$

$grd4 : c \notin dom(stateCtr)$

THEN

$act1 : stateCtr := stateCtr \cup \{c \mapsto idle\}$

END

EVENT idle2busy

REFINES idle2busy

ANY

c

WHERE

$grd1 : c \in pregstateCtr^{-1}[\{request\}]$

$grd2 : finishedArb = FALSE$

$grd3 : arbbusy = FALSE$

THEN

$act1 : finishedArb := TRUE$

$act2 : arbbusy := TRUE$

$act3 : HGRANT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{c \mapsto TRUE\}$

END

EVENT busy2busy

REFINES busy2busy

ANY

c

WHERE

$grd1 : c \in pregstateCtr^{-1}[\{request\}]$

$grd2 : finishedArb = FALSE$

$grd3 : prearbbusy = TRUE$

THEN

$act1 : arbbusy := TRUE$

$act2 : finishedArb := TRUE$

$act3 : HGRANT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{c \mapsto TRUE\}$

END

EVENT busy2idle

REFINES busy2idle

WHEN

$grd1 : arbbusy = TRUE$

$grd2 : pregstateCtr^{-1}[\{request\}] = \varnothing$

$grd3 : finishedArb = FALSE$

WITNESSES

$owner' : owner' = DefCtr$

THEN

$act1 : finishedArb := TRUE$

$act2 : arbbusy := FALSE$

$act3 : HGRANT := (CONTROLLERS \times \{FALSE\}) \lhd\!\!\!+ \{DefCtr \mapsto TRUE\}$

END

END

B.4 Propriedades AADL

B.4.1 Noções básicas de componentes AADL

CONTEXT aadl_c0

SETS

$COMPONENTS$
$CATEGORIES$

CONSTANTS

c_types
c_impls
$typeof$
$categoryof$
$system$
$processor$
$device$
$memory$
bus
$process$
$thread\_group$
$thread$
$subprogram$
$data$

AXIOMS

$axm1 : CATEGORIES = \{system, processor, device, memory, bus, process, thread\_group, thread, subprogram, data\}$
$axm2 : system \neq processor$
$axm3 : system \neq device$
$axm4 : system \neq memory$
$axm5 : system \neq bus$
$axm6 : system \neq process$
$axm7 : system \neq thread\_group$
$axm8 : system \neq thread$
$axm9 : system \neq subprogram$
$axm10 : system \neq data$
$axm11 : processor \neq device$
$axm12 : processor \neq memory$
$axm13 : processor \neq bus$
$axm14 : processor \neq process$
$axm15 : processor \neq thread\_group$
$axm16 : processor \neq thread$
$axm17 : processor \neq subprogram$
$axm18 : processor \neq data$
$axm19 : device \neq memory$
$axm20 : device \neq bus$
$axm21 : device \neq process$
$axm22 : device \neq thread\_group$

$axm23 : device \neq thread$
$axm24 : device \neq subprogram$
$axm25 : device \neq data$
$axm26 : memory \neq bus$
$axm27 : memory \neq process$
$axm28 : memory \neq thread\_group$
$axm29 : memory \neq thread$
$axm30 : memory \neq subprogram$
$axm31 : memory \neq data$
$axm32 : bus \neq process$
$axm33 : bus \neq thread\_group$
$axm34 : bus \neq thread$
$axm35 : bus \neq subprogram$
$axm36 : bus \neq data$
$axm37 : process \neq thread\_group$
$axm38 : process \neq thread$
$axm39 : process \neq subprogram$
$axm40 : process \neq data$
$axm41 : thread\_group \neq thread$
$axm42 : thread\_group \neq subprogram$
$axm43 : thread\_group \neq data$
$axm44 : thread \neq subprogram$
$axm45 : thread \neq data$
$axm46 : subprogram \neq data$
$axm47 : categoryof \in COMPONENTS \rightarrow CATEGORIES$
$axm48 : typeof \in c\_impls \rightarrow c\_types$
$axm49 : COMPONENTS = c\_impls \cup c\_types$
$axm50 : c\_impls \cap c\_types = \varnothing$
$axm51 : c\_impls \lhd typeof; categoryof = c\_impls \lhd categoryof$

END

B.4.2 Definição da semântica de subcomponentes

CONTEXT aadl_c1
REFINES aadl_c0

SETS

$NAMES$
$REFERENCES$

CONSTANTS

$root$
$self$
$subcomponentsof$
$nameof$
$componentof$
$parentof$
$path$

$subtree$

$pathleaf$

$rootcomp$

$parentcompof$

$comprel$

$tclcomp$

AXIOMS

$axm1 : self \in NAMES$

$axm2 : subcomponentsof \in c\_impls \rightarrow (NAMES \nrightarrow COMPONENTS)$

$axm3 : root \in REFERENCES$

$axm4 : nameof \in REFERENCES \rightarrow NAMES$

$axm5 : componentof \in REFERENCES \setminus \{root\} \rightarrow COMPONENTS$

$axm6 : parentof \in REFERENCES \setminus \{root\} \rightarrow REFERENCES$

$axm7 : \forall r \cdot (r \neq root \land parentof(r) \neq root \Rightarrow componentof(parentof(r)) \in c\_impls)$

$axm8 : \forall c \cdot (\forall n \cdot (n \in NAMES \land \{n \mapsto c\} \notin ran(subcomponentsof) \Rightarrow$
$(\exists r \cdot (r \in REFERENCES \setminus \{root\} \land parentof(r) = root \land$
$nameof(r) = n \land n = self \land componentof(r) = c))))$

$axm9 : \forall r1, r2 \cdot (r2 \in (REFERENCES \setminus \{root\}) \land r1 \in (REFERENCES \setminus \{root\}) \land r2 = parentof(r1) \Rightarrow$
$(nameof(r1) \mapsto componentof(r1) \in subcomponentsof(componentof(r2))))$

$axm10 : \forall r1, r2 \cdot (r2 \in (REFERENCES \setminus \{root\}) \land r1 \in (REFERENCES \setminus \{root\}) \land nameof(r1) = nameof(r2) \land$
$parentof(r1) = parentof(r2) \Rightarrow r1 = r2)$

$axm11 : \forall r1, n, c \cdot (r1 \in (REFERENCES \setminus \{root\}) \land componentof(r1) \in c\_impls \land n \mapsto c \in subcomponentsof(componentof(r1)) \Rightarrow$
$(\exists r2 \cdot (r2 \neq root \land nameof(r2) = n \land componentof(r2) = c \land parentof(r2) = r1)))$

$axm12 : \forall n, c2, c1 \cdot ((c1 \in c\_impls \land n \mapsto c2 \in subcomponentsof(c1) \land categoryof(c1) = data) \Rightarrow categoryof(c2) =$
$data \land categoryof(c1) \neq subprogram \land categoryof(c1) \neq bus \land categoryof(c1) \neq device \land (categoryof(c1) = thread \Rightarrow$
$categoryof(c2) = data) \land (categoryof(c1) = thread\_group \Rightarrow categoryof(c2) = data \lor categoryof(c2) = thread \lor categoryof(c2) =$
$thread\_group) \land (categoryof(c1) = process \Rightarrow categoryof(c2) = thread \lor categoryof(c2) = data \lor categoryof(c2) =$
$thread\_group) \land (categoryof(c1) = processor \Rightarrow categoryof(c2) = memory) \land (categoryof(c1) = memory \Rightarrow categoryof(c2) =$
$memory) \land (categoryof(c1) = system \Rightarrow categoryof(c2) = system \lor categoryof(c2) = data \lor categoryof(c2) = process \lor$
$categoryof(c2) = processor \lor categoryof(c2) = memory \lor categoryof(c2) = bus \lor categoryof(c2) = device))$

$axm13 : path = (\bigcup n \cdot (n \in \mathbb{N}) | (1 \mathinner{..} n) \rightarrow NAMES)$

$axm14 : subtree \in COMPONENTS \rightarrow \mathbb{P}(path)$

$axm15 : \forall p, name, num \cdot ((p \in path \land num \mapsto name \in p \land num = 1) \Rightarrow name = self)$

$axm16 : pathleaf \in path \rightarrow COMPONENTS$

$axm17 : rootcomp \in c\_impls$

$axm18 : parentcompof \in c\_impls \setminus \{rootcomp\} \rightarrow c\_impls$

$axm19 : \forall c, d \cdot (c \in c\_impls \setminus \{rootcomp\} \land$
$d \in c\_impls \setminus \{rootcomp\} \land (d \in ran(subcomponentsof(c)) \Leftrightarrow parentcompof(d) = c))$

$axm20 : \forall d \cdot (d \in c\_impls \setminus \{rootcomp\} \land (\neg(\exists c \cdot (c \in c\_impls \setminus \{rootcomp\} \land$
$(d \in ran(subcomponentsof(c)))) \Leftrightarrow parentcompof(d) = rootcomp)))$

$axm21 : comprel = c\_impls \leftrightarrow c\_impls$

$axm22 : tclcomp \in comprel \rightarrow comprel$

$axm23 : \forall c \cdot (c \in comprel \Rightarrow c \subseteq tclcomp(c))$

$axm24 : \forall c \cdot (c \in comprel \Rightarrow c; tclcomp(c) \subseteq tclcomp(c))$

$axm25 : \forall c, t \cdot (c \in comprel \land c \subseteq t \land c; t \subseteq t \Rightarrow tclcomp(c) \subseteq t)$

$axm26 : tclcomp(parentcompof) \cap id(c\_impls) = \varnothing$

THEOREMS

$thm1 : \forall r \cdot (r \neq root \land r \in REFERENCES \Rightarrow r \in dom(parentof) \land r \in dom(componentof))$

$thm2 : finite(COMPONENTS) \land finite(NAMES) \Rightarrow finite(REFERENCES)$

END

B.4.3 Especificação da semântica individual de propriedades

CONTEXT aadl_c2

REFINES aadl_c1

CONSTANTS

$Allowed\_Memory\_Binding\_Class$
$Allowed\_Memory\_Binding$
$Actual\_Memory\_Binding$

AXIOMS

$axm1 : Allowed\_Memory\_Binding\_Class \in COMPONENTS \leftrightarrow (\mathbb{P}(\{system, processor, memory\}) \times path)$
$axm2 : \forall cat, p, c \cdot (c \mapsto (cat \mapsto p) \in Allowed\_Memory\_Binding\_Class \Rightarrow$
$p \in subtree(c) \land categoryof(pathleaf(p)) \in$
$\{thread, thread\_group, process, system, device, subprogram, processor\})$
$axm3 : Allowed\_Memory\_Binding \in COMPONENTS \leftrightarrow (\mathbb{P}(path) \times path)$
$axm4 : \forall ref, listref, trgt, c \cdot (ref \in listref \land c \mapsto (listref \mapsto trgt) \in Allowed\_Memory\_Binding \Rightarrow ref \in subtree(c) \land$
$trgt \in subtree(c) \land categoryof(pathleaf(ref)) \in \{memory, system, processor\} \land categoryof(pathleaf(trgt)) \in \{thread, thread\_group, proce$
$axm5 : Actual\_Memory\_Binding \in COMPONENTS \leftrightarrow (path \times path)$
$axm6 : \forall ref, trgt, c \cdot (c \mapsto (ref \mapsto trgt)) \in Actual\_Memory\_Binding \Rightarrow categoryof(pathleaf(ref)) \in \{memory, system, processor\} \land$
$categoryof(pathleaf(trgt)) \in$
$\{thread, thread\_group, process, system, device, subprogram, processor\}$
$axm7 : \forall c \cdot (Actual\_Memory\_Binding[\{c\}] \subseteq subtree(c) \times subtree(c))$

END

B.4.4 Especificação de dependências entre propriedades

CONTEXT aadl_c3

REFINES aadl_c2

AXIOMS

$axm1 : \forall trgt, c, listref, cats \cdot ((c \mapsto (cats \mapsto trgt) \in Allowed\_Memory\_Binding\_Class \land c \mapsto (listref \mapsto trgt) \in$
$Allowed\_Memory\_Binding) \Rightarrow (pathleaf; categoryof)[listref] \subseteq cats)$
$axm2 : \forall trgt, c, ref, cats \cdot ((c \mapsto (cats \mapsto trgt) \in Allowed\_Memory\_Binding\_Class \land c \mapsto (ref \mapsto trgt) \in Actual\_Memory\_Binding) \Rightarrow$
$(pathleaf; categoryof)[\{ref\}] \subseteq cats)$
$axm3 : \forall trgt, c, ref, listref \cdot (c \mapsto (listref \mapsto trgt) \in Allowed\_Memory\_Binding \land c \mapsto (ref \mapsto trgt) \in Actual\_Memory\_Binding) \Rightarrow$
$(ref \in listref)$

END

Referências

[1] AADL Predictable Model-Based Engineering. `http://www.aadl.info`.

[2] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, and Tony Tseng. Modeling and implementing software architecture with ACME and ArchJava. In ICSE '05: Proceedings of the 27th international conference on Software engineering, pages 676–677, New York, NY, USA, 2005. ACM.

[3] Jean-Raymond Abrial. The B Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[4] Robert Allen. A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[5] K R Apt and D C Kozen. Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett., 22(6):307–309, 1986.

[6] ARM Holdings. ARM – The Architecture for the Digital World. `http://www.arm.com/`.

[7] ARM Limited. AMBA Specification (Rev. 2), 1999.

[8] Elisabeth Ball and Michael Butler. Event-B patterns for specifying fault-tolerance in multi-agent interaction. in: Methods, models and tools for fault tolerance. In Workshop on Methods, Models and Tools for Fault Tolerance, pages 4–13, November 2007.

[9] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. Proceedings of the IEEE, 91(1):64–83, 2003.

[10] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In Seminar on Concurrency, Carnegie-Mellon University, pages 389–448, London, UK, 1984. Springer-Verlag.

[11] Egon Börger and Robert F. Stark. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[12] Dominique Cansell, Dominique Méry, and Joris Rehm. Time constraint patterns for event b development. pages 140–154. 2006.

[13] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In Fourteenth annual ACM symposium on principles of programming languages, pages 178–188, January 1987.

[14] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Logic of Programs, Workshop, pages 52–71, London, UK, 1982. Springer-Verlag.

[15] ClearSy System Engineering. Méthode B. `http://www.methode-b.com/`.

[16] ClearSy System Engineering. Industrial Use of the B-Method, 2006. 30 slides.

[17] Paul C. Clements. A survey of architecture description languages. In IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[18] Kriangsak Damchoom, Michael Butler, and Jean-Raymond Abrial. Modelling and proof of a tree-structured file system in event-b and rodin. pages 25–44. 2008.

[19] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 85–94, New York, NY, USA, 1995. ACM.

[20] E. Allen Emerson and Kedar S. Namjoshi. Verification of a parameterized bus arbitration protocol. In CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification, pages 452–463, London, UK, 1998. Springer-Verlag.

[21] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nml. In EDTC '95: Proceedings of the 1995 European conference on Design and Test, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[22] Ricardo Bedin França, Jean-Paul Bodeveix, and Mamoun Filali Amine. AADLModeling of a Generic Bus. In Journées FAC, Toulouse, France, 2007.

[23] Ricardo Bedin França, Jean-Paul Bodeveix, Mamoun Filali, Jean-Francois Rolland, David Chemouil, and Dave Thomas. The AADL behaviour annex – experiments and roadmap. In ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), pages 377–382, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Ricardo Bedin França, J.-M. Farines, Jean-Paul Bodeveix, Leandro Becker, and Mamoun Filali. Modeling a Bus Protocol: an Incremental Approach. In 9th Workshop on Real-Time Systems (WTR 2007), Belém, Brasil, 2007.

[25] Ricardo Bedin França, Jean-François Rolland, Mamoun Filali Amine, Jean-Paul Bodeveix, and David Chemouil. Assessment of the AADL Behavioral Annex. In Journées FAC, Toulouse, France, 2007.

[26] David Garlan and Dewayne E. Perry. Introduction to the Special Issue on Software Architecture. IEEE Trans. Softw. Eng., 21(4):269–274, 1995.

[27] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: An Instruction Set Description Language for Retargetability. In DAC '97: Proceedings of the 34th Annual Conference on Design Automation, pages 299–302, New York, NY, USA, 1997. ACM.

[28] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: A Language for Architecture Exploration Through Compiler/-Simulator Retargetability. In Proceedings of the European Conference on Design, Automation and Test, pages 485–490, 1999.

[29] IEEE Standards Description: 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems Description. Technical report.

[30] Information Society Technologies. Event-B and the Rodin Platform. `http://www.event-b.org/`.

[31] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. Model checking. MIT Press, Cambridge, MA, USA, 1999.

[32] Leslie Lamport. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[33] Paul Le Guernic, M. Le Borgne, T. Gautier, and Cl. Le Maire. Programming real time applications with SIGNAL. Technical Report 1446, INRIA, June 1991.

[34] Alexei Lliasov. Refinement patterns for rapid development of dependable systems. In Workshop on Engineering fault tolerant systems, 2007.

[35] David C. Luckham. Rapide: a language and toolset for simulation of distributed systems by partial orderings of events. In POMIV '96: Proceedings of the DIMACS workshop on Partial order methods in verification, pages 329–357, New York, NY, USA, 1997. AMS Press, Inc.

[36] David C. Luckham, James Vera, and Sigurd Meldal. Key concepts in architecture definition languages. Foundations of component-based systems, pages 23–45, 2000.

[37] Grant Martin. SystemC and the Future of Design Languages: Opportunities for Users and Research. In SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[39] Prabhat Mishra and Nikil Dutt. Architecture Description Languages for Programmable Embedded Systems. In IEE Proceedings on Computers and Digital Techniques, pages 285–297, 2005.

[40] Haja Moinudeen, Ali Habibi, and Sofiene Tahar. Design for Verification of the PCI-X Bus. In FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design, pages 187–188, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Christophe Métayer, Jean-Raymond Abrial, and Laurent Voisin. Rodin Deliverable 3.2. Technical Report IST-511599, Information Society Technologies, 2005.

[42] Karim Oumalou, Ali Habibi, and Sofiene Tahar. Design for Verification of a PCI Bus in SystemC. In Proceedings of the 2004 International Symposium on System-on-Chip (SOC '04), pages 201–204. IEEE, 2004.

[43] Bhaskar Pal, Ansuman Banerjee, Pallab Dasgupta, and Partha Pratim Chakrabarti. The BUSpec Platform for Automated Generation of Verification Aids for Standard Bus Protocols. In Formal Methods and Models for Co-Design – MEMOCODE '04, San Diego, USA, 2004.

[44] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation, pages 933–938, New York, NY, USA, 1999. ACM.

[45] Amir Pnueli and Lenore D. Zuck. Model-checking and abstraction to the aid of parameterized systems. In VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, page 4, London, UK, 2003. Springer-Verlag.

[46] ProMARTE Working Group. `http://www.promarte.org`.

[47] P.S.I. Group. PCI Local Bus Specification Rev. 2.2., 1998.

[48] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. An Architecture-Based Dependability Modeling Framework Using AADL. CoRR, abs/0704.0865, 2007.

[49] SAE AS5C Committee. Proceedings of the Avionics Architecture Description Language seminar, 2002. 131 slides.

[50] Society of Automotive Engineers. Architecture Analysis & Design Language (AADL) AS5506, 2004.

[51] Society of Automotive Engineers. Architecture Analysis & Design Language (AADL) AS5506 Draft V1.6, 2008.

[52] Software Engineering Institute - Carnegie Mellon University. `http://www.sei.cmu.edu`.

[53] SPICES. D5.1.7 v1.0 SPICES Project Profile, 2006.

[54] SPICES. D5.1.8 v1.0 SPICES Posters for the ITEA Symposium, 2007.

[55] William Stallings. Computer Organization and Architecture - Designing for Performance. Prentice Hall, 1994.

[56] Stuart Swan. Systemc Transaction Level Models and RTL Verification. In DAC '06: Proceedings of the 43rd annual conference on Design automation, pages 90–92, New York, NY, USA, 2006. ACM.

[57] Toolkit in OPen-source for Critical Applications & SystEms Development. `http://www.topcased.org`.

[58] TOPCASED. Description technique du projet TOPCASED, 2006.

[59] UML Resource Page. `http://www.uml.org`.

[60] UML Profile for Schedulability, Performance, and Time Specification, 2002.

[61] Steve Vestal. Technical and Historical Overview of MetaH, 2000.

[62] John A. Zachman. A framework for information systems architecture. IBM Syst. J., 26(3):276–292, 1987.

[63] Yann Zimmermann. Développement formel de circuits électroniques par la méthode B. In Approches Formelles dans l'Assistance au Développement de Logiciels, pages 181–198, Namur, Belgium, 2007. ACM.