

IVAN JEUKENS

Uma Metodologia para Projeto de
Software Embarcado
Baseada em Múltiplos Modelos
Computacionais

Tese apresentada à Escola Politécnica
da Universidade de São Paulo
para obtenção do Título de
Doutor em Engenharia Elétrica.

São Paulo
2005

Jeukens, Ivan

Uma Metodologia para Projeto de Software Embarcado baseada em
Múltiplos Modelos Computacionais

São Paulo, SP, Brasil, 2005.
p. 218

Tese de Doutorado. Escola Politécnica da Universidade de São
Paulo. Departamento de Engenharia Elétrica.

1. Sistemas Embarcados 2. Modelos Computacionais 3.
Especificações Executáveis 4. Metodologia de projeto.
I.Universidade de São Paulo. Escola Politécnica. Departamento de
Engenharia Elétrica II. t.

*Esta tese é dedicada a
papai, mamãe e Ricco.*

AGRADECIMENTOS

A Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP) por conceder uma bolsa de doutorado, processo 00/12146-0.

Ao colega Ferrando Herrera da Universidade de Cantabria, que me proporcionou uma estadia agradável em sua cidade, além de ajudar com uma versão inicial para a implementação de um canal CSP.

RESUMO

Sistemas embarcados estão presentes em inúmeros setores da vida cotidiana. São empregados em uma grande gama de sistemas, como aparelhos de entretenimento, veículos, aeronaves, sistemas de segurança, processos industriais de manufatura e aparelhos de medidas. Sistemas embarcados são por definição reativos, heterogêneos e concorrentes.

A constante evolução da tecnologia de semicondutores e requisitos de funcionalidade e desempenho cada vez mais exigentes, fazem com que técnicas avançadas de projeto de software para sistemas embarcados sejam necessárias. Não é mais possível utilizar a solução de escrever um código monolítico em alguma linguagem de baixo nível.

Nesta tese, estudo a aplicação de uma metodologia baseada em especificações executáveis controladas por um ou mais modelos computacionais. Uma especificação executável é composta por elementos funcionais que interagem entre si. Um modelo computacional dita as regras para esta interação. Dentre outras vantagens, esta abordagem permite a escolha do conjunto de modelos computacionais mais adequados para cada parte do sistema embarcado.

Demonstro nesta tese como obter implementações a partir de especificações executáveis. Detalho como caracterizá-las a fim de comparar a eficiência de cada solução.

Abordo o problema da escolha de um modelo computacional através de uma estratégia para determinar se um dado modelo é válido para uma especificação ou parte dela. Implemento esta estratégia em uma ferramenta de software.

Os resultados obtidos são aplicados a dois estudos de caso. O primeiro é um subconjunto de um módulo codificador de voz. O segundo é um exemplo de sistema de controle, proposto a fim de comparar diferentes técnicas de modelamento e implementação de sistemas embarcados.

ABSTRACT

Embedded systems can be found in several areas of modern life. They are employed in a broad field of applications, such as entertainment devices, vehicles, aircrafts, security systems, industrial process control and measuring devices. Embedded systems are by definition reactive, heterogeneous and concurrent.

The constant evolution of semiconductor technology together with functional and performance requirements, make advanced software engineering techniques imperative. It is no longer feasible to employ a methodology centered in writing monolithic low level code.

In this thesis, I present the development of a methodology based in executable specifications controlled by one or more models of computation. An executable specification is composed of interacting functional elements. A model of computation dictates the rules of this interaction. Among other advantages, this approach allows for the selection of the most appropriate set of models for each part of the system.

I demonstrate how to obtain implementations starting from an executable specification and show how to analyse each implementation in order to compare the efficiency of each solution.

I propose a solution to the problem of choosing one or more models of computation by devising a strategy based on finding out if a specification, or part of it, is valid for a particular model of computation. A software tool that implements this strategy is presented.

Tests are performed on two case studies. The first one is a subset of a vocoder system. The second one is a control system, originally developed for comparing different system description languages.

Sumário

CAPÍTULO 1 - INTRODUÇÃO.....	25
1.1 Sistemas Eletrônicos Embarcados.....	25
1.2 Paradigma de Interação de Atores.....	27
1.3 Objetivos e Contribuições.....	29
1.4 Trabalhos Relacionados.....	32
CAPÍTULO 2 – MODELOS COMPUTACIONAIS.....	35
2.1 Introdução.....	35
2.2 Modelo de Fluxo de Dados.....	37
2.2.1 Fluxo de Dados Síncrono.....	38
2.3 Rede de Processos Kahn.....	42
2.4 Processos Seqüenciais Comunicantes.....	46
2.5 Síncrono Reativo.....	48
2.6 Eventos Discretos.....	52
2.7 Tempo Contínuo.....	54
2.8 Interação entre Modelos Computacionais.....	55
CAPÍTULO 3- METODOLOGIA DE PROJETO PARA SOFTWARE EMBARCADO.....	61
3.1 Introdução.....	61
3.2 Nível de Comunicação de Processos.....	66
3.2.1 O sistema operacional eCos.....	66
3.2.2 O modelo de programação.....	68
3.2.3 Caracterização de software embarcado.....	73
3.3 Nível de Interação de atores.....	75
3.3.2 Transformações independentes de um modelo computacional.....	77
3.3.3 Transformação a partir do modelo SDF.....	79
3.3.4 Transformação a partir do modelo SR.....	86
3.3.5 Transformação a partir do modelo CSP.....	93
3.3.6 Transformação a partir do modelo PN.....	98

CAPÍTULO 4 – VALIDAÇÃO DE ESPECIFICAÇÕES EXECUTÁVEIS.....	106
4.1 Introdução.....	106
4.2 Condições para uma especificação válida.....	108
4.2.1 SDF.....	108
4.2.2 SR.....	108
4.3 Polimorfismo.....	109
4.4 Estratégia para validação.....	111
4.5 A ferramenta FASI.....	112
4.5.1 Etapas da análise.....	113
4.5.1.1 Estrutura de dados.....	114
4.5.1.2 Fase de Pré-processamento	116
4.5.1.3 Simplificação de código.....	118
4.5.1.4 Validações independentes de um Modelo Computacional.....	120
4.5.2 Validação para o modelo SDF.....	121
4.5.2.1 Validação de atores atômicos.....	121
4.5.2.2 Validação de uma topologia.....	125
4.5.3 Validação para o modelo SR.....	127
4.5.3.1 Validação de atores atômicos.....	127
4.5.3.2 Validação de uma topologia.....	134
4.5.4 Implementação da estratégia de validação.....	135
4.5.5 Resultados Experimentais.....	138
4.6 Considerações.....	141
CAPÍTULO 5 - ESTUDO DE CASOS.....	143
5.1 Codificador de voz padrão GSM.....	143
5.1.1 Especificação.....	143
5.1.2 Especificação executável.....	147
5.1.3 Análise.....	152
5.1.4 Implementação.....	155
5.1.4.1 A versão SDF + SR.....	155
5.1.4.2 A versão PN.....	159
5.1.5 Resultados.....	160

5.2 Controle de trenó.....	162
5.2.1 Especificação.....	163
5.2.2 Especificação executável.....	166
5.2.3 Análise.....	176
5.2.4 Implementação.....	178
5.2.5 Resultados.....	179
5.3 Considerações.....	180
CAPÍTULO 6 - CONCLUSÕES E TRABALHOS FUTUROS.....	183
6.1 Considerações Finais.....	183
6.2 Trabalhos Futuros.....	185
ANEXO A - O AMBIENTE PTOLEMY II.....	188
A.1 Introdução.....	188
A.2 O pacote Kernel.....	188
A.3 O pacote Actor.....	190
A.4 Anatomia de Atores.....	193
A.4.1 Atores Atômicos.....	193
A.4.2 Atores hierárquicos.....	198
ANEXO B – RESUMO SOBRE TEORIA DE ORDENS.....	202
REFERÊNCIAS BIBLIOGRÁFICAS.....	204
APÊNDICE A – CARACTERIZAÇÃO DOS PROCEDIMENTOS DO AMBIENTE ECOS.....	210
APÊNDICE B – EXEMPLO DE RELATÓRIO DE OCUPAÇÃO E ESCALONAMENTO DE PROCESSOS.....	212
APÊNDICE C - A FERRAMENTA FASI.....	216

Lista de Figuras

Figura 1(a) um ator; (b) um grafo exemplificando a interação entre atores.	37
Figura 2Dois grafos SDF. O número associado ao terminal de uma aresta próximo a um ator fonte é uma taxa de produção, enquanto que o número associado ao terminal próximo a um ator destino uma taxa de consumo. Por exemplo, o ator B consome 1 dado e produz 2 na figura (b).	40
Figura 3Matrizes de topologia dos grafos da Figura 2.	40
Figura 4Duas topologias SDF inválidas com matrizes consistentes.	42
Figura 5(a) Uma topologia com dois atores PN; (b) Pseudo-código para a topologia da figura (a).	45
Figura 6(a) Dependência; (b) Paradoxo; (c) Ambigüidade.	50
Figura 7A CPO do modelo SR.	50
Figura 8Captura conceitual de um sistema EDO em uma topologia de atores.	56
Figura 9Interação entre atores hierárquicos com modelos distintos.	57
Figura 10Um exemplo de topologia CT contendo ator DE	58
Figura 11Um nível de abstração.	62
Figura 12Arquitetura alvo da metodologia.	63
Figura 13Metodologia para projeto de software embarcado.	65
Figura 14Diagrama simplificado do sistema eCos [MAS03].	68
Figura 15Exemplo de uso de processos.	71
Figura 16Uma topologia SDF.	81
Figura 17Pseudo-código para A(2 B(2 C)).	81
Figura 18Exemplo de topologia SDF.	85
Figura 19Resultado da alocação de memória.	86
Figura 20Procedimentos de execução da topologia.	87
Figura 21Exemplo de topologia SR.	89
Figura 22Topologia do ator hierárquico Block.	89
Figura 23Código para os métodos fire() e postfire() do ator NonStrictDelay.	90
Figura 24Tradução do ator NonStrictDelay.	91
Figura 25Memória alocada para a topologia da Figura 21.	92
Figura 26Trecho do procedimento que implementa o método fire().	93

Figura 27Tradução de um ator atômico associado a um processo.	94
Figura 28Os procedimentos <code>csp_write()</code> e <code>csp_read()</code> .	95
Figura 29Exemplo de topologia CSP.	97
Figura 30Código dos atores Producer e Consumer da Figura 29.	98
Figura 31Estrutura de dados para um canal PN com tamanho fixo.	100
Figura 32Procedimentos de escrita e leitura em um canal PN de capacidade fixa.	101
Figura 33Estrutura de dados para fila de capacidade variável.	102
Figura 34Procedimento de leitura de um canal PN com capacidade variável.	103
Figura 35Exemplo de topologia no modelo PN.	104
Figura 36Resultado da transformação aplicada a topologia da Figura 35.	105
Figura 37Um ator específico para semânticas de fluxo de dados.	110
Figura 38Método <code>fire()</code> de um ator polimórfico.	111
Figura 39As etapas da análise da ferramenta FASI.	114
Figura 40Grafo de métodos para o código da Figura 38.	116
Figura 41Conexão genérica entre nós do GFC.	124
Figura 42Exemplo de ciclo em uma topologia SDF.	127
Figura 43Método <code>fire()</code> do ator <code>NonStrictLogicFunction</code> .	129
Figura 44Estrutura genérica de um ator atômico no modelo SR.	131
Figura 45Método <code>fire()</code> do ator <code>NonStrictDelay</code> adequado ao modelo SR.	134
Figura 46Exemplo de composição no modelo SDF.	137
Figura 47Trecho de código do ator <code>SequenceToArray</code> .	139
Figura 48Trecho de código do ator <code>RecordAssembler</code> .	141
Figura 49Código fonte de nível 0 do codificador.	144
Figura 50Ponteiros para os pacotes do codificador.	145
Figura 51O procedimento implementando a análise LPC.	146
Figura 52Agora hierárquico de nível 0 do codificador.	147
Figura 53Método <code>fire()</code> do ator <code>FrameGen</code> .	148
Figura 54Topologia do ator <code>Coder</code> .	148
Figura 55Topologia do ator <code>PreProcess</code> .	149
Figura 56Método <code>fire()</code> do ator <code>FilterAndScale</code> .	149
Figura 57Topologia do ator <code>Coder_12k2</code> .	150

Figura 58Topologia do ator LPAnalysis.	151
Figura 59Topologia do ator Find_Az.	152
Figura 60Trecho do método fire() do ator VAD_Lp.	154
Figura 61Análise dos atores hierárquicos do codificador.	154
Figura 62Trecho de código do processo referente ao ator FrameGen.	156
Figura 63Código simplificado para o processo associado ao ator Coder.	157
Figura 64Um possível escalonamento para o ator Coder.	158
Figura 65Método fire() do ator JoinAt.	159
Figura 66Modelo do sistema do trenó.	163
Figura 67Topologia de nível 0 para o sistema do trenó.	167
Figura 68Topologia do ator PlantDynamics.	168
Figura 69Topologia do ator hierárquico Crane.	169
Figura 70Trecho do método fire() para a verificação dos sensores.	170
Figura 71Trecho do método postfire() da verificação dos sensores.	171
Figura 72Topologia do ator Diagnostics.	172
Figura 73Método postfire() comum aos atores de diagnóstico.	173
Figura 74Topologia do ator hierárquico Control.	174
Figura 75Máquina de estados do ator ControlAlg.	175
Figura 76Topologia do ator CraneSystem.	175
Figura 77Resultado da análise para os atores hierárquicos do sistema de controle.	177
Figura 78Trecho de código do processo que implementa o ator CraneSystem.	179
Figura 79Topologia não hierárquica.	189
Figura 80Exemplo de topologia hierárquica.	190
Figura 81Exemplo de troca de dados entre atores.	191
Figura 82Exemplo de topologia hierárquica para ilustrar múltiplos diretores.	193
Figura 83Exemplo de código fonte de um ator atômico.	197
Figura 84Nível 0 de uma especificação executável.	199
Figura 85O ator hierárquico Singen.	199
Figura 86Topologia de nível 0.	217
Figura 87Topologia do ator CTSubsystem.	218
Figura 88Resultado da análise: árvore de atores e seus modelos válidos.	219

Lista de Tabelas

Tabela 1 - Tradução dos tipos de dados.	75
Tabela 2 - Alocação de memória para a topologia da Figura 18.	82
Tabela 3 - Resultados da análise efetuada pela ferramenta FASI em atores atômicos.	136
Tabela 4 - Resultado da análise dos atores atômicos do codificador de voz	151
Tabela 5 - Resultados de tempo de resposta das implementações do codificador de voz.	158
Tabela 6 - Resultados de quantidade de memória das implementações do codificador de voz.	160
Tabela 7 - Resultado da análise dos atores atômicos do sistema de Trenó.	174
Tabela 8 – Caracterização de processos.	207
Tabela 9 – Caracterização da primitiva mutex.	207
Tabela 10 – Caracterização da primitiva semáforo.	207
Tabela 11 – Caracterização da primitiva flag.	208

Lista de Abreviaturas e Siglas

CI – Componente Integrado

CPO – *Complete Partial Order*

CSP – *Communicating Sequential Processes*

CT – *Continuous Time*

DE – *Discrete Events*

EDO – Equação Diferencial Ordinária

EFR – *Enhanced Full Rate*

ETSI – *European Telecommunication Standards Institute*

LPC – *Linear Prediciton Codes*

PN – *Process Networks*

SDF – *Synchronous Data Flow*

SR – *Synchronous Reactive*

CAPÍTULO 1 - INTRODUÇÃO

*“Certas pessoas nunca aprendem alguma coisa,
pois logo entendem tudo.”*

Alexander Pope

1.1 Sistemas Eletrônicos Embarcados

Um sistema, ou parte de um sistema, é dito eletrônico quando este é composto por um agregado de dispositivos eletrônicos discretos ou integrados (CI). Uma classe de circuitos integrados são os baseados em programação (processadores), isto é, o circuito implementa um conjunto genérico de instruções e sua funcionalidade é determinada por um programa (conjunto de instruções) desenvolvido pelo projetista. A possibilidade de executar diferentes funções com um mesmo CI é a principal vantagens de tais circuitos sobre circuitos dedicados.

O primeiro processador desenvolvido foi o 4004, criado em 1971 pela recém fundada companhia Intel. O projeto original visava o desenvolvimento de um conjunto de CIs para uma calculadora, mas os projetistas da Intel perceberam a possibilidade de utilizar apenas um circuito, desde que este fosse genérico e programável.

Uma máquina que aproveita a flexibilidade proporcionada pela tecnologia de processadores são os computadores. Devido à sua programabilidade, este equipamento pode ser empregado para diferentes funções e em diferentes ambientes. Por exemplo, computadores com grande capacidade de processamento e confiabilidade são utilizados para estudar o genoma humano ou efetuar transações bancárias. Já computadores fisicamente pequenos e móveis (dependentes de baterias), são utilizados por exemplo para redigir teses de doutorado, desenvolver programas, ver vídeos, escutar músicas.

Entretanto, o uso de um processador não está restrito a um computador de uso genérico. Qualquer sistema eletrônico pode conter um ou mais processadores, desde que seu uso seja justificado. Um sistema cuja finalidade primordial não é computacional pode usufruir das vantagens de uma solução contendo processadores. Telefones celulares, aviões, carros, máquinas de lavar roupa, toca discos, são

exemplos de máquinas ou utensílios contendo subsistemas eletrônicos baseados em processadores. Quando um sistema eletrônico é empregado nestas condições, ele é denominado de **embarcado**. Um processador embarcado é aquele utilizado em um sistema embarcado. Contrastando com um computador de uso genérico, um sistema embarcado é desenvolvido para uma finalidade específica.

Devido à sua natureza de estar embutido em equipamentos e máquinas, às vezes de maneira não perceptível, sistemas embarcados apresentam uma grande importância econômica. Do total de aplicações que utilizam processadores, apenas 2% são computadores de uso genérico, sendo as demais embarcadas [TEN00]! Desta forma, fica evidente a importância do desenvolvimento eficiente de aplicações embarcadas.

Por ser utilizado como parte de um sistema não necessariamente computacional, uma aplicação embarcada apresenta características particulares [LEE01]:

- Sistemas embarcados apresentam em geral um comportamento reativo, isto é, operam a uma taxa ditada pelo meio físico em que estão operando;
- Exibem um alto grau de concorrência;
- Sua execução não é finita, isto é, uma vez iniciados, executam indefinidamente¹;
- Mesclam tecnologias heterogêneas.

Alguns sistemas embarcados também apresentam restrições físicas e requisitos específicos de desempenho. Em alguns casos, é possível utilizar hardware e software de uso genérico. Em outros é impossível. Por exemplo, um telefone celular possui uma restrição de área física muito grande. Não é possível utilizar bancos de memória dinâmica externos ao processador. O máximo possível são alguns circuitos de memória. Desta forma, a quantidade de memória está limitada o que influencia diretamente o seu desenvolvimento.

Considerando-se as características específicas de um sistema embarcado, torna-se necessário o desenvolvimento de tecnologias específicas de hardware e de software. Exemplos na área de projeto de hardware são processadores com

¹ Desde que haja energia e não ocorram falhas.

arquitetura e construção específica para determinadas classes de aplicação, barramentos de comunicação e tecnologias de sensores. Na área de software, compiladores com otimizações específicas, bibliotecas de software básico e sistemas operacionais otimizados são exemplos de tecnologias voltadas a sistemas embarcados.

A programação de aplicações embarcadas também é influenciada pelas particularidades e requisitos específicos. Linguagens de programação convencionais procuram se distanciar da unidade física onde operam através de um modelo abstrato de máquina. Isto está na direção oposta da programação embarcada, pois por definição, tais sistemas tem uma forte interação com o meio físico. Diversas extensões de linguagens ou novas linguagens foram desenvolvidas para estes sistemas [EDW00].

Um modelo de programação que se destaca na comunidade embarcada é o de processos comunicantes²[LEW95][LEA96]. Ele permite o uso de linguagens convencionais de programação. Um sistema operacional embarcado fornece a possibilidade de mais de um fluxo de execução, através de processos. Processos podem interagir através de variáveis compartilhadas ou outros mecanismos mais elaborados, construídos sobre tais variáveis. Primitivas de sincronização estão disponíveis a fim de evitar situações de conflito. As duas principais vantagens deste modelo são:

- Suporta a programação reativa;
- Apresenta um modelo de concorrência de fácil entendimento.

O modelo de processos comunicantes é o padrão para a indústria de software embarcado, haja visto a quantidade de sistemas operacionais embarcados disponíveis no mercado [FRI01].

1.2 Paradigma de Interação de Atores

O modelo de processos comunicantes é ideal para sistemas de pequeno porte

² Utilizo o termo processo como tradução para *thread*.

ou que apresentem processos independentes ou com pouca interação. Entretanto, para sistemas com um número razoável de processos (>10) e com forte interação, esta abstração apresenta algumas desvantagens. Dentre elas, destaco:

- Possibilidade de não-determinismo, dificultando a validação do sistema;
- Primitivas de sincronização de baixo nível aumentam a complexidade do software e dificultam seu entendimento;
- Possibilidade do sistema entrar em *deadlock*. Por exemplo, situações não previstas de acesso a recursos ou má escolha da prioridade de processos são causas de *deadlock*;
- Dificuldade de se fazer previsões com relação ao tempo de resposta do sistema, devido a imprevisibilidade da execução;
- Penalidade associada à criação, troca de contexto entre processos e uso de primitivas de sincronização pode não ser desprezível.

Além destas desvantagens, o paradigma de processos foi concebido originalmente como uma maneira de programar sistemas genéricos que apresentam concorrência. Assim, este paradigma possui pouca proximidade semântica com aplicações embarcadas, isto é, o modelo de processos é demasiadamente genérico. Desta forma, ele não é adequado para capturar a heterogeneidade presente em sistemas embarcados.

Uma alternativa ao uso de processos é o paradigma de interação de atores. Este paradigma foi inicialmente proposto por Carl Hewitt [HEW77] e aprimorado por Gul Agha [AGH86]. A principal entidade deste paradigma é um ator: um elemento que captura uma função. Associadas a um ator estão entidades denominadas de portos. Através dos portos, um ator obtém valores para o cálculo de sua função. Portos de atores podem ser conectados, formando uma rede de comunicação, isto é, um ator produz dados utilizados por outros atores. A interação com o meio externo se dá através de portos especiais. Desta forma, o paradigma de interação de atores privilegia a troca de dados entre suas entidades, ao invés do chaveamento do fluxo de execução.

A um ator podem estar associados atributos indicando propriedades e

características. Alguns atributos estão associados a variáveis. São denominados de parâmetros. O valor de um parâmetro pode ser especificado pelo desenvolvedor e alterado durante a execução da especificação. Parâmetros são fundamentais na reutilização de um ator por diferentes sistemas.

O ordenamento da execução dos atores e os mecanismos de troca de dados são definidos por um modelo computacional, isto é, um conjunto de regras e definições [DAV01]. Em determinados casos, é possível associar mais de um modelo a uma rede de atores. Isto é talvez a característica de maior importância do paradigma de interação de atores, uma vez que permite capturar a heterogeneidade de sistemas embarcados.

Exemplos de ambientes que suportam o paradigma de atores incluem Ptolemy II³, SystemC⁴, SCADE⁵ e Simulink⁶. Estes últimos dois são ambientes comerciais utilizados em sistemas industriais.

1.3 Objetivos e Contribuições

Ao mesmo tempo que o paradigma de interação de atores apresenta vantagens, ele também trás novos problemas ao desenvolvedor. A criação de uma especificação baseada em atores implica em efetuar uma decomposição funcional e escolher um ou mais modelos computacionais para coordenar os atores. Conhecer todos os modelos computacionais e utilizá-los adequadamente pode não ser uma tarefa trivial.

A maneira tradicional de se escolher um modelo computacional é unicamente baseado na experiência do desenvolvedor. Isto implica que este conheça, com certo grau de detalhamento, a semântica de cada modelo computacional que eventualmente possa ser utilizado, e o ambiente de interação de atores utilizado no projeto.

O problema com a solução tradicional para a escolha de um modelo computacional é que caso um modelo não seja adequado, será responsabilidade

³ ptolemy.eecs.berkeley.edu

⁴ www.systemc.org

⁵ www.estrel-technologies.com

⁶ www.mathworks.com

única do desenvolvedor adequa-lo para a captura do comportamento desejado. Além disto, a escolha do modelo torna-se uma tarefa a mais a ser controlada pelo projetista, sobrecarregando-o.

A principal contribuição desta tese é o desenvolvimento de uma metodologia para auxiliar o desenvolvedor na escolha de um modelo computacional. A pergunta fundamental que qualquer metodologia desta natureza deve responder é:

Dado uma decomposição funcional de um sistema e um conjunto de modelos computacionais, qual conjunto de modelos devo utilizar ?

A fim de estudar uma resposta para esta pergunta, desmembro-a em duas perguntas fundamentais:

1. Quais modelos computacionais posso utilizar ?
2. Qual é o melhor (conjunto de) modelo(s) ?

Nesta tese, pretendo dar uma contribuição para responder estas duas perguntas. Para tal, escolhi um conjunto relevante de modelos computacionais e bem descritos na literatura. No capítulo 2, descrevo informalmente a semântica de cada modelo e da interação entre modelos afim de facilitar o entendimento dos capítulos subsequentes. Utilizo alguns exemplos para ilustrar os principais conceitos de cada modelo computacional. Devido a limitação de espaço, a descrição de cada modelo se concentra nos aspectos mais importantes. O leitor interessado pode obter descrições mais detalhadas sobre cada modelo computacional nas referências bibliográficas que apresento no final deste texto.

Em seguida, no capítulo 3, especifico todos os passos de uma metodologia de projeto que traduz uma especificação de atores em uma de processos comunicantes, para então implementá-la utilizando técnicas convencionais. O objetivo desta metodologia é permitir a caracterização das implementações obtidas, e assim, comparar diferentes soluções iniciais (diferentes conjuntos de modelos). É fundamental ressaltar que meu objetivo ao especificar uma metodologia, que a partir de descrições baseadas em atores produza implementações em software, é prover ao

desenvolvedor uma maneira de responder a segunda pergunta (qual o melhor modelo ?). Desta forma, a metodologia não contempla várias alternativas de arquiteturas alvo, fases de projeto e tecnologias. Estes temas são pertinentes a trabalhos futuros.

No capítulo 4, descrevo uma estratégia e sua implementação em uma ferramenta de análise para determinar se uma especificação executável é válida com respeito a dois modelos computacionais em particular. Desta forma, através do uso desta ferramenta, o projetista é auxiliado no processo de associação de modelos computacionais a uma especificação e determinar se a especificação está bem construída. Isto é fundamental para responder a primeira pergunta (quais modelos posso utilizar ?)

Apresento no capítulo 5 dois estudos de caso de natureza distinta a fim de testar a metodologia de projeto e validação de uma especificação.

Além destes capítulos, este texto possui dois anexos com informações importantes para o entendimento deste texto. Para o leitor que não esteja familiarizado com tais conceitos, é então necessário referir-se a estes anexos, nos momentos que indiquei ao longo do texto.

Destaco as principais contribuições desta tese:

1. Desenvolvimento de uma ferramenta de software capaz de determinar modelos computacionais válidos para uma dada especificação executável;
2. Estudos de casos demonstrando a criação, validação e implementação de uma especificação executável com diferentes modelos computacionais;
3. Detalhamento de uma metodologia de projeto de software embarcado a partir de especificações baseadas na interação de atores.

Para o desenvolvimento deste trabalho, empreguei o ambiente Ptolemy II para a criação de especificações executáveis baseadas na interação de atores. As principais razões para esta escolha foram:

- disponibilidade do código fonte;
- maturidade do código fonte;
- extensa documentação;

- utilização prévia no meu trabalho de mestrado [JEU00].

Desta forma, os algoritmos e técnicas que desenvolvi são baseados na implementação deste ambiente. Entretanto, é possível aplicar este trabalho a outros ambientes similares.

1.4 Trabalhos Relacionados

Esta tese envolve principalmente cinco tópicos relacionados ao projeto de software:

- Linguagens de programação e ambientes de projeto;
- Modelos computacionais;
- Técnicas de validação;
- Técnicas de implementação;
- Sistemas embarcados.

Desta forma, há vários trabalhos que estão relacionados ao que desenvolvi nesta tese. Uma lista inicial pode ser obtida na bibliografia presente no final deste texto. Entretanto, existem poucos trabalhos na literatura que abordam a escolha de um modelo computacional ou a validação de uma especificação com relação a um modelo computacional. Somente recentemente surgiram alguns trabalhos diretamente relacionados ao objetivo desta tese. Destaco:

- [NEU04]: descreve um sistema capaz de especializar automaticamente uma especificação executável, isto é, produzir um código fonte em linguagem Java independente do ambiente Ptolemy II, para especificações descritas em um modelo computacional em particular. Este trabalho está relacionado ao processo de tradução de uma especificação executável que proponho no capítulo 3. Em ambos os casos, é necessário especificar todas as tarefas necessárias para a tradução. Entretanto, o objetivo principal de [NEU04] é aumentar a eficiência da execução de uma especificação. No caso desta tese, o objetivo é produzir

implementações de software embarcado;

- [JAN03]: trata-se de um livro que contém a descrição de várias características de modelos, com diversos exemplos. Desta forma, ele é útil como forma de auxiliar a escolha de um modelo computacional, mas utilizando a solução tradicional, isto é, o projetista decidindo com base em sua experiência;
- [WAN03]: este trabalho está diretamente relacionado a ferramenta de análise que descrevo no capítulo 4. Trata-se de uma proposta de algoritmos para a análise estática de especificações de atores escritos na linguagem CAL. Apenas um modelo computacional é considerado e sua análise é parcial. Entretanto, o autor não apresenta resultados, apenas um exemplo de análise para um ator.

CAPÍTULO 2 – MODELOS COMPUTACIONAIS

“Você não é burro, só foi mal adestrado.”

Dito popular.

2.1 Introdução

Durante as décadas de 1930 e 1940, um tema de grande interesse no campo da matemática era capturar precisamente o conceito de “função computável”. Procurava-se entender que classe de funções seriam estas e descrever métodos precisos para resolvê-las. Entre outros, os matemáticos Alan Turing⁷ e Alonzo Church⁸ desenvolveram independentemente soluções para estes questionamentos [HOP79][GOR94]. Seus trabalhos não produziram nenhuma máquina no sentido físico, mas sim, **modelos de computação**, isto é, teorias matemáticas que procuravam capturar precisamente o conceito de “função computável”. Turing demonstrou que seu modelo de máquina universal era capaz de capturar todas as funções parcialmente recursivas, isto é, as funções ditas computáveis. Ele demonstrou que qualquer outro modelo que também capturasse tais funções seria equivalente ao seu. Desta forma, todo computador clássico⁹ construído até o presente momento pode ser considerado uma “máquina de Turing envenenada”.

A despeito do resultado da equivalência de Turing, diferentes modelos computacionais são úteis. A razão para este fato é que o modelo de Turing é construído sobre primitivas bastante simples. Apenas 7 instruções são necessárias no modelo de Turing. Para várias funções, descrevê-las utilizando apenas tais instruções torna-se demasiadamente trabalhoso.

A descrição da semântica de uma linguagem de programação é um exemplo onde um modelo de computação mais elaborado é necessário. Uma solução é criar um modelo computacional abstraindo-se a máquina alvo da linguagem, e descrever precisamente os comandos e construções da linguagem sobre esse modelo. Tal solução é conhecida por semântica operacional [TEN91].

⁷ Turing desenvolveu o modelo que se conhece por máquina de Turing.

⁸ Church criou o modelo conhecido por cálculo lambda.

⁹ Faço aqui a distinção entre o modelo clássico, isto é de Turing, e um modelo de computador quântico [NIE00].

A programação de software embarcado também utiliza linguagens de programação, algumas inclusive sendo as mesmas¹⁰ utilizadas para software não-embarcado. Entretanto, como o software embarcado apresenta características específicas e distintas do não-embarcado, modelos computacionais próprios para sua descrição se fazem necessários.

Da mesma forma que um modelo computacional é utilizado para descrever a semântica de uma linguagem de programação, ele pode ser utilizado para definir a semântica de uma especificação baseada na interação de atores. A Figura 1 apresenta um ator e um diagrama genérico que captura a sintaxe de uma especificação baseada no paradigma de interação de atores, na forma de um grafo direcionado.

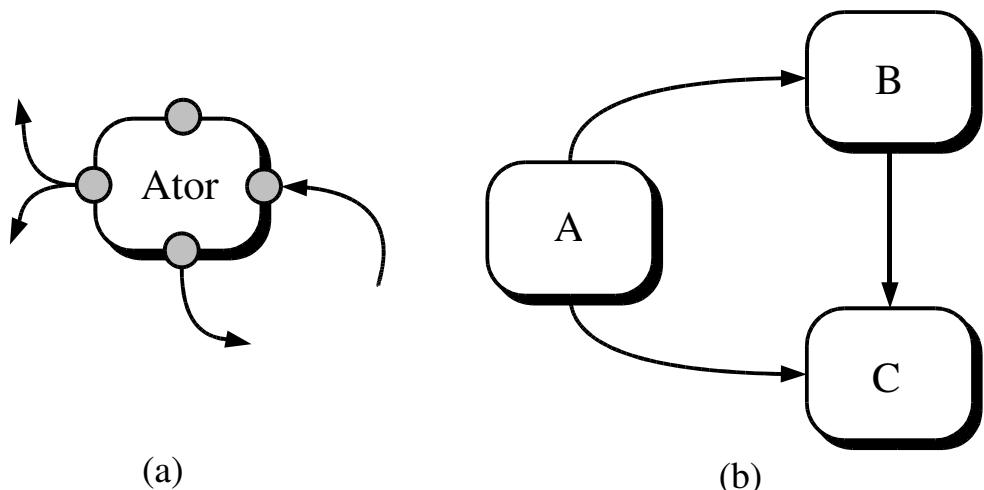


Figura 1(a) um ator; (b) um grafo exemplificando a interação entre atores.

A Figura 1 (a) detalha os elementos sintáticos de um ator: a função do ator (retângulo), portos (círculos) e relações (setas associadas a círculos). Uma interação entre atores pode ser estabelecida quando há uma relação entre eles. Relações entre atores são direcionadas e representadas através de tuplas de portos. Uma relação sempre possui um porto fonte e um ou mais portos de destino. Um porto que é fonte de uma relação é denominado de saída, enquanto que um porto de destino é denominado de entrada. Diferentes relações podem ser especificadas através do uso de diferentes portos. Mais de uma relação pode ser associada a um porto.

Na Figura 1 (b), um grafo é utilizado para exemplificar um sistema contendo

¹⁰ Ou com algumas modificações.

três atores¹¹¹². Grafos como o da Figura 1 (b) são denominados de **topologias**. O papel do modelo computacional em uma topologia é determinar a natureza dos atores e das relações entre eles. Por exemplo, para um modelo computacional de um computador convencional baseado em instruções, cada ator pode ser considerado como uma instrução de um programa seqüencial e a relação entre eles um ordenamento entre as instruções.

A Figura 1 (b) apresenta um exemplo de um sistema fechado. Como consequência, existe pelo menos um ator que não é destino¹³ de nenhuma relação do grafo e pelo menos um ator que não é origem de nenhuma relação. Quando uma topologia possuir relações onde o destino ou a fonte de pelo menos uma relação não for um ator da topologia, ela é denominada de sistema aberto.

Um ator pode ser classificado como atômico ou hierárquico. Um ator atômico é descrito em uma linguagem de programação ou em uma linguagem específica [EKE03][STR]. Desta forma, a semântica da descrição da função é determinada pelo modelo computacional associado a linguagem utilizada. Um ator hierárquico contém uma topologia de atores. Em uma topologia de atores, não há restrições sobre quantos e quais atores devem ser atômicos ou hierárquicos.

Nos demais itens deste capítulo descrevo brevemente seis modelos computacionais que utilizei durante o desenvolvimento deste tese. Embora exista uma grande variedade de modelos computacionais descritos na literatura, estes seis modelos são de grande importância para o projeto de sistemas embarcados [DAV01] [JAN03]. Todos estão implementados no ambiente de especificações executáveis Ptolemy II. O Anexo A apresenta uma breve descrição deste ambiente.

2.2 Modelo de Fluxo de Dados

O modelo de fluxo de dados foi desenvolvido originalmente para descrever e analisar programas paralelos [DEN74]. Neste modelo, um ator implementa alguma

11 A fim de tornar a figura mais simples, não são mostrados os portos dos atores.

12 Precisamente, um grafo representando uma topologia contém **instâncias** de atores. Entretanto, uso somente o termo ator quando não for necessário discriminar entre diferentes instâncias de um mesmo ator. Vide Anexo A.

13 Um ator é destino (origem) de uma relação se algum de seus portos de entrada (saída) pertencer a tal relação.

operação¹⁴ sobre um conjunto de dados e a relação entre atores indica dependência de dados. Conceitualmente, cada conexão entre dois atores da topologia contém uma fila unidirecional utilizada para a comunicação de dados. É introduzido o conceito de **ativamento** de um ator: um ator só efetua sua operação quando todos os portos de entrada possuírem pelo menos um dado. Um ator nesta condição é dito **pronto**. O ativamento de um ator é atômico, isto é, a execução da operação associada não é interrompida durante o ativamento do ator.

2.2.1 Fluxo de Dados Síncrono

O modelo computacional de fluxo de dados síncrono (SDF) [LEE87] é um caso particular de modelo de fluxo de dados. O modelo SDF, além de adotar a semântica de fluxo de dados, determina que seja especificado para cada ator quantos dados são consumidos e produzidos a cada ativamento nos respectivos portos de entrada e de saída. Estes valores (também chamados de taxas de amostragem) são números inteiros e não podem ser alterados ao longo de uma execução¹⁵. Um ator no modelo SDF está pronto somente quando existir, em cada porto de entrada, pelo menos a quantidade de dados especificado pela respectiva taxa de amostragem. Os portos de um ator podem apresentar taxas de amostragem distintas. Quando todas as taxas de amostragem de um ator forem iguais a 1, o ator é dito **homogêneo**. Quando uma especificação contém apenas atores homogêneos, ela é dita homogênea.

A noção de ativamento do modelo de fluxo de dados induz a uma execução baseada em escalonamento, isto é, a determinação da ordem de execução de cada ator. Uma característica importante do modelo SDF é que ele apresenta um escalonamento estático, sendo possível determinar antes do início da operação do sistema o número de vezes que cada ator deve ser ativado e alguma ordem de ativamento dos atores. Existem algoritmos para obter escalonamentos seqüenciais e paralelos [PIN95]. Neste trabalho, utilizei a versão dos algoritmos para escalonamento seqüencial, pois não utilizei arquitetura multi-processadas.

¹⁴ Nos primeiros modelos de fluxo de dados, tais operações normalmente eram simples, como somas e multiplicações. Em outros modelos, tais operações podem ser funções complexas.

¹⁵ O modelo de fluxo de dados síncrono com parâmetros (PSDF) [BHA02] permite a modificação controlada das taxas de amostragem.

Escalonamento SDF seqüencial

A Figura 2 apresenta a topologia da Figura 1 (b) anotada com dois conjuntos diferentes de taxas de amostragem.

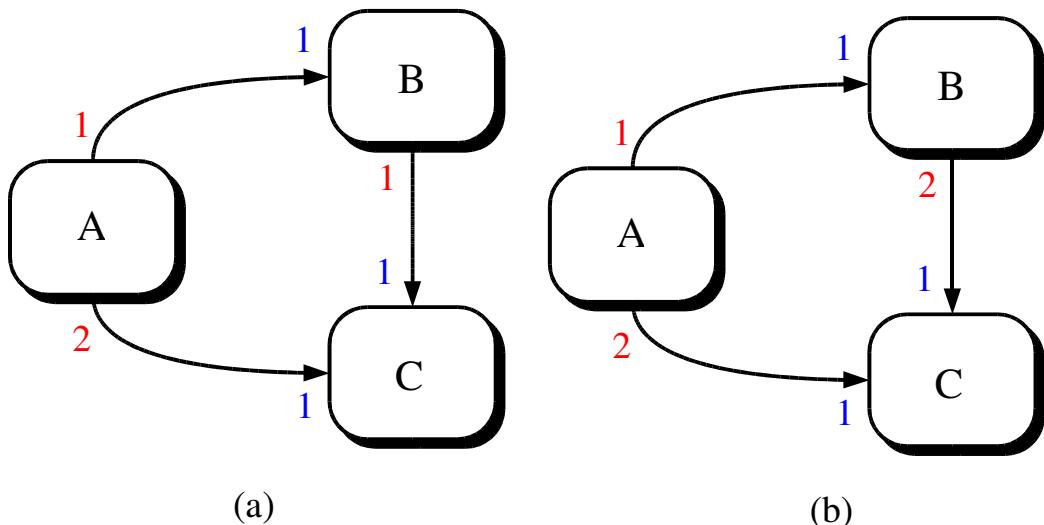


Figura 2 Dois grafos SDF. O número associado ao terminal de uma aresta próximo a um ator fonte é uma taxa de produção, enquanto que o número associado ao terminal próximo a um ator destino uma taxa de consumo. Por exemplo, o ator B consome 1 dado e produz 2 na figura (b).

Uma topologia associada ao modelo SDF pode ser descrita por uma matriz de incidência: as linhas representam as arestas e as colunas os atores. Cada posição da matriz é uma taxa de amostragem: positiva quando o ator produz um dado na respectiva aresta, negativa quando ele consome. Esta matriz é denominada de matriz de topologia. A Figura 3 apresenta as matrizes de topologia para os grafos da Figura 2.

$$\Gamma_a = \begin{bmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \quad \Gamma_b = \begin{bmatrix} 1 & -1 & 0 \\ 2 & 0 & -1 \\ 0 & 2 & -1 \end{bmatrix}$$

(a)
(b)

Figura 3 Matrizes de topologia dos grafos da Figura 2.

Durante a execução da topologia SDF, a quantidade de dados em cada fila de

comunicação irá variar. A variação de dados em cada fila pode ser descrita pela equação:

$$\mathbf{b}(n + 1) = \mathbf{b}(n) + \Gamma \mathbf{v}(n) \quad (1)$$

Em um determinado momento n , a quantidade de dados em cada fila é dada por um vetor coluna $\mathbf{b}(n)$. Para um escalonamento seqüencial, o vetor coluna $\mathbf{v}(n)$ indica qual ator é ativado no instante n .

O valor do vetor \mathbf{b} quando $n = 0$ indica a quantidade de dados em cada aresta antes do início da execução.

Para se determinar um escalonamento de uma topologia SDF, é necessário verificar se a matriz de topologia é consistente, ou seja, se as taxas de amostragem especificadas são válidas. Foi demonstrado em [LEE87] a condição necessária para a validade da matriz: o número de linhas ou colunas linearmente independentes (*rank*) deve ser igual ao número de atores menos um. Por exemplo, o *rank* da matriz Γ_a é igual a três, enquanto que o *rank* da matriz Γ_b é igual a dois. Desta forma, a topologia mostrada na Figura 2 (a) não possui um conjunto de taxas de amostragem válidas, enquanto que a da Figura 2 (b) sim.

Quando a matriz de topologia for válida, existe um vetor coluna \mathbf{q} que satisfaç:

$$\mathbf{q}\Gamma = \mathbf{0} \quad (2)$$

Para a matriz Γ_b , $\mathbf{q} = J[1 \ 1 \ 2]^T$, para qualquer J positivo. O vetor \mathbf{q} determina quantas vezes cada ator deve ser ativado para a obtenção de um escalonamento periódico, ou seja, um escalonamento que quando completado, faz com que o vetor \mathbf{b} retorne ao seu valor inicial. Uma **iteração** no modelo SDF é então definida como um período do escalonamento.

Mesmo quando a matriz de topologia é consistente, é possível que a topologia SDF não esteja correta. Isto ocorre quando existirem ciclos no grafo sem a quantidade adequada de dados no momento inicial da execução (um valor inicial de \mathbf{b} inadequado). Essa condição é denominada de *deadlock*. A Figura 4 ilustra dois grafos em *deadlock*.

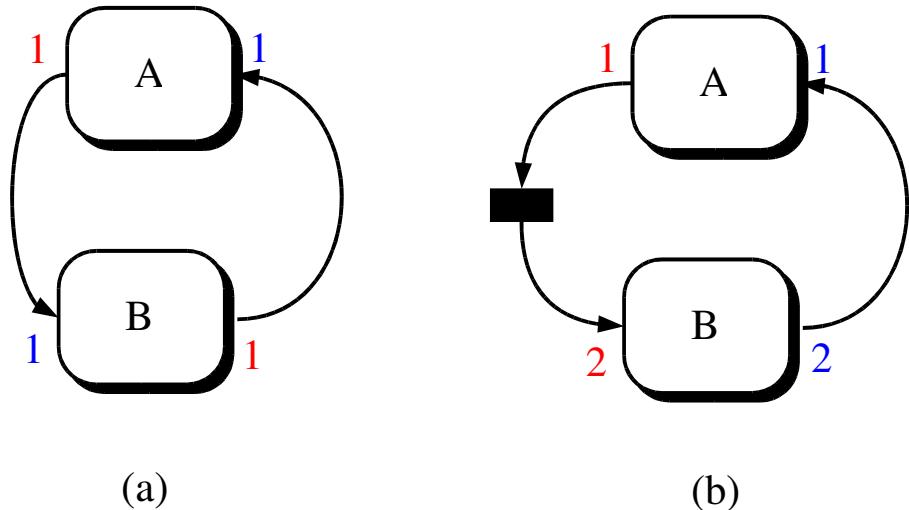


Figura 4 Duas topologias SDF inválidas com matrizes consistentes.

O retângulo da Figura 4(b) representa um ator de atraso unitário, ou seja, um ator que na iteração n envia o dado recebido na interação $n - 1$ (a entrada no vetor b com $n = 0$ para esta aresta é maior que zero). Ambas as topologias da Figura 4 não possuem condições iniciais válidas: a da Figura 4 (a) não possui nenhum atraso, e a da Figura 4 (b) apresenta um atraso de apenas um dado, o que é insuficiente. A quantidade de dados produzidos inicialmente e a posição dos atores de atraso devem ser determinadas pelo desenvolvedor da especificação.

Quando a topologia SDF for consistente e com a quantidade necessária de atrasos, o seguinte algoritmo pode ser usado para a obtenção de um escalonamento seqüencial [LEE87]:

Algoritmo 2.1: Escalonamento SDF seqüencial:

Entrada: uma topologia anotada com taxas de amostragem.

Saída: uma lista contendo uma seqüência de atores.

1. Encontre o menor vetor \mathbf{q} ;
2. Obtenha uma lista L de todos os atores da topologia;
3. Percorra a lista L e para cada ator i pronto, adicione-o à lista de saída;
4. Caso um ator i tenha sido escalonada q_i vezes, removê-o da lista L ;
5. Caso nenhum ator i da lista L esteja pronto, ocorreu um *deadlock*. Vá para 8;

6. Caso a lista L esteja vazia, vá para 8;
7. Do contrário, vá para 3;
8. FIM.

Para encontrar o vetor \mathbf{q} do passo 1, o seguinte método é utilizado:

Algoritmo 2.2: cálculo do número de repetições de um ator SDF.

Entrada: uma topologia anotada com taxas de amostragem.

Saída: para cada ator da topologia, o número de vezes que deve ser ativado por iteração.

1. Escolha aleatoriamente um ator a e assuma que ele é ativado uma única vez, ou seja, $q_a = 1$;
2. Para um ator b adjacente a a , calcule o valor de $q_b = (q_a \cdot p_a) / c_b$, onde p_a é a quantidade de dados produzidos pelo ator a na aresta e c_b a quantidade de dados consumidos pelo ator b na aresta. O valor obtido pode ser uma fração, mas sempre racional;
3. Repita o passo 2 para um ator adjacente ao ator b ;
4. Uma vez obtidos todos os valores de q , encontre o menor denominador comum destes valores a fim de torná-los inteiros;
5. FIM.

É fácil perceber que o algoritmo 2.2 irá encontrar o vetor \mathbf{q} em tempo linear ao número de atores e arestas da topologia. Uma consequência importante do escalonamento periódico é que a topologia SDF pode ser executada indefinidamente com capacidade limitada e conhecida de memória.

2.3 Rede de Processos Kahn

O modelo Rede de Processos Kahn (PN) [KAH74][KAH77] considera cada ator como sendo uma unidade computacional independente, com sua própria

autonomia de execução. Uma relação entre atores representa um conjunto de canais¹⁶ de comunicação unidirecional, utilizado para troca de informação entre os atores. Conceitualmente, um ator possui dois comandos para utilizar estes canais: *wait* e *send*¹⁷. O comando *wait* suspende a execução de um ator até que algum dado esteja disponível em **um** canal específico. O comando *send* é utilizado para enviar um dado pelo canal especificado no comando. São impostas as seguintes restrições a uma topologia no modelo PN:

1. A troca de dados através de canais é a única forma de interação entre atores;
2. Um ator utiliza o comando *wait* para esperar dados em apenas um canal. Não é permitido testar a presença de dados em qualquer canal;
3. Não é permitido que mais de um ator envie um dado por um mesmo canal.

O modelo PN considera que um canal carrega uma seqüência infinita de dados, também chamada de história, de algum tipo **T**. A função de um ator é então definida como sendo um mapeamento entre as histórias dos canais de entrada para histórias dos canais de saída¹⁸.

A definição da semântica do modelo PN associa a história de um canal a uma cadeia¹⁹, possivelmente infinita, de elementos do conjunto **T**. As possíveis histórias de um canal formam então uma CPO²⁰ T^w . Um ator deve implementar uma função contínua sobre as histórias dos canais de entrada. Desta forma, uma topologia pode ser vista como um sistema de equações do tipo:

$$X_o = f(X_{i1}, X_{i2} \dots X_{in}), \quad (3)$$

onde X_o é uma variável que representa a história de um canal de saída do

16 Um canal é definido somente entre um ator de destino e um de origem. Assim, uma relação que contenha mais de um ator de destino especifica mais de um canal.

17 Implementados respectivamente pelos métodos *get()* e *send()* do ambiente Ptolemy II.

18 Um canal de entrada (saída) é aquele associado a um porto de entrada (saída) do ator.

19 Vide o anexo B para um resumo das definições básicas sobre teoria de ordens.

20 Vide o anexo B para um resumo das definições básicas sobre teoria de ordens.

ator, X_{in} é uma variável associada a história de algum canal de entrada e f a função (contínua) do ator. Kahn cita um teorema em [KAH74] que garante que tal sistema de equações apresenta uma solução mínima e única. Em outros termos, satisfeitas as condições da semântica do modelo PN, a sua execução é determinística.

As condições impostas pela semântica do modelo PN podem ser satisfeitas por construção. A Figura 5 (a) apresenta um exemplo de uma topologia no modelo PN.

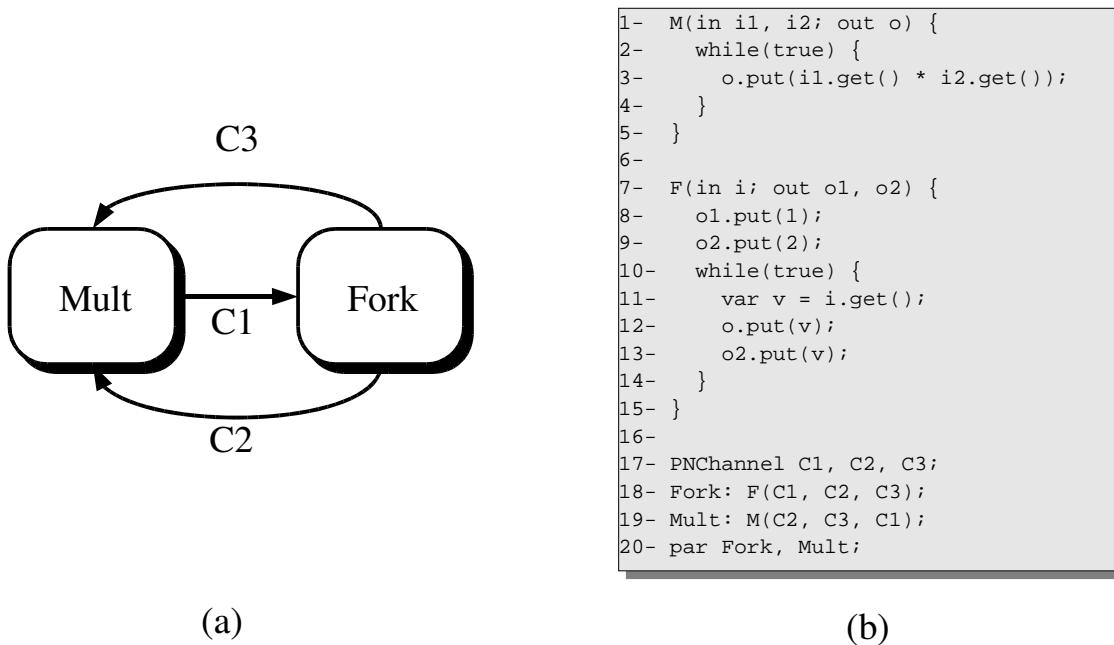


Figura 5(a) Uma topologia com dois atores PN; (b) Pseudo-código para a topologia da figura (a).

A topologia da Figura 5 (a) contém dois atores e três canais de comunicação. A Figura 5 (b) apresenta um pseudo-código para a topologia da Figura 5 (a). A função **M** (linha 1) e **F** (linha 7) implementam respectivamente as funções dos atores *Mult* e *Fork*. A declaração de cada função determina quais são os canais de entrada (*in*) e saída (*out*). Na linha 17, três canais são declarados e associados às funções dos atores na linha 18 e 19. O comando *get()* implementa a leitura bloqueante²¹ no canal, enquanto que o comando *put()* escreve um novo dado no canal. Note que apenas um ator pode escrever em um canal, e apenas um ator pode ler de um canal. Para o exemplo da Figura 5, a história do canal C1 é uma progressão geométrica, e

²¹ Leitura bloqueante é aquela que quando um canal estiver vazio, a função do ator que executou o comando de leitura é suspensa até que o canal possua algum dado.

as histórias dos canais C2 e C3 são idênticas.

O exemplo da Figura 5 possui uma execução determinística, pois satisfaz às condições da semântica do modelo PN. Além das restrições sobre o acesso a canais de comunicação, todos os atores implementam funções contínuas. Isto é obtido através da leitura bloqueante sobre um canal, conforme demonstrado em [LEE95].

Duas perguntas podem ser feitas sobre qualquer topologia PN:

1. Qual é a capacidade máxima de armazenamento necessária para cada canal PN ?
2. É possível que a execução de todos os atores esteja interrompida em algum momento, pois todos tiveram suas execuções suspensas ?

Ambas as questões acima são importantes pois muitos sistemas embarcados são projetados para nunca serem desligados, processando continuamente. Uma topologia PN para tais sistemas deve executar ininterruptamente.

A primeira questão procura saber se é possível executar uma topologia PN com tamanho limitado de memória. Do contrário, em algum dado momento, o sistema irá parar de executar, ou executar com comportamento errôneo. Foi demonstrado em [PAR95] que a resposta para esta pergunta é um problema não computável. Desta forma, é impossível determinar para uma topologia PN qualquer a capacidade máxima necessária de cada um de seus canais de comunicação. Em [LEE95], um ator PN é interpretado como uma seqüência possivelmente infinita de ativações de um ator de fluxo de dados implementando a mesma função. Caso os atores de fluxo de dados equivalentes sejam todos válidos para o modelo SDF, então podemos afirmar que os canais PN têm capacidade máxima limitada, e inclusive, determinar o seu valor.

A segunda questão é referente à possibilidade de uma topologia PN interromper sua execução, sem que isto seja desejado. Esta condição é denominada de *deadlock*. Novamente, para uma topologia PN qualquer, é impossível criar um algoritmo para verificar a possibilidade de um *deadlock*.

Em [PAR95], Parks propõe uma solução para abordar as duas questões acima. Ela compreende em tornar a escrita a um canal PN bloqueante²². Desta forma,

²² Quando um canal estiver cheio de dados, isto é, com uma quantidade de dados igual a sua

no início da execução da topologia, a capacidade de cada canal é igual a 1 dado. Quando todos os atores estiverem com suas execuções suspensas, é verificado se existe pelo menos um ator que está bloqueado na escrita. Caso negativo, então a topologia alcançou um *deadlock*. Caso contrário, a execução é dita estar em um *deadlock* artificial. Nesta situação, todos os canais que estão cheios têm suas capacidades aumentadas em uma unidade, reestabelecendo assim a execução dos respectivos atores bloqueados, e a execução da topologia PN pode prosseguir. É argumentado em [PAR95] que tal solução é capaz de permitir a execução de uma topologia PN que necessite de capacidade limitada de canais.

A implementação do modelo PN no ambiente Ptolemy II, além de adotar a escrita bloqueante, introduz uma noção de passagem de tempo. O tempo é representado por um valor real, sempre crescente, e compartilhado entre todos os atores. Um ator possui então uma operação adicional: requisitar a suspensão de sua execução até que um certo instante de tempo seja alcançado ou ultrapassado. Com esta extensão ao modelo, um *deadlock* artificial pode ser composto de atores bloqueados na escrita e/ou atores bloqueados esperando um instante de tempo. Quando o instante de tempo especificado for alcançado ou ultrapassado, os respectivos atores esperando tal instante têm suas execuções reestabelecidas.

2.4 Processos Seqüenciais Comunicantes

No modelo computacional Processos Seqüenciais Comunicantes (CSP) [HOA78], assim como no modelo PN, o ator possui um fluxo de execução independente. Uma relação entre atores representa um conjunto de canais de comunicação. A comunicação no modelo CSP não utiliza armazenamento de dados. Quando um ator deseja enviar ou receber um dado de outro ator relacionado, ambos devem sincronizar a fim de completar a troca. O protocolo de sincronização é conhecido como *rendezvous*. Neste protocolo, um ator A que queira enviar um dado para um ator B, tem sua execução suspensa enquanto o ator B não executar um comando para ler o dado. Após a leitura do dado pelo ator B, este permanece suspenso até o reativamento da execução do ator A. Uma situação análoga acontece

capacidade de armazenamento, a tentativa de se adicionar um novo dado ao canal interrompe a execução do respectivo ator que solicitou a escrita.

quando um ator tenta ler um dado de outro ator.

O exemplo da Figura 5 também serve para ilustrar uma topologia CSP. A única modificação necessária seria alterar a declaração da linha 17 a fim de instanciar um canal do tipo CSP.

Uma topologia CSP pode apresentar situações de *deadlock*. Considere dois atores (A e B), cada um com dois portos de comunicação (p1 e p2). O porto p1 de A está conectado ao porto p1 de B, e da mesma forma se conectam os portos p2. Um cenário de *deadlock* seria onde A está tentando enviar um dado para B através de p1, e B está tentando ler um dado de A através de p2. Determinar se uma topologia CSP qualquer é passível de *deadlock* antes da execução é um problema não computável [MAR96].

O modelo CSP permite que atores se comportem de maneira não-determinística através da introdução de expressões de comunicação condicional. Estas expressões apresentam a seguinte forma:

condição; comunicação => lista de comandos;

A condição é uma expressão que retorna um valor booleano. O cálculo da condição não pode produzir alterações no estado interno do ator. Caso o valor retornado seja falso, o restante da expressão é ignorada. Caso contrário, o comando de comunicação especificado é iniciado e quando for completado, a lista de comandos é executada.

Dois tipos de comandos que utilizam expressões condicionais são definidos: **CIF** e **CDO**. O comando **CIF** apresenta a seguinte estrutura:

```
CIF {
    condição1; comunicação1 => lista de comandos1;
}
    condição2; comunicação2 => lista de comandos2;
}
....
```

Para cada expressão condicional, as respectivas condições são verificadas. Esta verificação é feita concorrentemente, ou seja, é irrelevante a ordem das expressões condicionais. Caso uma ou mais expressões sejam verdadeiras, os respectivos comandos de comunicação são iniciados. Caso nenhum comando de comunicação esteja pronto, todo o comando **CIF** irá bloquear até que uma comunicação esteja pronta. Caso mais de uma comunicação esteja pronta, a escolha é feita de maneira aleatória. Uma vez que a comunicação é completada, a lista de comandos associada é executada e o comando **CIF** é encerrado. O comando **CIF** é ignorado quando todas as condições forem falsas.

O comando **CDO** é similar ao comando **CIF**. Entretanto, ao acabar a execução de uma das listas de comandos, o comando **CDO** é reiniciado. Este comando é encerrado apenas quando todas as condições forem falsas.

A versão do modelo CSP implementada no ambiente Ptolemy II [SYM98] apresenta uma noção de tempo similar a do modelo PN. Cada ator pode suspender sua execução até que um determinado instante de tempo seja alcançado e/ou superado. O valor temporal é avançado quando todos os atores se suspenderam ou quando todos os atores estão bloqueados devido ao *rendezvous* e pelo menos um processo se suspendeu. Nestas situações, o tempo atual é avançado o suficiente para ativar algum processo que se suspendeu.

2.5 Síncrono Reativo

O modelo computacional Síncrono Reativo (SR) é fundamentado na hipótese de sincronismo [BER84]: *o sistema deve ser capaz de produzir uma resposta a estímulos externos de forma infinitamente rápida*. Cada reação do sistema é instantânea e atômica. A evolução do tempo é dividida em uma série de **instantes** discretos, isto é, cada reação do sistema a um novo conjunto de eventos externos constitui um instante. A cada instante, a função de cada ator deve ser computada a fim de obter uma resposta ao conjunto de entradas. O modelo SR, além de implementado no ambiente Ptolemy II, é utilizado nas linguagens Lustre [HAL91] e Esterel, entre outras [BEN03].

A utilização da hipótese de sincronismo apresenta algumas dificuldades

quando aplicada a topologias que contêm ciclos. A Figura 6 ilustra alguns exemplos onde obter a solução do sistema em um dado instante não é imediato.

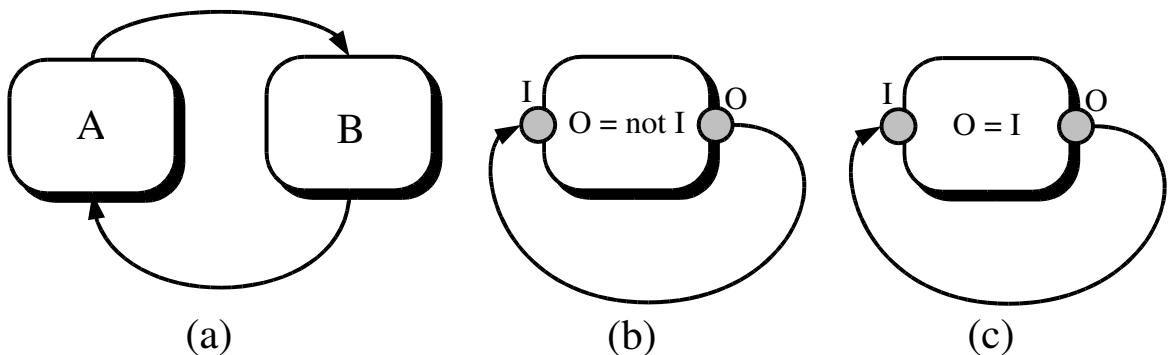


Figura 6(a) Dependência; (b) Paradoxo; (c) Ambigüidade.

Para a topologia da Figura 6 (a), qual ator deve ter sua função computada em primeiro lugar? Na Figura 6 (b), o ator implementa uma função de inversão lógica. Qual valor pode ser atribuído a relação conectando os portos I e O ? Na Figura 6 (c), o ator implementa uma função de identidade. Quantos valores podem ser associados a relação entre os portos I e O ?

A definição da semântica do modelo SR deve ser capaz de esclarecer os casos apresentados na Figura 6. Algumas soluções para estas questões já foram propostas [BEN03]. Descrevo aqui a semântica implementada no ambiente Ptolemy II para o modelo SR, proposta por Edwards em [EDW97].

Semântica do Modelo SR²³

Cada relação entre atores é representada por uma variável cujo valor pertence a uma CPO. Esta CPO é definida como sendo plana, isto é, possui apenas dois estados: indefinido e definido. A Figura 7 ilustra um diagrama para esta CPO.

23 A partir deste momento, toda a vez que me referir ao modelo SR, estou me referindo à semântica proposta por Edwards em [EDW97].

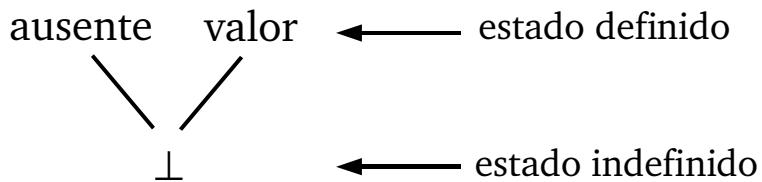


Figura 7A CPO do modelo SR.

São três os possíveis elementos da CPO: ausente, valor e \perp . Ausente significa que não há um valor na relação em um dado instante, valor representa um valor específico na relação e \perp é o limite inferior. A relação de ordem destes elementos é dado por: $\perp \subseteq$ ausente, $\perp \subseteq$ valor, ausente $\not\subseteq$ valor e valor $\not\subseteq$ ausente.

Um ator no modelo SR é associado a uma função vetorial da forma:

$$f: I \rightarrow O,$$

onde $I = I_1 \times \dots \times I_n$ e $O = O_1 \times \dots \times O_m$ são vetores de CPOs pontuais. O vetor I representa os portos de entrada do ator, enquanto que o vetor O os de saída. Uma restrição fundamental para a função de cada ator é que estas devem ser contínuas. Como a CPO do modelo SR é finita e pontual, a função de um ator deve também ser monotônica²⁴. Informalmente, isto significa que uma vez que um ator atribua um estado definido a um porto de saída, este deve permanecer inalterado durante o instante.

Executar uma topologia no modelo SR significa então encontrar um ponto fixo para um sistema de funções monotônicas sobre CPOs pontuais. Nestas condições, é garantido que tal solução existe e é unica (menor ponto fixo). Assim, o modelo SR é determinístico.

No modelo SR, um ator é classificado em dois tipos: *Strict* e *NonStrict*. Um ator *Strict* é aquele que necessita que todos os portos de entrada apresentem um estado definido a fim de definir seus portos de saída. Um ator *NonStrict* é aquele que para definir o estado de pelo menos um de seus portos de saída, não necessita que todos os portos de entrada estejam em um estado definido. Por exemplo, para a Figura 6 (a) apresentar alguma solução, pelo menos um dos atores deve ser do tipo

²⁴ Vide anexo B.

NonStrict.

Um teorema demonstrado em [EDW97] afirma que:

Seja $f:D \rightarrow D$ uma função contínua sobre um CPO pontual D. Então:

$$\text{fix}(f) = \nabla\{\perp, f(\perp), f(f(\perp)), \dots, f^k(\perp), \dots\}$$

existe e é tanto um único menor ponto fixo como o único menor ponto prefixo de f . Este teorema permite o desenvolvimento de um algoritmo capaz de obter uma solução para a topologia em um dado instante. Ele consiste em associar um estado não definido a todas as relações da topologia, menos as que não possuam fonte na topologia, e executar as funções dos atores até que nenhum porto altere mais seu estado. Temos então o algoritmo:

Algoritmo 2.3: Escalonamento dinâmico SR

Entrada: uma topologia de atores.

Saída: uma topologia de atores, onde todas as relações que não possuam destino na topologia estão em um estado definido.

1. Construa uma lista L com todos os atores do sistema;
2. Pegue o próximo ator da lista L;
 1. Verifique se ele está na lista M de atores prontos para o ativação;
 2. Se estiver, execute a função do ator;
 3. Senão,
 1. Se o ator for do tipo *Strict* e todos os portos de entrada apresentem um estado definido, atribua um valor ausente para todos os portos de saída que estejam em um estado indefinido;
 3. Caso tenha havido alguma alteração do estado de algum porto da topologia, vá para 2;
 4. FIM.

Um ator está pronto quando ele for *NonStrict* ou quando for *Strict* e todos os

portos de entrada estejam em um estado definido.

Em [EDW94], Edwards prova que este algoritmo de escalonamento irá encontrar o menor ponto fixo em um número finito de iterações. É possível desenvolver outros algoritmos para encontrar um escalonamento de uma topologia a fim de obter um menor ponto prefixo. Em particular, Edwards descreve em [EDW97] outro algoritmo capaz de calcular um escalonamento de atores antes do início da execução do sistema (escalonamento estático).

2.6 Eventos Discretos

O modelo computacional Eventos Discretos (DE) [BEN96] associa a cada relação entre atores a comunicação de um evento entre o ator fonte e os atores destino. Um evento é um par composto por um dado e um valor numérico real, denominado de *timestamp*. A finalidade do *timestamp* é estabelecer uma ordem total sobre todos os eventos produzidos durante a execução da topologia, isto é, modelar a passagem do tempo.

Um ator no modelo DE é associado a uma função, calculada em qualquer momento em que exista eventos em pelo menos um dos portos de entrada do ator. O ator pode ou não produzir novos eventos. Toda vez que um evento é produzido por um ator, o seu *timestamp* deve ser igual ou maior que o *timestamp* dos eventos presentes nos portos de entrada no dado momento do cálculo da função.

Um evento produzido não é enviado diretamente para o porto do(s) ator(es) de destino, mas sim para uma fila de eventos. Nesta fila, os eventos são mantidos em ordem crescente com base no valor do *timestamp*. O ator a ter sua função calculada em um dado instante é determinado removendo-se o próximo evento da fila²⁵ e observando-se qual é o ator de destino de tal evento. Entretanto, a semântica de eventos do modelo DE apresenta situações de conflito na decisão de qual deve ser o próximo ator a ter sua função executada. Isto ocorre na presença de eventos simultâneos, isto é, eventos para atores distintos mas com um mesmo *timestamp*. Na topologia da Figura 1 (b), caso o ator A produza um evento para o ator B e outro para o ator C com um mesmo *timestamp*, qual deve ser o próximo ator ter sua função

²⁵ Pode haver mais de um evento em um dado instante de tempo para o mesmo ator.

executada?

A solução adotada no ambiente Ptolemy II [MUL99] associa a cada ator um valor inteiro não negativo e único (prioridade). Esse valor é obtido através de uma ordem topológica de um grafo, onde os nós são os atores e os arcos indicam a dependência entre atores. Um ator depende de outro quando houver uma relação entre eles na topologia e existir pelo menos um porto de saída do ator fonte que produza eventos sem atraso²⁶ para o ator destino. Um evento é então ordenado na fila de eventos com base no *timestamp* e no valor da prioridade do ator destino do evento. Como argumentado em [MUL99], esta solução torna a execução da especificação determinística. Entretanto, é possível que o grafo gerado contenha ciclos sem atraso. Isto significa que a especificação contém um ciclo de dependência, situação que impede a execução da topologia.

A execução de uma topologia DE é controlada por um escalonador que determina qual deve ser o próximo ator a ter sua função computada. A execução de uma especificação no modelo DE é interrompida quando não há mais nenhum evento na fila ou quando um valor de tempo especificado para o fim da execução é alcançado. O algoritmo de escalonamento é composto de nove passos:

Algoritmo 2.4: Escalonamento de uma topologia DE

Entrada: (1) uma topologia onde os atores tenham sido associados a prioridades. (2)

Um valor real maior que zero onde se deve interromper a execução da topologia.

Saída: uma lista de funções de atores que foram computadas.

1. Caso a fila de eventos esteja vazia ou o instante de tempo final tenha sido alcançado, vá para o passo 9;
2. Obtenha o próximo evento da fila;
3. Atualize o instante de tempo atual para o valor do *timestamp* do evento obtido no passo 2;
4. Determine o ator de destino do evento obtido e introduza-o no respectivo porto;
5. Caso não exista nenhum outro evento simultâneo para o ator do passo 4, vá

²⁶Uma produção sem atraso significa que o *timestamp* do evento gerado é igual a dos eventos que ativaram o ator para a produção.

para o passo 7;

6. Obtenha todos os eventos simultâneos para o ator do passo 4 e introduza-os nos respectivos portos de destino;
7. Execute a função do ator até que todos os eventos em seus portos tenham sido consumidos;
8. Vá para o passo 1;
9. FIM.

É possível o algoritmo 2.4 entrar em uma iteração infinita. Isto acontece pois não há nenhum tipo de restrição sobre a função de um ator, nem sobre a produção de eventos.

2.7 Tempo Contínuo

O modelo tempo contínuo (CT) [LIU01] foi desenvolvido especificamente para capturar um sistema de equações diferenciais ordinárias (EDO). Um sistema EDO pode ser descrito através do seguinte grupo de equações:

$$\dot{x} = F(x, u, t) \quad (4)$$

$$y = G(x, u, t) \quad (5)$$

$$x(t_0) = x_0 \quad (6)$$

, onde

$t \in \mathbb{R}$, $t \geq t_0$ representa o tempo como uma variável contínua, x é uma tupla de N-dimensões representando os possíveis estados do sistema, u é uma tupla de I-dimensões representando os valores de entrada, y é uma tupla de O-dimensões representando os valores de saída, e x_0 é o estado inicial do sistema.

Um ator no modelo CT é uma função que mapeia sinais de entrada em sinais de saída, ou seja, cada ator expressa a relação matemática entre os sinais. Uma relação entre atores representa um sinal contínuo. Um sinal é uma função sobre um intervalo fechado de \mathbb{R} . A Figura 8 apresenta uma diagrama genérico de atores que

captura um sistema EDO.

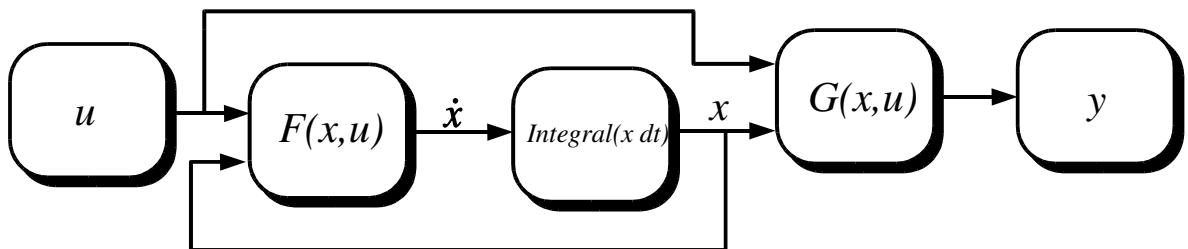


Figura 8Captura conceitual de um sistema EDO em uma topologia de atores.

O ator $Integral(x dt)$ é um ator especial que define a tupla de possíveis estados do sistema (x) a partir de sua derivada.

A execução de uma topologia CT significa calcular os valores da tupla de estados (x) e da tupla de valores de saída (y), para todos os valores de tempo a partir do instante inicial. Uma estratégia é discretizar o tempo contínuo em uma série instantes de tempo crescentes, e para cada um destes instantes, calcular através de métodos numéricos uma aproximação para o valor de cada sinal no instante. Quanto maior o número de instantes, maior a precisão de cada sinal. Existem vários algoritmos que calculam o estado e saída do sistema para um conjunto de instantes discretos [LIU01].

2.8 Interação entre Modelos Computacionais

Um modelo computacional pode ou não estar associado a um ator hierárquico²⁷. Se nenhum modelo estiver associado, o ator hierárquico deve estar necessariamente contido em uma outra topologia que seja governada por um modelo computacional. Um ator hierárquico associado a um modelo computacional é denominado de **opaco**. Um ator atômico também é classificado como **opaco**, por definição.

Quando dois atores hierárquicos opacos estiverem relacionados, temos uma situação de interação entre modelos computacionais. A Figura 9 ilustra uma situação em que um ator contido (B) em um ator hierárquico opaco (C) está relacionado a

27 Vide Anexo A

outro ator (A) contido na topologia de nível 0²⁸. É possível determinar a natureza da relação entre eles ? Qual deve ser a natureza de tal relação ?

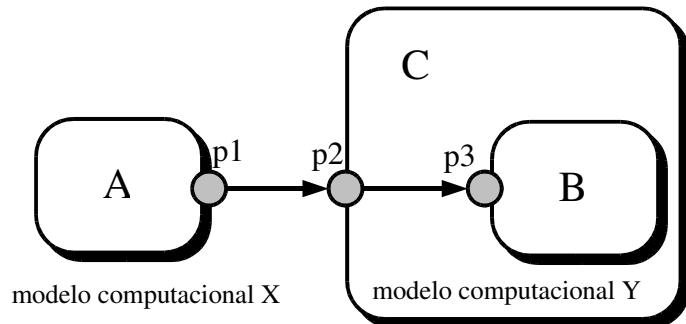


Figura 9 Interação entre atores hierárquicos com modelos distintos.

Uma estratégia para responder a tais perguntas é considerar um ator hierárquico opaco (C) como um ator atômico [DAV01]. A função do ator hierárquico contido (C) é determinada pela sua topologia. Do ponto de vista da topologia que contém o ator hierárquico opaco, este deve se comportar como um ator atômico, isto é, não há distinção entre um ator atômico e um hierárquico opaco. Na Figura 9, o modelo X controla dois atores opacos, A e C, relacionados através dos portos p1 e p2. O modelo Y controla o ator B, que se relaciona com o meio externo através através do porto p3.

Utilizando-se esta estratégia, deve-se determinar para cada par de modelo computacionais, como um ator hierárquico para um modelo computacional pode ser ao mesmo tempo ser um ator atômico para outro modelo. Nem todas as combinações de modelos computacionais são interessantes. Por exemplo, um ator hierárquico controlado pelo modelo PN em uma topologia SR não é adequado devido à hipótese de sincronismo. Nos próximos ítems desta seção, descrevo informalmente a interação para as combinações de modelos computacionais que utilizei no desenvolvimento desta tese.

28 Uma topologia de nível 0 é aquela que não está contida em nenhuma outra topologia.

CT e DE

A fim de utilizar um ator hierárquico DE em uma topologia CT, é necessário determinar a natureza das relações em ambos os sentidos. Para um porto de entrada do ator hierárquico DE, um sinal contínuo deve ser transformado em eventos discretos, e para um porto de saída, eventos produzidos devem gerar um sinal na topologia CT. Isto pode ser implementado através de atores atômicos especiais associados ao modelo CT [LIU01]. A Figura 10 apresenta um exemplo de topologia CT com um ator hierárquico DE.

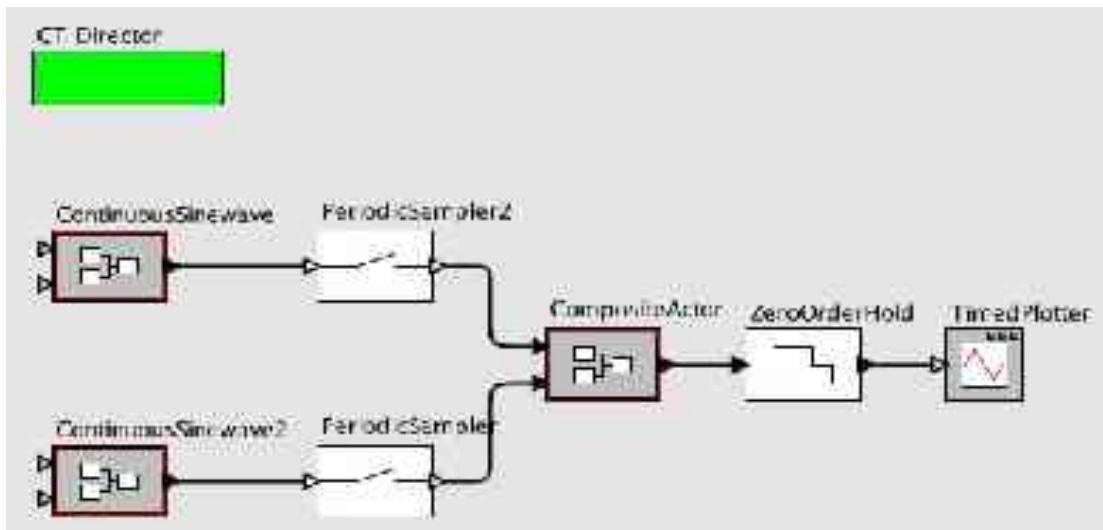


Figura 10 Um exemplo de topologia CT contendo ator DE

O ator *CompositeActor* está associado ao modelo DE. Os ator *PeriodicSampler* é responsável por traduzir um sinal contínuo em eventos discretos, através de uma amostragem periódica. O ator *ZeroOrderHold* traduz eventos discretos em um sinal contínuo, onde seu valor é modificado a cada novo evento discreto.

DE e SR

Quando um ator hierárquico SR é embutido em uma topologia DE, um instante é definido como sendo um ativação devido a presença de um e somente um evento em pelo menos um porto de entrada. A ausência de um evento em um

porto de entrada é traduzido como um estado definido e ausente. Ao final de um instante do ator hierárquico SR, os estados indefinido e definido ausente são traduzidos como a não emissão de um evento. O estado definido e com um valor corresponde a produção de um evento com o respectivo valor.

Não é permitido que mais de um evento com o mesmo *timestamp* esteja presente em qualquer porto de entrada do ator hierárquico SR, em um dado ativamento. Para a topologia DE, é considerado que todos os portos de saída do ator hierárquico SR dependam de todos seus portos de entrada.

PN e (SDF ou SR)

Um ator hierárquico SDF em uma topologia PN é ativado somente quando houver dados suficientes em todos os seus portos de entrada. Isto é obtido através da leitura bloqueante do número de dados especificados nas taxa de consumo de cada porto de entrada. A cada ativamento, uma iteração do escalonamento da topologia SDF é efetuada, e os dados produzidos são escritos nas filas associadas aos portos de saída do ator hierárquico SDF.

Um ator hierárquico SR embutido em uma topologia PN é restringido a se comportar como um ator de fluxo de dados. Desta forma, um instante ocorre somente quando houver exatamente um dado em cada porto de entrada do ator SR. Estes dados são obtidos através da leitura bloqueante das filas associadas aos portos de entrada. Caso um valor seja definido em um dado instante em algum porto de saída do ator hierárquico SR, este será escrito nas respectivas filas associadas ao porto.

CSP e (SDF ou SR)

Um ator hierárquico SDF empregado em uma topologia CSP deve obter a quantidade de dados especificados pelas taxas de amostragem através do *rendezvous* com os canais associados aos seus portos de entrada. Uma vez que todos os *rendezvous* tenham sido completados, o ator hierárquico SDF pode executar uma iteração de seu escalonamento. Os dados produzidos são escritos nos canais

associados aos portos de saída do ator SDF. Quando todos os *rendezvous* associados aos portos de saída tenham sido completados, um novo conjunto de dados pode ser lido.

Um ator hierárquico SR empregado em uma topologia CSP deve obter um dado para cada porto de entrada. Quando todos os *rendezvous* forem completados, o ator SR pode calcular a resposta ao conjunto de dados obtidos. Para os portos de saída onde valores tenham sido definidos, os respectivos dados são escritos nos canais CSP associados.

SDF e SR

Um ator hierárquico SR deve se comportar como um ator de fluxo de dados. Como a CPO do modelo SR é plana, as taxas de amostragem dos portos de entrada do ator SR devem ser homogêneas. Um instante para o ator hierárquico SR corresponde a um ativamento pela topologia SDF. Deve-se garantir que o ator hierárquico SR sempre produza um valor em seus portos de saída, pois o estado definido e ausente implica em uma taxa de amostragem igual a 0.

SR e SDF

O ator hierárquico SDF é considerado do tipo *Strict*. Como a CPO do modelo SR é plana, todos os portos de entrada e saída do ator SDF devem ter taxas de amostragem homogêneas. Um ativamento do ator SDF em um instante da topologia SR corresponde a uma iteração do escalonamento da topologia do ator SDF.

CAPÍTULO 3- METODOLOGIA DE PROJETO PARA SOFTWARE EMBARCADO

“Pesquisa básica é o que estou fazendo quando não sei o que estou fazendo.”

Werner von Braun

3.1 Introdução

Defino uma metodologia de projeto como sendo uma seqüência de níveis de abstração. Um nível de abstração representa uma etapa do projeto do sistema. A Figura 11 ilustra os elementos presentes em um nível de abstração.

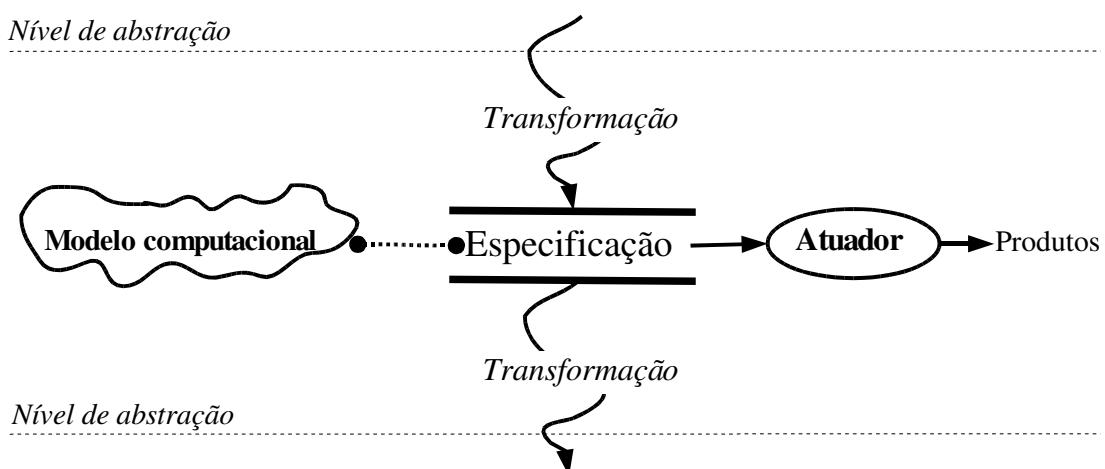


Figura 11 Um nível de abstração.

Uma especificação é um conjunto de informações sobre um sistema sendo projetado. Um nível de abstração determina que conjunto de informações sobre o sistema deve estar presente na especificação. Uma especificação pode estar associada a um ou mais modelos computacionais. Quando isto ocorre, ela é denominada de especificação executável. Uma especificação executável é capaz de capturar e reproduzir o comportamento funcional do sistema. Transformações podem ser efetuadas sobre uma especificação, produzindo como resultado outra especificação, no mesmo ou em outro nível de abstração. Especificação inicial é aquela que não é fruto de uma transformação. Uma função denominada de atuador pode ser efetuada

sobre uma especificação, produzindo como resultado informações que não constituem uma especificação do sistema. Por exemplo, um aturador pode ser uma ferramenta que analisa características de construção da especificação, e produz um conjunto de mensagens.

Uma metodologia de projeto contém um conjunto de níveis de abstração e uma relação parcial entre eles. Para todo nível de abstração, a relação indica quais são os possíveis níveis sucessores.

A implementação do sistema em algum conjunto de tecnologias, que denomino de arquitetura alvo, é um nível de abstração especial, pois não possui sucessores. Neste nível, o modelo computacional associado à especificação corresponde à própria arquitetura alvo. Qualquer outro nível de abstração da metodologia determina um número menor de informações sobre uma especificação que o nível de implementação. Desta forma, o objetivo de uma metodologia de projeto é a partir de uma especificação inicial, obter uma especificação para a arquitetura alvo.

Dentre inúmeras arquiteturas alvo que poderiam ser utilizadas em uma metodologia para projeto de software embarcado, escolhi uma composta de uma unidade de processamento baseada em microprocessador. Esta arquitetura é suficientemente completa para a implementação de software embarcado. Embora outras arquiteturas são possíveis, eventualmente até mais adequadas, o objetivo deste trabalho não está voltado na obtenção de implementações eficientes, mas sim no estudo do emprego de modelos computacionais. A Figura 12 ilustra seus principais componentes.

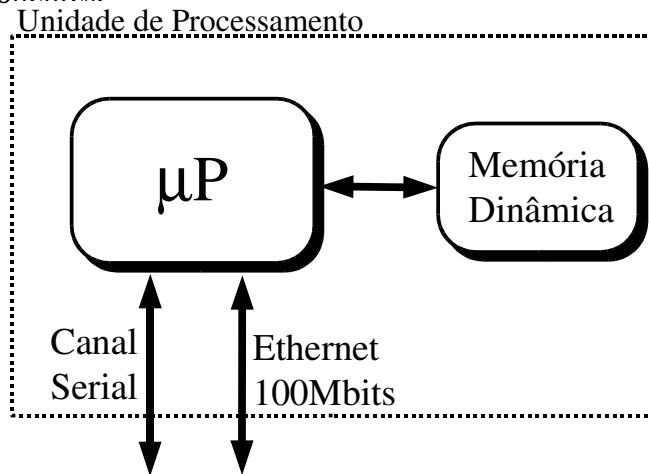


Figura 12 Arquitetura alvo da metodologia.

A arquitetura da Figura 12 é composta por um processador, um banco de memória dinâmica, um canal de comunicação serial e um canal de comunicação *ethernet*. O processador AMD Athlon foi escolhido por razões de disponibilidade de equipamento. A versão desse processador é aquela encontrada em computadores de uso genérico. Dessa forma, este processador não é o mais adequado para projeto de software embarcado, pois seu projeto é voltado para a execução de software genérico em computadores de uso geral²⁹. Entretanto, a arquitetura alvo que utilizei permite explorar satisfatoriamente o uso de diferentes modelos computacionais para desenvolvimento de software embarcado.

Além da memória *cache* disponível internamente no circuito do processador, a arquitetura alvo possui um banco de memória dinâmica, utilizado para armazenar instruções e dados. A comunicação com o meio externo se dá através de dois canais bidirecionais. O primeiro é uma conexão serial UART. O segundo é uma rede *ethernet* de 100Mbits/s. O canal serial é mais adequado para enviar e receber dados de controle, uma vez que a taxa de transmissão é muito limitada. O canal *ethernet* é utilizado para fazer a descarga do programa a ser executado na arquitetura e para receber e enviar dados para a aplicação sendo executada.

Uma vez determinada a arquitetura alvo, caracterizando-se assim o nível de implementação, deve-se especificar os demais níveis de abstração da metodologia. A Figura 13 ilustra a metodologia que utilizei nesta tese para projeto de software embarcado. Ela é composta por quatro níveis de abstração: nível de concepção; nível de interação de atores; nível de comunicação de processos; nível de implementação.

²⁹ Em <http://www.ece.utexas.edu/~bevans/courses/ee382c/lectures/processors.html>, há uma comparação entre um processador específico para aplicações embarcadas (TMS320C62X) e o processador Pentium.

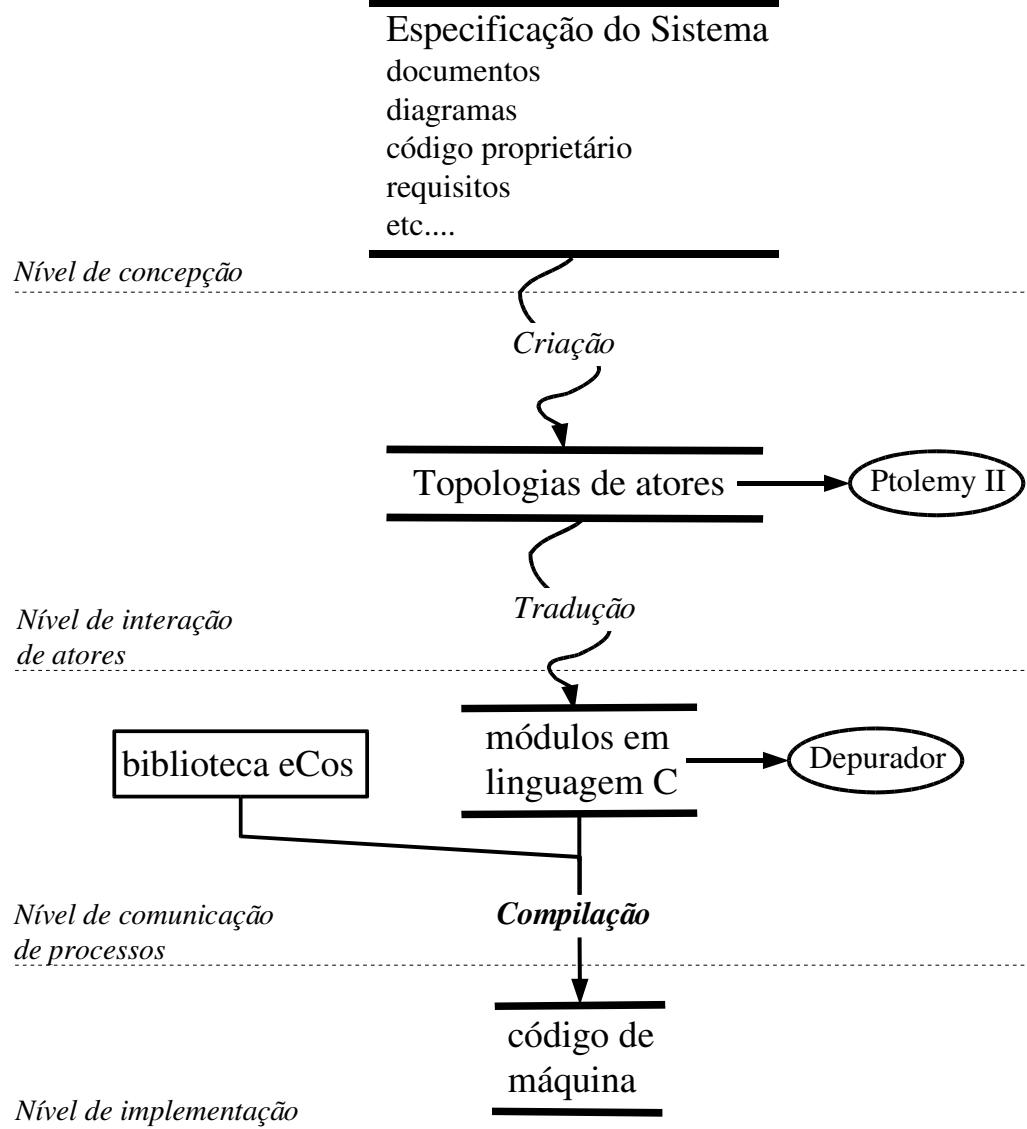


Figura 13 Metodologia para projeto de software embarcado.

O nível de concepção é composto por uma ou mais especificações iniciais para o sistema. Algumas são executáveis, como código fonte de versões anteriores da implementação do sistema ou códigos reutilizáveis de bibliotecas. Outras não são executáveis, como documentos de requisitos, restrições de implementação e descrições de funcionalidade.

No capítulo 5, descrevo dois estudos de caso utilizando a metodologia da Figura 13. Em um deles, apenas uma especificação inicial não executável foi utilizada. No outro, utilizei um documento descrevendo a funcionalidade e restrições do sistema conjuntamente com um código fonte implementando os blocos descritos

na documentação.

Uma especificação no nível de interação de atores captura parcialmente ou totalmente as especificações iniciais em topologias compostas de atores que interagem entre si. A criação de uma especificação baseada em atores é feita integralmente pelo desenvolvedor. A Figura 13 ilustra um atuador deste nível de abstração: o ambiente Ptolemy II. Através deste atuador, é possível executar uma especificação de atores, observando o comportamento do sistema para diferentes situações de teste. Isso permite determinar se a especificação está corretamente construída para os casos testados. Observações também podem ser efetuadas após a execução de uma especificação, determinando modificações a serem feitas nela. Outros atuadores são possíveis para esse nível de abstração. Em [JEU00] descrevi uma ferramenta que coleta informações sobre a execução da especificação, permitindo observar a evolução da execução e analisar a comunicação de dados entre atores. No Capítulo 4, apresento um novo e importante atuador operando sobre especificações contendo atores.

O nível de comunicação de processos é caracterizado por dividir o sistema em processos com fluxos de execução independentes. Embora exista um espaço global de variáveis que podem ser lidas e modificadas por todos os processos, cada processo também possui seu espaço local de variáveis. Para a arquitetura alvo que utilizo, todos os processos executam no mesmo processador. Neste nível de abstração, um sistema operacional (biblioteca eCos) é utilizado para permitir a criação e comunicação dos diferentes processos, além de fornecer serviços de acesso a arquitetura alvo. Como cada processo constitui um fluxo de execução autônomo, o sistema operacional também fornece primitivas para a sincronização de processos. Um exemplo de atuador neste nível de abstração é uma ferramenta de depuração, que permite executar passo a passo o código fonte produzido.

Uma especificação no nível de comunicação de processos pode ser traduzida para uma especificação no nível de implementação através de um processo de compilação. Este tipo de tradução é bem descrita na literatura [APP98]. Em vários casos, também é possível traduzir uma especificação baseada na interação de atores para uma no nível de comunicação de processos. Este é o tema que abordo neste capítulo. Na seção 3.3, descrevo os passos necessários na tradução de uma especificação baseada em atores, focando quatro modelos computacionais. Antes

descrevo em maior detalhe o nível de comunicação de processos, a fim de tornar claro o alvo da tradução de uma especificação baseada em atores.

É importante notar que meu intuito não é obter uma metodologia de projeto avançada para sistemas embarcados, isto é, que conte cole diferentes arquiteturas com várias fases e ferramentas de projeto. Meu interesse é introduzir um ambiente baseado na interação de atores em uma metodologia de projeto, a fim de ser possível de caracterizar as diferentes implementações obtidas a partir de especificações utilizando atores.

3.2 Nível de Comunicação de Processos

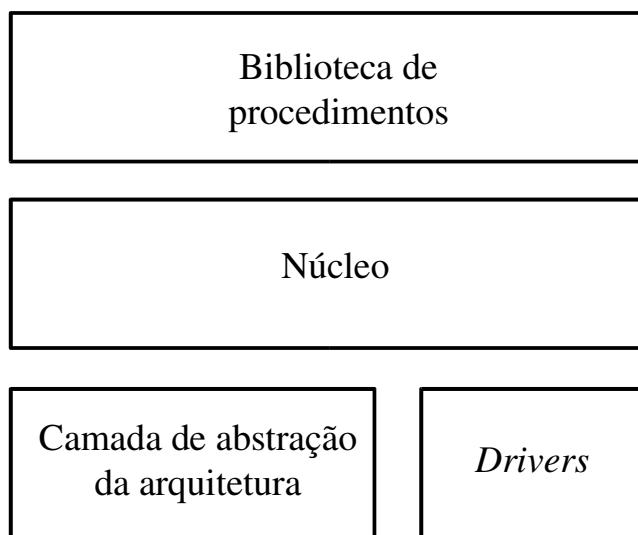
3.2.1 O sistema operacional eCos

Uma especificação de um sistema poderia ser escrita no nível de implementação através da linguagem montadora do processador da arquitetura alvo. Entretanto, tal solução deve ser evitada devido a fatores como propensão a erros, dificuldade de compreensão e reutilização do código. A alternativa utilizada para projeto de software embarcado é o emprego de uma linguagem estruturada, como a linguagem de programação C [KER88]. Para o desenvolvimento de software embarcado, é também empregado um sistema operacional embarcado [LI03].

A finalidade de um sistema operacional embarcado é similar a de um sistema operacional genérico. Entre outras, temos: (1) facilitar o acesso aos serviços da arquitetura alvo, através de dispositivos denominados de *drivers*; (2) prover um modelo de programação. Entretanto, devido às restrições físicas de um sistema embarcado e arquiteturas específicas desenvolvidas para tais sistemas, sistemas operacionais embarcados são projetados com características específicas, como minimização do tamanho do código do sistema, resposta eficiente a interrupções, uso eficiente do processamento e portabilidade. Dentre os vários sistemas operacionais embarcados disponíveis, utilizei neste trabalho o sistema eCos (*Embedded Configurable Operational System*) [MAS03], da companhia RedHat. Trata-se de um sistema operacional de domínio público, cuja principal característica é sua configurabilidade.

O sistema eCos é composto por 14 pacotes que implementam diferentes aspectos do sistema. Existem mais de 200 opções de configuração para todos esses pacotes. O desenvolvedor pode selecionar a configuração que lhe é mais adequada para uma dada implementação, tornando assim o sistema mais eficiente. Por exemplo, não utilizei o sistema de arquivos ou o servidor do protocolo http disponíveis como pacotes no sistema eCos. A Figura 14 apresenta um diagrama simplificado dos módulos do sistema eCos.

Especificação do Sistema



Arquitetura Alvo

Figura 14 Diagrama simplificado do sistema eCos [MAS03].

A camada de abstração da arquitetura e os *drivers* permitem o acesso aos serviços da arquitetura alvo. Para cada arquitetura alvo disponível pelo sistema eCos, diferentes implementações destes dois blocos são necessários. Entretanto, estas implementações estão embutidas no sistema eCos, sendo invisíveis ao desenvolvedor do sistema.

O núcleo do sistema suporta o modelo de programação baseado em processos comunicantes. Seu principal componente é um escalonador de processos, responsável por escolher um processo a ser executado pelo processador em um dado instante. Duas configurações importantes que o desenvolvedor pode efetuar no

núcleo é o tipo de algoritmo de escalonamento (existem 3 distintos) e configurar a fração de tempo máxima que um processo pode executar no processador. O núcleo do sistema também implementa um conjunto de primitivas para comunicação e sincronismo dos processos.

A biblioteca de procedimentos contém funções utilizadas na programação de software, como conjuntos de operações matemáticas e operações sobre cadeias de caracteres. Embora não seja mostrado na Figura 14, outros blocos podem ser adicionados ao sistema através de configurações. Um exemplo de pacote adicional ao sistema básico que utilizei é o suporte ao protocolo TCP/IP.

3.2.2 O modelo de programação

O núcleo do sistema eCos provê ao desenvolvedor dois componentes importantes na implementação de software embarcado: processos e métodos de sincronização. Um processo é um fluxo autônomo de execução, com uma pilha própria para armazenar variáveis locais e informações sobre o estado de sua execução. Múltiplos processos compartilham um mesmo segmento de memória. O núcleo do sistema eCos contém uma interface de programação composta por métodos para criar, manipular e destruir processos. Dentre eles, os principais são:

- *cyg_thread_create()*: adiciona um processo a lista de processos da aplicação. O estado do processo, após a execução deste procedimento, é suspenso³⁰. Deve ser especificado como parâmetro do método um valor de prioridade, uma região de memória para a pilha do processo, o procedimento a ser executado pelo processo, entre outros;
- *cyg_thread_delay()*: muda o estado do processo para suspenso, por uma duração especificada por um parâmetro. A duração é medida em eventos do relógio. A quantidade de eventos por segundo é definida como uma configuração do núcleo do sistema eCos. Após o término da duração especificada, o estado do processo passa a ser pronto;
- *cyg_thread_yield()*: altera o estado do processo para pronto, permitindo

³⁰ Os estados [TAN01] de um processo no sistema eCos são: executando, pronto e suspenso.

que outro processo seja executado;

- *cyg_thread_kill()*: encerra a execução de um processo;
- *cyg_thread_measure_stack_usage()*: determina o tamanho máximo utilizado na pilha do respectivo processo, até o momento da chamada deste método;
- *cyg_thread_suspend()*: muda o estado do processo para suspenso. O processo retorna ao estado pronto somente após a chamada do procedimento *cyg_thread_resume()*;
- *cyg_thread_resume()*: muda o estado de um processo para pronto.

O valor de prioridade associado a um processo é utilizado pelo escalonador no momento de decidir, dentre todos os processos que estejam em um estado pronto para a execução, qual deve ser o próximo processo a ser executado. Este valor costuma ser utilizado pelos desenvolvedores de software embarcado a fim de garantir restrições de tempos de execuções. Isto é, quanto menor o tempo de resposta exigido sobre um dado processo, maior deve ser sua prioridade.

A Figura 15 exemplifica o uso de processos por aplicações.

```

1- externC void cyg_start( void ) {
2-     cyg_thread_create(4, simple_program, (cyg_addrword_t) 0,
3-         "Thread A", (void *) stack[0], 4096,
4-         &simple_threadA, &thread_s[0]);
5-     cyg_thread_resume(simple_threadA);
6-
7-     cyg_thread_create(4, simple_program, (cyg_addrword_t) 1,
8-         "Thread B", (void *) stack[1], 4096,
9-         &simple_threadB, &thread_s[1]);
10-    cyg_thread_resume(simple_threadB);
11-
12-    cyg_thread_create(3, estorvo, (cyg_addrword_t) 2,
13-        "Estorva o processador", (void *) stack[2], 4096,
14-        &estorvoThread, &thread_s[2]);
15-    cyg_thread_resume(estorvoThread);
16-
17-    Cyg_Scheduler::scheduler.start();
18- }
19-
20- void estorvo(cyg_addrword_t data) {
21-     int r, r2, count = 0;
22-
23-     for(;;) {
24-         r = rand();
25-         r2 = rand();
26-         acc = r + r2;
27-         count++;
28-         if(count == 100000) {
29-             cyg_thread_delay(1);
30-             count = 0;
31-         }
32-     }
33- }
34-
35- void simple_program(cyg_addrword_t data) {
36-     int message = (int) data;
37-     int delay;
38-     for(;;) {
39-         delay = 200 + (rand() % 50);
40-         diag_printf("Thread %d: delay %d\n", message, delay);
41-         cyg_thread_delay(delay);
42-     }
43- }

```

Figura 15Exemplo de uso de processos.

O exemplo da Figura 15 é composto por três processos: dois executam independentemente o procedimento *simple_program()* (linhas 35 a 42) e outro executa o procedimento *estorvo()* (linhas 20 a 32). Os processos são criados nas linhas 2 a 4, 7 a 9 e 12 a 14. Após a criação de cada processo, o procedimento *cyg_thread_resume()* é utilizado para mudar o estado de cada processo para pronto (linhas 5, 10 e 15). Note que os procedimentos que são executados pelos processos,

possuem laços infinitos (linha 23 e 38). Toda vez que um procedimento alcança sua instrução de retorno, o respectivo processo associado é encerrado. Desta forma, o exemplo da Figura 15 foi desenvolvido para nunca terminar sua execução.

O sistema eCos possui seis tipos de primitivas de sincronismo. Dentre elas, utilizei as primitivas *mutex*, semáforo e *flag*.

Um *mutex* é uma primitiva utilizada para serializar o acesso a algum recurso compartilhado por mais de um processo, como por exemplo uma região de memória. Um *mutex* possui dois estados: bloqueado e livre³¹. Um processo pode requisitar a mudança do estado de um *mutex* de livre para bloqueado. Um processo que efetue esta transição de estado é dito o proprietário do *mutex*. Um *mutex* só pode possuir um proprietário em um dado momento. Desta forma, algum processo que tente se tornar o proprietário de um *mutex* no estado bloqueado tem sua execução interrompida. Ela é reestabelecida somente quando o proprietário do *mutex* modificar seu estado para livre. Os principais procedimentos do sistema eCos para manipulação de *mutex* são:

- *cyg_mutex_init()*: inicializa o *mutex*;
- *cyg_mutex_destroy()*: elimina da memória o *mutex*. Não é recomendado executar este procedimento quando o estado do *mutex* for bloqueado;
- *cyg_mutex_lock()*: se o estado do *mutex* for livre, modifica-o para bloqueado e associa como proprietário o processo que chamou o procedimento. Se o estado for bloqueado, interrompe a execução do processo que chamou o procedimento.
- *cyg_mutex_trylock()*: tenta modificar o estado do *mutex* de livre para bloqueado. Se o *mutex* estiver no estado bloqueado, o procedimento retorna imediatamente;
- *cyg_mutex_unlock()*: modifica o estado do *mutex* de bloqueado para livre.

Um semáforo é uma primitiva que possui um contador indicando se um dado recurso está bloqueado ou livre. Um processo pode incrementar ou decrementar o valor do contador de um semáforo. A tentativa de decrementar um contador cujo

³¹ O estado inicial de qualquer *mutex* é livre.

valor é zero resulta na suspensão da execução do processo que tentou o decremento. A execução de processos suspensos devido a um contador com valor igual a zero é reestabelecida somente quando o contador for incrementado. Um semáforo pode ser binário, isto é, o valor máximo assumido pelo contador é igual a 1. Um semáforo binário não é igual a um *mutex*, pois um semáforo não possui a noção de proprietário. Apenas um processo pode modificar o estado de um *mutex* de bloqueado para livre, o que não acontece com um semáforo. Os principais procedimentos para manuseio de semáforos são:

- *cyg_semaphore_init()*: inicializa um semáforo especificando o valor máximo de seu contador;
- *cyg_semaphore_destroy()*: elimina da memória um semáforo. Não é recomendado executar este procedimento quando houverem processos suspensos devido ao semáforo;
- *cyg_semaphore_wait()*: decrementa o valor do contador do semáforo;
- *cyg_semaphore_post()*: incrementa o valor do contador do semáforo.

Uma *flag* é uma palavra de 32 bits, onde cada bit representa uma condição. Um processo pode ter sua execução interrompida até que uma ou mais condições da *flag* sejam satisfeitas. Uma condição é satisfeita quando o respectivo bit da *flag* possui valor lógico verdade. Um processo pode alterar o valor de um bit da *flag*, indicando assim o estado de alguma condição. Os principais procedimentos para manuseio de *flags* são:

- *cyg_flag_init()*: inicializa uma *flag*;
- *cyg_flag_destroy()*: elimina da memória uma *flag*, desde que nenhum processo esteja suspenso esperando alguma condição da *flag*;
- *cyg_flag_setbits()*: atribui o valor verdade a bits da *flag*, especificados por um parâmetro do procedimento. Processos que estejam suspensos esperando condições associadas a estes bits têm seus estados alterados para pronto;
- *cyg_flag_maskbits()*: atribui o valor falso a bits da *flag*, especificados por

- um parâmetro do procedimento;
- *cyg_flag_wait()*: especifica um conjunto de condições que devem ser satisfeitas para prosseguir com a execução do processo.

3.2.3 Caracterização de software embarcado

Uma especificação no nível de comunicação de processos é traduzida para o nível de implementação através de um processo de compilação. Após este processo, além de determinar se a funcionalidade do sistema está corretamente capturada pela implementação, é necessário caracterizar a qualidade da implementação obtida. Esta caracterização permite determinar se as restrições de desempenho foram cumpridas e fornece uma maneira de comparar a eficiência de diferentes implementações para o mesmo sistema. Dentre os possíveis parâmetros de caracterização, inclui os seguintes na metodologia de projeto:

1. Tempo de resposta a um conjunto de estímulos;
2. Taxa de processamento de dados de entrada;
3. Quantidade de memória necessária para armazenar as instruções da implementação;
4. Quantidade de memória utilizada para armazenar os dados;
5. Taxa de ocupação do processador.

Os dois primeiros itens envolvem medir quanto tempo dura a execução de um dado trecho de código. A passagem do tempo no sistema eCos é baseada em ciclos do relógio principal. O período de um ciclo é especificado como um parâmetro de configuração no núcleo do sistema. O valor padrão para o período do relógio principal é de 10 ms, valor especificado para a arquitetura que utilizei. O sistema eCos provê em seu núcleo um procedimento (*cyg_current_time()*) que retorna quantos ciclos de relógio transcorreram desde o início da execução. Desta forma, o valor retornado por esse procedimento é um número inteiro. A fim de obter um valor mais preciso, o ambiente eCos possui uma macro (*HAL_CLOCK_READ*) que retorna quanto tempo transcorreu desde a última interrupção provocada por um novo ciclo

de relógio. Combinando a macro e o procedimento, é possível medir com precisão de micro-segundos a duração de algum trecho de código. Em particular, o ambiente fornece um exemplo de aplicação que mede para diferentes procedimentos do sistema, tais como criação e destruição de processos, manipulação de primitivas de sincronismo, sua duração média. O apêndice A apresenta uma tabela resumida dos valores obtidos para a arquitetura alvo que utilizei.

A aferição da quantidade de memória necessária para armazenar as instruções de uma implementação é feita através do aplicativo *size*, que está incluso no conjunto de ferramentas do processo de compilação. Esta ferramenta discrimina a quantidade de bytes alocados para as instruções e variáveis estáticas. Desta forma, o aplicativo *size* também é utilizado no cálculo da quantidade de memória utilizada para armazenar as variáveis da implementação. Em particular, as pilhas de cada processo estão incluídas neste grupo.

Além do espaço utilizado por variáveis estáticas, seria interessante determinar a quantidade de memória associada às variáveis alocadas dinamicamente. Entretanto, não implementei este tipo de análise. Para tal, é necessário estudar e alterar o pacote de manutenção de memória dinâmica presente no sistema eCos.

O último parâmetro de caracterização de uma implementação é a taxa de ocupação do processador. Este parâmetro indica o percentual de tempo em que o processador esteve efetuando cálculos. No sistema eCos, quando nenhum cálculo deve ser realizado, ou não é possível de ser realizado, um processo especial, denominado de *Idle*³², é executado. Este processo não apresenta nenhuma função. Desta forma, para calcular a taxa de ocupação do processador, é necessário calcular quanto tempo o processo *Idle* ocupou o processador.

A informação de quanto tempo um processo ocupou o processador é útil para todos os processos, não somente o processo *Idle*. Isto pode ser calculado conhecendo-se os instantes em que o processo esteve executando. Além de fornecer a ocupação do processador por um processo, estes dados permitem analisar o comportamento do escalonamento dos processos.

A captura e processamento dos instantes de execução de um processo não

32 Palavra da língua inglesa que significa ocioso.

estão disponíveis em nenhuma ferramenta fornecida conjuntamente com o sistema eCos. Desta forma, implementei um aplicativo para esta função. O ambiente eCos possui um pacote de instrumentação que auxilia na captura de eventos produzidos durante a execução. Por exemplo, toda vez que uma primitiva de sincronização é criada, é possível armazenar tal evento em uma variável do tipo vetor, incluindo informações como o momento da criação e qual processo chamou tal procedimento. Utilizei a funcionalidade do pacote de instrumentação para armazenar os momentos onde houve a mudança de estado de um processo e a ocorrência de um novo ciclo de relógio. Uma vez que o vetor de eventos é preenchido, este é enviado para um computador conectado a arquitetura alvo, através do canal *ethernet*. Desenvolvi um aplicativo que processa os dados e gera um relatório contendo a evolução dos estados para cada tarefa, bem como uma lista ordenada no tempo contendo qual tarefa estava sendo executada em um dado momento. A taxa de ocupação do processador também é fornecida. O apêndice B contém o relatório produzido para o exemplo da Figura 15.

3.3 Nível de Interação de atores

Uma especificação neste nível de abstração utiliza as classes do ambiente Ptolemy II a fim de fornecer uma sintaxe e semântica para a especificação, além de associar modelos computacionais a topologias. O processo de tradução deve eliminar totalmente o uso destas classes, implementando o sistema como um conjunto de processos no ambiente eCos. Além disto, é necessário traduzir o código escrito em linguagem Java para a linguagem C.

Dos seis modelos computacionais que utilizei neste trabalho, e descritos no capítulo 2, implementei um processo de tradução não-automatizada para os modelos SDF, SR, PN e CSP. Não contelei o modelo CT pois este é específico para simulação, isto é, ele é utilizado para modelar um (sub)sistema cuja implementação não é um software. Embora seja possível traduzir uma topologia associada ao modelo DE, não considerei interessante pois as características do modelo DE são orientadas à simulação de sistemas discretos. Por exemplo, não há uma tradução clara do conceito de *timestamp*. Embora não adequados para a implementação de

software, os modelos CT e DE são muito úteis para a simulação de sistemas.

3.3.1 Etapas da Transformação

Sete tarefas formam o processo de tradução de uma especificação baseada na interação de atores para uma composta por processos comunicantes:

1. Tradução dos tipos de dados: para todo tipo do ambiente Ptolemy II é necessário encontrar um tipo equivalente na linguagem C;
2. Tradução dos métodos de comunicação de dados: a troca de dados entre atores é efetuada através de métodos da classe que implementa o conceito de porto de E/S. Desta forma, estes métodos devem apresentar uma tradução na implementação da especificação baseada em processos comunicantes;
3. Implementação de parâmetros de um ator: a um ator podem estar associado objetos denominados de atributos. Para alguns atributos é possível associar valores resultantes do cálculo de uma expressão. Este tipo de atributo, denominado de parâmetro, é muito útil na implementação da função do ator. Desta forma, é necessário que parâmetros sejam traduzidos;
4. Implementação de tokens³³ produzidos ou consumidos: atores interagem entre si trocando *tokens*, que são objetos que encapsulam um dado. Os *tokens* devem ser convertidos em variáveis;
5. Implementação dos métodos de execução de um ator atômico: a função de um ator atômico é dividida em cinco métodos de execução. Tais métodos devem ser traduzidos para procedimentos, sendo chamados em algum momento da execução;
6. Associar atores atômicos a processos: em uma especificação no nível de processos comunicantes, a entidade que realiza uma função é o procedimento associado ao processo. É necessário associar a execução de um ator atômico a algum processo;

33 Um objeto que encapsula um valor no ambiente Ptolemy II. Vide Anexo A.

7. Implementação dos métodos de execução de um ator hierárquico opaco: parte da implementação de um modelo computacional está dividida nos cinco métodos de execução que controlam uma topologia. O processo de tradução deve encontrar um mapeamento para estes métodos.

As tarefas 1, 3 e 4 independem de um modelo computacional. As tarefas 2, 5, 6 e 7 são específicas para cada modelo computacional. A ordem de execução destas quatro tarefas também depende do modelo computacional em questão.

Além dos métodos de comunicação, um ator atômico também pode manipular uma série de outros métodos associados a portos de E/S e a classe base que define o conceito de ator. Alguns destes métodos não possuem traduções, pois são pertinentes apenas à sintaxe ou semântica abstrata do ambiente Ptolemy II. Por exemplo, o método `getReceivers()` de um porto de entrada retorna todos os objetos do tipo receptor associados a ele. Este método não apresenta correspondente em uma especificação de processos comunicantes, pois a noção de receptor foi descartada. Para ser possível a tradução de um ator, todas as características utilizadas e dependentes do ambiente Ptolemy II devem apresentar alguma forma de tradução.

3.3.2 Transformações independentes de um modelo computacional

Tarefa 1 – Tradução dos tipos de dados

O ambiente Ptolemy II permite associar um tipo aos dados produzidos e/ou consumidos por um porto[XIO02]. Para todo tipo presente no ambiente Ptolemy II, é necessário encontrar seu par na linguagem C, uma vez que esta é a linguagem utilizada na construção de uma especificação baseada em processos. Temos então os seguintes tipos com suas respectivas traduções:

Tabela 1 – Tradução dos tipos de dados.

<i>Tipo do ambiente Ptolemy II</i>	<i>Tipo da Linguagem C</i>
<i>UnsignedByte</i>	<i>char</i>

<i>Tipo do ambiente Ptolemy II</i>	<i>Tipo da Linguagem C</i>
<i>Boolean</i>	<i>char</i> , onde 0 corresponde ao valor false e 1 ao valor true.
<i>Int</i>	<i>int</i>
<i>Long</i>	<i>long</i>
<i>Fix</i>	<i>fix_t</i> (biblioteca)
<i>Double</i>	<i>double</i>
<i>Complex</i>	<i>complex_t</i> (biblioteca)
<i>BooleanMatrix</i>	<i>char [][]</i>
<i>IntMatrix</i>	<i>int [][]</i>
<i>DoubleMatrix</i>	<i>double [][]</i>
<i>ComplexMatrix</i>	<i>complex_t [][]</i>
<i>Object</i>	<i>struct *</i>
<i>String</i>	<i>char *</i>
<i>Record</i>	<i>struct</i>

Tanto o tipo *Complex* (número complexo) quanto o tipo *Fix* (tipo real com precisão limitada) não estão presentes na definição da linguagem C. Embora não tenha necessitado utilizar nenhum destes dois tipos, ambos podem ser adicionados a linguagem C através de bibliotecas. O tipo *Complex* pode ser adicionado utilizando um estrutura com duas variáveis do tipo *double*. Implementações para o tipo *Fix* estão disponíveis em bibliotecas de domínio público³⁴.

Tarefa 3 – Implementação de parâmetros

A um ator atômico e hierárquico podem estar associados a parâmetros. Todo parâmetro possui um valor padrão, sendo possível alterá-lo antes da execução da especificação. Modificar o valor de um parâmetro durante a execução não deve ser efetuado de maneira indiscriminada, pois pode resultar em especificações com execução imprevisível. Em [NEU04], Neuendorffer estuda quando e de que forma um parâmetro pode ser alterado em especificações associados a modelos de fluxo de dados, sem provocar um comportamento errôneo da especificação.

34 Um exemplo é a biblioteca SystemC (www.systemc.org).

Um parâmetro pode ser facilmente implementado através de uma variável compartilhada. Entretanto, seu valor só pode ser alterado antes do início da execução, isto é, apenas o procedimento de iniciação do ator pode atribuir um valor à variável.

Tarefa 4 – Implementação de *tokens*

Todo *token* deve ser transformado em uma variável local ao escopo onde é utilizado. O tipo da variável deve ser compatível ao respectivo *token*, conforme listado na Tabela 1.

3.3.3 Transformação a partir do modelo SDF

A execução de um ator de fluxo de dados é dividida em uma seqüência, possivelmente infinita, de cálculos de sua função. Um escalonamento determina, para um dado instante, qual deve ser o ator a ter sua função computada. Para o modelo SDF, este escalonamento é periódico e calculado antes da execução da topologia. Desta forma, associo a uma topologia SDF um processo (tarefa 6). O processo permanece suspenso até que um novo conjunto de dados satisfaça as taxas de amostragem dos portos de entrada da topologia. Cada método de execução de um ator atômico é associado a um procedimento (tarefa 5). Um ativamento de um ator corresponde a chamar o procedimento que implementa o método *prefire()*. Caso um valor booleano verdade seja retornado, os procedimentos implementando os métodos *fire()* e *postfire()* são chamados, respectivamente.

No capítulo 2, descrevi um algoritmo para encontrar um escalonamento para uma topologia SDF. Ele se baseia na simulação da execução a fim de encontrar um escalonamento válido, e não possui nenhum objetivo de otimização. Mais de um escalonamento para uma dada topologia SDF podem ser obtidos, dependendo da implementação do algoritmo. Embora todos os escalonamentos implementem corretamente uma iteração da topologia, alguns podem ter um custo menor que outros. A Figura 16 ilustra um exemplo de tal situação.

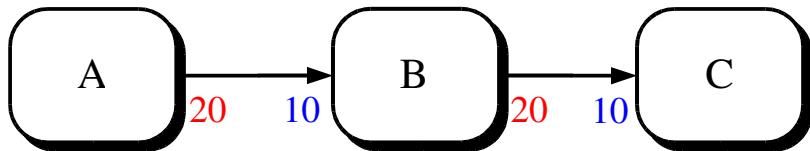


Figura 16Uma topologia SDF.

As taxas de amostragem da topologia da Figura 16 implicam que para cada ativação do ator A, o ator B deve ser ativado duas vezes e o ator C quatro vezes. Escalonamentos válidos para a topologia da Figura 16 são:

- ABCBCCC
- A(2 B(2 C))
- A(2 B)(4 C)
- A(2BC)(2C)

A utilização de parênteses na notação dos escalonamentos ilustra a ocorrência de repetições de ativamentos, isto é, (n XY) significa ativar XY n vezes. Assim, o escalonamento A(2 B(2 C)) = ABCCBCC. Um escalonamento escrito na notação de parênteses também é chamado de escalonamento iterado (*looped schedule*). Cada ocorrência de um ator em um escalonamento é também chamado de aparição do ator. Um escalonamento iterado onde a aparição de cada ator é unitária é denominado de escalonamento iterado de aparição única [BHA94].

A vantagem de um escalonamento iterado é a facilidade de se determinar os laços de repetições de atores. Por exemplo, o pseudo-código para o escalonamento A (2 B(2 C)) seria:

```

  ative o ator A
  para i = 0 até 2 faça
    ative o ator B
    para j = 0 até 2 faça
      ative o ator C
  
```

Figura 17Pseudo-código para A(2 B(2 C)).

Utilizando uma transformação que substitui a chamada de um procedimento

pelo seu código fonte e desconsiderando-se o código necessário para os laços de iteração, a quantidade de memória necessária para armazenar as instruções associadas a um processo é proporcional ao número de aparições de atores no escalonamento. Desta forma, um escalonamento de aparição única é o mais eficiente em termos do número de instruções.

Além de determinar a quantidade de instruções associadas a um processo, um escalonamento também influencia a quantidade de memória necessária para armazenar os dados comunicados entre atores. Por exemplo, os escalonamentos para o exemplo da Figura 16 necessitam de 50, 40, 60 e 50 posições de memória³⁵, respectivamente. Em [BHA94], Bhattacharyya desenvolve uma técnica para calcular escalonamentos iterados eficientes, cujo primeiro objetivo de otimização é minimizar a quantidade de instruções e o segundo é minimizar a quantidade de memória necessária para armazenar dados. Apliquei os algoritmos desenvolvidos por Bhattacharyya para escalonar as topologias SDF neste trabalho.

Após o escalonamento, é necessário associar a cada fila de comunicação uma região de memória utilizada para armazenar os dados produzidos por um ator. Uma alternativa é utilizar um vetor indexado. Existem duas possibilidades de implementação da indexação: estática ou dinâmica. Um vetor indexado estaticamente é aquele em que o n -ésimo dado produzido por um ator e consumido por outro reside no mesmo endereço de memória, independentemente do número de iterações já efetuadas do escalonamento. Um vetor indexado dinamicamente é aquele que não é indexado estaticamente. Um vetor indexado estaticamente pode ser implementado por uma região contínua de memória e uma variável inteira indicando o endereço do próximo dado. Um vetor indexado dinamicamente pode ser implementado com uma região contínua de memória e duas variáveis: uma indicando o endereço de escrita do próximo dado e outra indicando o endereço do próximo dado a ser lido. A cada operação de leitura ou escrita, o respectivo índice é atualizado adicionando-se um a seu valor e tomando o módulo com relação ao tamanho total do vetor.

O tamanho de qualquer vetor associado a uma fila de comunicação deve ser maior ou igual ao número máximo de dados que podem existir na fila durante uma iteração. Utilizando as taxas de amostragem de cada ator e o escalonamento

³⁵ A quantidade exata depende do tipo associado a cada dado.

associado a topologia, é possível determinar quantos dados irão existir ao máximo em uma dada conexão entre atores durante uma iteração. Em [BHA94], Bhattacharyya estabelece as condições necessárias e suficientes para a utilização de indexamento estático:

1. Quando não houver atraso na aresta entre o par de atores;
2. Há atraso na aresta entre o par de atores. Neste caso o indexamento estático é possível se o número total de dados consumido pelo ator destino for diretamente proporcional ao tamanho do vetor, para alguma constante inteira maior ou igual a 1.

O processo de alocação de memória, efetuado após o escalonamento da topologia, é composto pelas seguintes etapas:

1. Para cada conjunto de atores relacionados, determinar o número máximo de dados vivos durante uma iteração. Um dado é dito vivo em um determinado momento da iteração quando ele foi produzido mas ainda não consumido;
2. Para cada conjunto de atores relacionados sem haver atraso entre eles, atribuir um vetor indexado estaticamente;
3. Para cada par de atores relacionados com atraso, determinar o tamanho necessário para utilizar indexamento estático e comparar com o tamanho necessário com indexamento dinâmico;
4. Para os vetores indexados estaticamente, percorrer o escalonamento a fim de determinar a seqüência de valores dos índices.

O primeiro passo consiste em simular a execução da topologia mantendo-se um registro de quantos dados vivos existem em cada fila, durante uma iteração. No início da iteração, apenas os pares de atores relacionados com atraso apresentam dados vivos. Para cada aparição de ator no escalonamento, a quantidade de dados vivos nas filas onde o ator consome e produz é atualizada com base nas taxas de amostragem do ator.

O quarto passo é similar ao primeiro em execução, mas ao invés de determinar a quantidade de dados vivos para um vetor, é mantido um registro da variação do valor do índice associado ao vetor.

Um dado produzido por um ator pode ser consumido por mais de um ator. Neste caso, há duas opções: (1) considerar o dado vivo enquanto houver pelo menos um ator que não tenha consumido o dado ou (2) criar uma cópia do dado para cada ator destino. A segunda opção é redundante, além de custosa, pois a quantidade de dados envolvida pode não ser desprezível. Utilizei então a primeira abordagem. Desta forma, tanto a etapa 1 quanto a etapa 4 do processo de alocação levam em conta múltiplos consumidores na definição de dado vivo. Caso exista atraso em algum par de atores de uma relação contendo múltiplos consumidores, então um vetor indexado dinamicamente é usado especificamente para cada conexão com atraso.

A partir do resultado da alocação de memória, é possível substituir os métodos de produção (*put()* e *broadcast()*) e consumo (*get()*) de dados por atribuições e referências aos respectivos vetores indexados (tarefa 2). No modelo SDF, os métodos *isKnown()*, *hasToken()* e *hasRoom()* sempre retornam verdade. Os métodos *broadcastClear()* e *sendClear()* não possuem significado, e devem ser eliminados. Uma vez substituído todos os métodos de comunicação, os vetores utilizados pelo ator são passados como parâmetro no procedimento que implementa o método *fire()* do ator.

A tarefa 7 é a última na tradução de uma topologia SDF. O método *initialize()* corresponde a iniciação da topologia. Ele é traduzido em um procedimento cuja função é chamar os procedimentos de iniciação de cada ator atômico contido na topologia. O método *fire()* é traduzido em um procedimento que executa uma iteração do escalonamento associado a topologia. Os demais métodos de execução não são necessários, pois estão relacionados a operação do ambiente Ptolemy II.

Exemplo de transformação

A Figura 18 apresenta um exemplo de topologia controlada pelo modelo SDF.

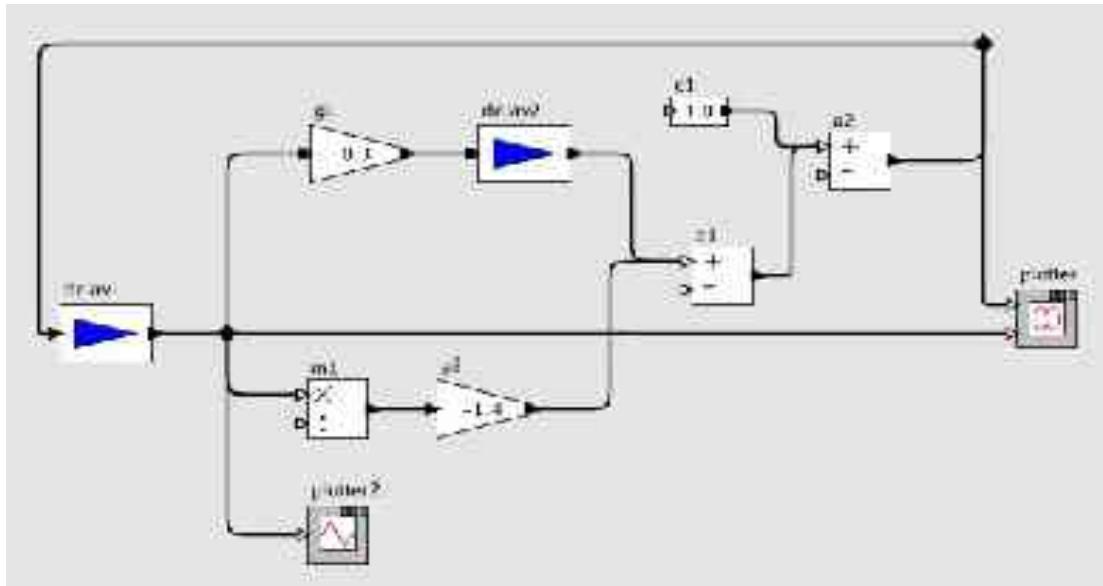


Figura 18 Exemplo de topologia SDF.

As instâncias *delay1* e *delay2* do ator atômico *SampleDelay* representam uma unidade de atraso. Desta forma, há um atraso entre os pares $(a2, g1)$, $(a2, m1)$, $(a2, plotter2)$, $(a2, plotter)$ e $(g1, a1)$. Duas instâncias (*plotter* e *plotter2*) do ator *SequencePlotter* são utilizadas. Este ator é responsável por desenhar gráficos bidimensionais, utilizando os recursos de interface gráfica da linguagem Java. Desta forma, estas instâncias não foram consideradas no processo de tradução.

Um escalonamento para a topologia da Figura 18 é: $(c1, g1, m1, g2, a1, a2)$. As taxas de amostragem de todos os atores desta topologia são homogêneas. Efetuando as etapas da alocação de memória, determinei a seguinte configuração de vetores:

Tabela 2 – Alocação de memória para a topologia da Figura 18.

Saída do ator	Tipo de vetor	Tamanho do vetor
<i>m1</i>	estático	1
<i>g2</i>	estático	1
<i>a1</i>	estático	1
<i>c1</i>	estático	1
<i>a2</i>	dinâmico	2
<i>g1</i>	dinâmico	2

A Figura 19 apresenta a implementação em vetores do resultado da alocação de memória.

```

1- #define G1_OUT_LEN 2
2- #define A2_OUT_LEN 2
3- #define UPDATE_INDEX(a, b) (a = (a + 1) % b)
4-
5- double g1_out[G1_OUT_LEN];
6- int g1_out_rindex = 0;
7- int g1_out_windex = 1;
8-
9- double m1_out;
10- double g2_out;
11- double a1_out;
12- double c1_out;
13-
14- double a2_out_1[A2_OUT_LEN];
15- int a2_out_1_rindex_g1 = 0;
16- int a2_out_1_rindex_m1 = 0;
17- int a2_out_1_windex = 1;

```

Figura 19 Resultado da alocação de memória.

Como o resultado do ator $a2$ é entrada para os atores $m1$ e $g1$, dois índices de leitura são necessários (linhas 15 e 16). A macro `UPDATE_INDEX` é utilizada para atualizar um índice após seu uso. Os vetores associados às saídas dos atores $m1$, $g2$, $a1$ e $c1$ possuem apenas uma posição. Neste caso particular, eles podem ser implementados através de variáveis escalares (linhas 9 a 12) e o índice é dispensável. A Figura 20 apresenta os procedimentos de execução para a topologia.

```

1- void initChaos(void) {
2-     a2_out_1[0] = 0.0;
3-     g1_out[0] = 0.0;
4- }
5-
6- void chaos(void) {
7-     Const(&c1_out, c1_value);                                /* C1 */
8-
9-     scale(a2_out_1[a2_out_1_rindex_g1],
10-           &g1_out[g1_out_windex], g1_factor);                  /* G1 */
11-     UPDATE_INDEX(a2_out_1_rindex_g1, A2_OUT_LEN);
12-     UPDATE_INDEX(g1_out_windex, G1_OUT_LEN);
13-
14-     MultiplyDivide(a2_out_1[a2_out_1_rindex_m1], a2_out_11
15-                     [a2_out_1_rindex_m1], &m1_out);                /* M1 */
16-     UPDATE_INDEX(a2_out_1_rindex_m1, A2_OUT_LEN);
17-
18-     scale(m1_out, &g2_out, g2_factor);                    /* G2 */
19-
20-     add(g1_out[g1_out_rindex], g2_out, &a1_out);        /* A1 */
21-     UPDATE_INDEX(g1_out_rindex, G1_OUT_LEN);
22-
23-     add(c1_out, a1_out, &a2_out_1[a2_out_1_windex]);    /* A2 */
24-     UPDATE_INDEX(a2_out_1_windex, A2_OUT_LEN);
25- }

```

Figura 20Procedimentos de execução da topologia.

O procedimento de iniciação está implementado entre as linhas 1 e 4. No exemplo em questão, a única iniciação necessária é atribuir os valores iniciais às conexões que apresentem atraso. Uma iteração do escalonamento está implementada entre as linhas 7 e 24. Note que instâncias diferentes de um mesmo ator, como por exemplo as instâncias *a1* e *a2*, estão associadas ao mesmo procedimento mas com parâmetros diferentes. Neste caso, os parâmetros são os vetores associados aos portos de E/S dos atores. Outro exemplo de variáveis que devem ser associadas aos parâmetros dos procedimentos são as variáveis internas e parâmetros do ator.

3.3.4 Transformação a partir do modelo SR

Uma topologia SR é executada através da iteração de um escalonamento de atores que encontra um ponto fixo para o sistema composto pelas relações da topologia. Desta forma, a técnica de tradução que apliquei para este modelo computacional é similar à do modelo SDF. Um processo é responsável por executar uma iteração do escalonamento. Os métodos de execução dos atores atômicos são

implementados através de procedimentos. Os métodos de execução da topologia implementados para este modelo são: *initialize()*, *fire()* e *postfire()*. O procedimento associado ao método *initialize()* inicializa todos os atores atômicos bem como atribui valores aos parâmetros dos atores da topologia. O procedimento que implementa o método *fire()* executa uma iteração do escalonamento dos atores. O procedimento associado ao método *postfire()* é chamado após o ponto fixo ter sido encontrado e executa o procedimento que implementa o método *postfire()* de cada ator atômico. Para um ator atômico, este método é utilizado para modificar o seu estado interno.

O cálculo do escalonamento de atores no modelo SR pode ser efetuado de maneira estática. O próprio ambiente Ptolemy II implementa um algoritmo para esta função, conforme descrito em [EDW97]. No capítulo 4, utilizei uma versão mais restrita da semântica do modelo SR. Nesta versão, é possível obter escalonamentos a partir de uma ordem topológica de um grafo associado à topologia. A etapa de alocação de memória é trivial, pois exatamente uma posição de memória é necessária para cada porto de saída.

No modelo SR, para cada relação entre atores está associada uma variável cujo valor pertence a uma CPO. Desta forma, a tradução deste modelo deve contemplar os diferentes estados que o valor de uma relação pode assumir. A fim de implementar esta característica, utilizei dois vetores de bits, onde cada bit representa uma relação da topologia. O primeiro vetor indica o estado (indefinido ou definido) enquanto que o segundo vetor indica a presença ou não de um valor, para todas as relações da topologia. Seja v a variável associada à relação conectada ao porto especificado como parâmetro de um método de comunicação. A tradução dos métodos de comunicação é:

- *get()*: retorna o valor de v ;
- *isKnown()*: retorna o valor do bit do vetor de estado associado a v ;
- *hasToken()*: retorna o valor do bit do vetor de presença de valor associado a v ;
- *hasRoom()*: sempre retorna o valor booleano verdade;
- *put()* e *broadcast()*: atribui um valor a v e um valor booleano verdade aos bits dos vetores de estado e presença de dados associados a v ;

- *sendClear()* e *broadcastClear()*: atribui um valor booleano verdade ao bit do vetor de estado e um valor booleano falso ao bit do vetor de presença de dados associados a *v*.

Exemplo de transformação

As Figuras 21 e 22 ilustram um exemplo de topologia associada ao modelo SR. A topologia da Figura 21 possui três instâncias (*Block1*, *Block2* e *Block3*) do ator hierárquico *Block*, cuja topologia é apresentada na Figura 22.

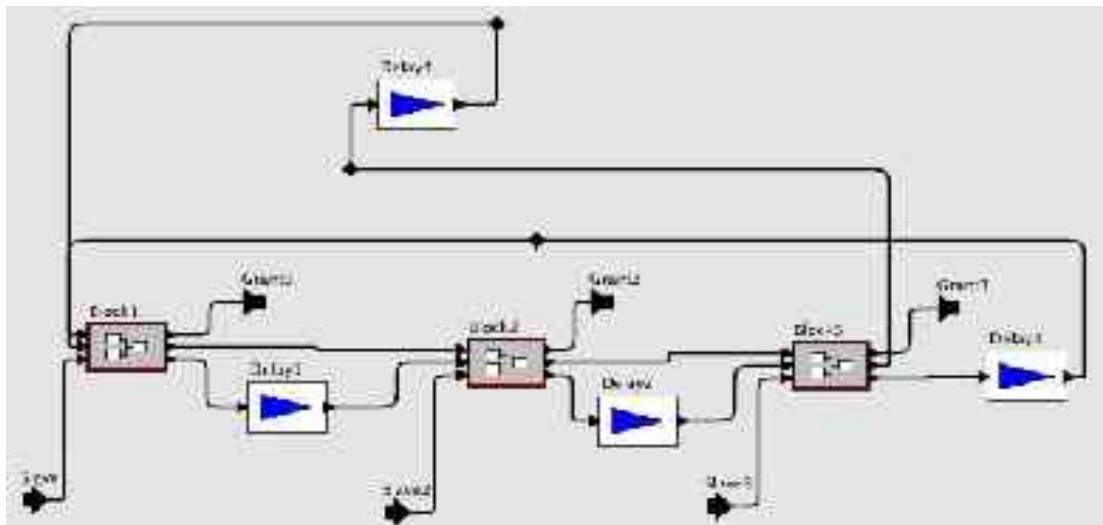


Figura 21 Exemplo de topologia SR.

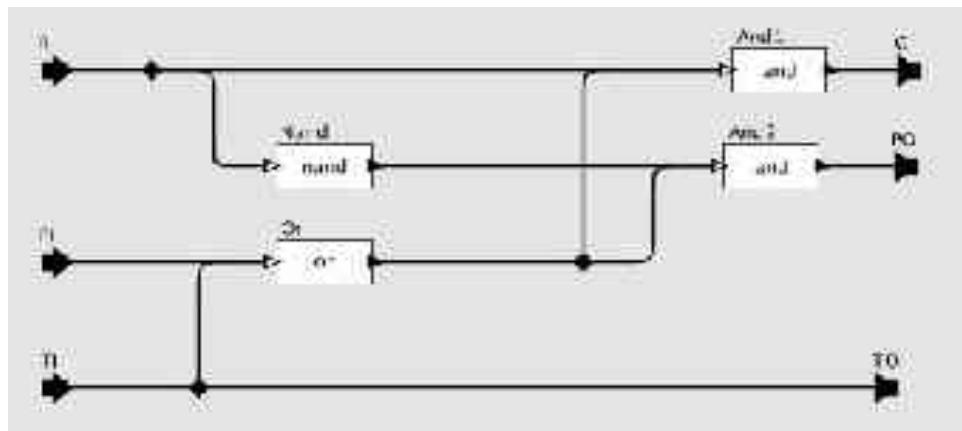


Figura 22 Topologia do ator hierárquico *Block*.

A Figura 23 apresenta os métodos *fire()* e *postfire()* do ator *NonStrictDelay*, presente na topologia da Figura 21.

```

1- public void fire() throws IllegalActionException {
2-     if(input.isKnown(0)) {
3-         if(input.hasToken(0)) {
4-             _currentToken = input.get(0);
5-         } else {
6-             _currentToken = AbsentToken.ABSENT;
7-         }
8-     }
9-
10-    if(_previousToken != null) {
11-        if(_previousToken == AbsentToken.ABSENT) {
12-            output.sendClear(0);
13-        } else {
14-            output.send(0, _previousToken);
15-        }
16-    } else {
17-        output.sendClear(0);
18-    }
19- }
20-
21- public boolean postfire() throws IllegalActionException {
22-     _previousToken = _currentToken;
23-     _currentToken = null;
24-     return super.postfire();
25- }
```

Figura 23Código para os métodos *fire()* e *postfire()* do ator *NonStrictDelay*.

No método *fire()* (linhas 1 a 19), é verificado se há um valor presente no canal 0 do porto de entrada *input* (linha 2 e 3). Caso haja um valor, este é lido (linha 4). Caso contrário, é armazenado o estado definido e ausente da relação conectada ao porto, caso esta esteja definida (linha 6). Entre as linhas 10 e 18, é determinado que valor o porto de saída *output* deve assumir no instante atual. O método *postfire()* (linhas 21 a 25) é responsável por verificar se o estado do porto de entrada *input* é definido ao final do instante, e armazená-lo em uma variável do ator.

A Figura 24 apresenta a tradução do ator *NonStrictDelay*.

```

1- void NonStrictDelayFire(char inputs, int inputBit,
2-     char *output, int outputBit, char previous, char *current) {
3-
4-     if(GET_BIT(knowFlags, inputBit)) {
5-         if(GET_BIT(presentFlags, inputBit)) {
6-             *current = inputs;
7-         }
8-     else {
9-         *current = -1;
10-    }
11- }
12-
13- if(previous != -1) {
14-     *output = previous;
15-     SET_BIT_TRUE(knowFlags, outputBit);
16-     SET_BIT_TRUE(presentFlags, outputBit);
17- }
18- else {
19-     SET_BIT_TRUE(knowFlags, outputBit);
20- }
21- }
22-
23- void NonStrictDelayPostfire(char *previous, char *current) {
24-     *previous = *current;
25-     *current = -1;
26- }

```

Figura 24 Tradução do ator *NonStrictDelay*.

O vetor de presença de dados é armazenado na variável *presentFlags* e o vetor de estado na variável *knowFlags*. O procedimento *NonStrictDelayFire()* implementa o método *fire()*. Seus parâmetros são as variáveis associadas aos portos do ator (*inputs* e *output*), bem como quais posições estas variáveis ocupam no vetor de presença de dados e de estados (*inputBit* e *outputBit*). Os parâmetros *previous* e *current* são utilizados para acessar os parâmetros do ator *NonStrictDelay*. As macros *GET_BIT* e *SET_BIT_TRUE* são utilizadas para obter o valor de um bit em um vetor e para atribuir um valor booleano verdade a um bit de um vetor, respectivamente. Note que o valor explícito de ausência de dados é utilizado nas linhas 6 e 11 do código do ator. Este valor é mapeado em um valor inválido para o tipo de dado associado. No caso da Figura 24, o tipo é booleano, e o valor inválido empregado é -1 (linhas 9 e 25). É possível que um tipo empregado não permita distinguir entre um valor válido e inválido. Por exemplo, um número real armazenado em um tipo *double*. Nestas condições, uma variável adicional deve ser empregada para indicar o valor inválido do tipo.

A Figura 25 apresenta a alocação de memória obtida para a especificação.

```

1- static char Delay1_O;
2- static char Delay2_O;
3- static char Delay3_O;
4- static char Delay4_O;
5-
6- static char B1_Nand_O;
7- static char B1_Or_O;
8- static char B1_And1_O;
9- static char B1_And2_O;
10-
11- static char B2_Nand_O;
12- static char B2_Or_O;
13- static char B2_And1_O;
14- static char B2_And2_O;
15-
16- static char B3_Nand_O;
17- static char B3_Or_O;
18- static char B3_And1_O;
19- static char B3_And2_O;
20-
21- static int knownFlags;
22- static int presentFlags;
23-
24- static char Delay1_previous;
25- static char Delay1_current;
26- static char Delay2_previous;
27- static char Delay2_current;
28- static char Delay3_previous;
29- static char Delay3_current;
30- static char Delay4_previous;
31- static char Delay4_current;

```

*Figura 25*Memória alocada para a topologia da Figura 21.

Entre as linhas 1 e 19, as variáveis que armazenam o valor de cada relação são declaradas. O vetor de presença de dados e de estados são declarados nas linhas 22 e 21 respectivamente. Entre as linhas 24 e 31, as variáveis de cada instância do ator *NonStrictDelay* são declaradas.

Utilizando-se as técnicas descritas no ítem 2.5, um escalonamento para a topologia da Figura 21 é: (*Delay1*, *Delay2*, *Delay3*, *Delay4*, *Block1*, *Block2*, *Block3*, *Delay1*, *Delay2*, *Delay3*, *Delay4*). Cada instância do ator hierárquico *Block* possui o seguinte escalonamento: (*Nand*, *Or*, *And1*, *And2*). A Figura 26 apresenta um trecho do procedimento que implementa o método *fire()* para o sistema, destacando o escalonamento de uma das instâncias do ator *Block*.

```

1-   in[0] = Request1;
2-   in_flag[0] = 0;
3-   out[0] = &B1_Nand_O;
4-   out_flag[0] = 10;
5-   LogicFunctionFire(in, 1, in_flag, out, out_flag, _AND, 1);
6-
7-   in[0] = Delay3_O;
8-   in_flag[0] = 8;
9-   in[1] = Delay4_O;
10-  in_flag[1] = 9;
11-  out[0] = &B1_Or_O;
12-  out_flag[0] = 11;
13-  LogicFunctionFire(in, 2, in_flag, out, out_flag, _OR, 0);
14-
15-  in[0] = Request1;
16-  in_flag[0] = 0;
17-  in[1] = B1_Or_O;
18-  in_flag[1] = 11;
19-  out[0] = &B1_And1_O;
20-  out_flag[0] = 12;
21-  LogicFunctionFire(in, 2, in_flag, out, out_flag, _AND, 0);
22-
23-  in[0] = B1_Nand_O;
24-  in_flag[0] = 10;
25-  in[1] = B1_Or_O;
26-  in_flag[1] = 11;
27-  out[0] = &B1_And2_O;
28-  out_flag[0] = 13;
29-  LogicFunctionFire(in, 2, in_flag, out, out_flag, _AND, 0);

```

Figura 26Trecho do procedimento que implementa o método fire().

O código da Figura 26 ilustra um exemplo onde diferentes instâncias de um mesmo ator possuem diferentes conexões. Todas as quatro instâncias contidas na topologia da Figura 22 são do ator *LogicFunction*. Este ator possui dois parâmetros: o primeiro permite escolher entre três funções lógicas e o segundo determina se a saída, cujo tipo é um valor booleano, deve ser negada ou não (sexto e sétimo parâmetros do procedimento *LogicFunctionFire()*). No caso da instância *Nand*, apenas um canal está associado ao seu porto de entrada. No caso da instância *Or*, dois canais estão presentes. A tradução de tal ator contempla múltiplos canais através de vetores cujas posições estão associadas às variáveis que contenham os dados do porto em questão. Na Figura 26, os vetores *in* e *out* estão associados aos canais do porto de entrada e saída respectivamente. Os vetores *in_flag* e *out_flag* contêm os bits dos vetores de presença de dados e estado dos respectivos canais. O procedimento *LogicFunctionFire()* implementa a função do ator *LogicFunction*.

A técnica de utilizar vetores para representar os canais de um porto de E/S também é utilizada na tradução de atores atômicos em todos os outros modelos computacionais.

3.3.5 Transformação a partir do modelo CSP

Uma instância de um ator atômico no modelo CSP tem uma correspondência direta com um processo. Associado a cada instância há um procedimento que é atrelado ao processo designado para executar sua função. Por sua vez, este procedimento chama os procedimentos de execução do ator, com os parâmetros adequados à instância. A Figura 27 apresenta um trecho de código exemplificando a tradução de atores atômicos.

```

1- ...
2- void procInstanciaAotor(cyg_addrword_t p) {
3-     for(;;) {
4-         if(procPrefireAotor(...)) {
5-             procFireAotor(...);
6-             if(!procPostfireAotor(...)) {
7-                 break;
8-             }
9-         }
10-    }
11- }
12-
13- ...
14- cyg_thread_create(prioridade, procInstanciaAotor, 1,
15-     "Instancia Aotor", pilha, TAMANHO_PILHA, identificador,
16-     processo);
17- ...

```

Figura 27 Tradução de um ator atômico associado a um processo.

Entre as linhas 14 e 16, o processo associado a uma instância é criado. O procedimento *procInstanciaAotor()* é especificado como sendo o conjunto de instruções a ser executado pelo processo. Este procedimento (linhas 2 a 10), por sua vez, executa os métodos de execução do ator em questão (*procPrefireAotor()*, *procFireAotor()* e *procPostfireAotor()*) indefinidamente.

Toda troca de dados é efetuada durante o *rendezvous* de dois processos. Os métodos de comunicação de dados são transformados da seguinte maneira:

- `get()`: obtém um dado;
- `isKnown()`, `hasToken()` e `hasRoom()`: sempre retorna o valor booleano verdade;
- `put()` e `broadcast()`: envia um dado;
- `sendClear()` e `broadcastClear()`: não possui tradução.

A troca de dados efetuada pelos métodos `get()`, `put()` e `broadcast()` deve seguir o protocolo de *rendezvous*. Isto significa que é necessário implementar um sincronismo entre qualquer par de processos que venham eventualmente a trocar algum dado. Implementei este sincronismo através de uma estrutura de dados, denominada de canal, que possui dois campos: um valor temporário utilizado para armazenar o dado sendo intercambiado e uma primitiva de sincronização do tipo *flag*. A única maneira de manipular o conteúdo de um canal é através de procedimentos específicos. São eles:

- `csp_init()`: inicializa a estrutura de dados do canal;
- `csp_write()`: escreve um dado no canal, respeitando o protocolo de *rendezvous*;
- `csp_read()`: lê um dado do canal, respeitando o protocolo de *rendezvous*.

Para cada conexão entre dois atores, um canal deve ser criado. Toda chamada, por parte do ator fonte, ao método `put()`³⁶ deve ser transformada em uma chamada ao procedimento `csp_write()` do devido canal. Da mesma maneira, toda chamada ao método `get()` deve ser substituída por uma chamada ao procedimento `csp_read()` do devido canal.

A Figura 28 ilustra o código fonte dos procedimentos `csp_read()` e `csp_write()`.

³⁶ Um método `broadcast()` é igual a chamar o método `put()` para cada canal do porto.

```

1- void csp_write(cspchannel_t *channel, void *data) {
2-     channel->data = data;
3-
4-     cyg_flag_setbits(&channel->channel_flag, caller_starts_FLAG);
5-     cyg_flag_wait(&channel->channel_flag, accepter_ends_FLAG,
6-                   CYG_FLAG_WAITMODE_OR);
7-     cyg_flag_maskbits(&channel->channel_flag,
8-                       ~accepter_ends_FLAG);
9- }
10-
11- void *csp_read(cspchannel_t *channel) {
12-     void *data;
13-
14-     cyg_flag_wait(&channel->channel_flag, caller_starts_FLAG,
15-                   CYG_FLAG_WAITMODE_OR);
16-     cyg_flag_maskbits(&channel->channel_flag,
17-                       ~caller_starts_FLAG);
18-
19-     data = channel->data;
20-
21-     cyg_flag_setbits(&channel->channel_flag, accepter_ends_FLAG);
22-
23-     return data;
24- }
```

Figura 28 Os procedimentos *csp_write()* e *csp_read()*.

O parâmetro *channel* (linhas 1 e 11) é um canal entre dois processos. No procedimento *csp_write()*, após o dado ter sido armazenado na variável auxiliar *data* (linha 2), é atribuída a *flag* utilizada para a sincronização (*channel_flag*) um padrão de bits que sinaliza que o ator fonte completou a escrita (linha 4). Em seguida, o procedimento suspende o processo do ator fonte, esperando por um padrão de bits que indica que a leitura do dado foi efetuada (linhas 5 e 6). Quando isto ocorrer, eventualmente o processo do ator fonte retorna à execução e reinicializa o conteúdo da *flag* do canal (linha 8). No procedimento *csp_read()*, inicialmente é verificado o padrão de bits da *flag* do canal (linhas 14 e 15). Caso um dado tenha sido escrito, o valor da *flag* é reiniciado (linhas 16 e 17) e o dado é lido (linha 19). Em seguida, um padrão de bits é escrito na *flag* do canal (linha 21), a fim de indicar que a leitura foi efetuada.

A fim de simplificar o tradução, sem comprometer sua utilidade, contepliei uma versão simplificada dos comandos de comunicação condicional presentes no modelo CSP. Dado um conjunto qualquer de processos, apenas um pode efetuar a comunicação através de um comando condicional. Associo a cada processo que

utiliza a comunicação condicional uma *flag* cuja finalidade é indicar quais processos que desejam trocar dados com ele e estão prontos para completar um comando de comunicação. Então, algum destes comandos é escolhido aleatoriamente, desde que a condição associada ao comando seja verdadeira. A única modificação necessária aos procedimentos *csp_write()* e *csp_read()* é a indicação para o processo efetuando o comando condicional que um comando de comunicação está pronto. Isto é realizado atribuindo um valor específico a *flag* associado ao processo que está realizando o comando condicional.

A implementação do modelo CSP no ambiente Ptolemy II apresenta uma noção de tempo fundamentada na suspensão de um processo por uma duração especificada, através do método *fireAt()*. Esta noção pode ser traduzida diretamente, através da substituição de tal método pelo procedimento *cyg_thread_delay()*.

Para toda topologia, apenas um procedimento é gerado. Ele possui quatro funções: (1) atribuir valores iniciais aos parâmetros e variáveis internas de cada instância de ator atômico; (2) iniciar os canais de comunicação; (3) criar os processos; (4) iniciar o escalonador de processos. O escalonamento dos processos é efetuado pelo núcleo do sistema eCos.

Exemplo de transformação

A Figura 29 apresenta uma topologia controlada pelo modelo CSP.

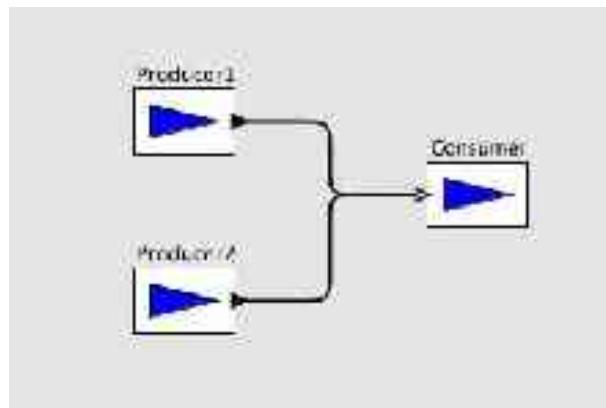


Figura 29 Exemplo de topologia CSP.

A topologia da Figura 29 apresenta dois canais: um entre as instâncias

Producer1 e *Consumer* e outro entre as instâncias *Producer2* e *Consumer*. O ator atômico *Consumer* utiliza um comando condicional para obter um dado de qualquer ator conectado a ele. A Figura 30 ilustra o código fonte deste ator e de uma instância do ator *Producer*.

```

1- void producer2(cyg_addrword_t data) {
2-     for(;;) {
3-         csp_writeCond(&channel2, &consumerFlag, 1, '2');
4-     }
5- }
6-
7- void consumer(cyg_addrword_t data) {
8-     int plrv, p2rv;
9-     int guard1, guard2;
10-    char c;
11-    int choosed, flagvalue, waitcond;
12-
13-    do {
14-        plrv = p2rv = 0;
15-        do {
16-            guard1 = plrv < 20;
17-            guard2 = plrv < 20;
18-
19-            waitcond = guard1 | (guard2 << 1);
20-            cyg_flag_wait(&consumerFlag, waitcond,
21-                CYG_FLAG_WAITMODE_OR);
22-            flagvalue = (int) cyg_flag_peek(&consumerFlag);
23-            choosed = chooseRV(flagvalue, 2);
24-            cyg_flag_maskbits(&consumerFlag, ~(0x1 << choosed));
25-
26-            switch(choosed) {
27-                case 0: {
28-                    c = csp_read(&channel1);
29-                    plrv++;
30-                } break;
31-
32-                case 1: {
33-                    c = csp_read(&channel2);
34-                    p2rv++;
35-                } break;
36-
37-                default: break;
38-            }
39-        } while(guard1 || guard2);
40-    } while(1);
41- }
```

Figura 30 Código dos atores *Producer* e *Consumer* da Figura 29.

O procedimento *producer2()* (linhas 1 a 5) está associado ao processo do ator

Producer2. Ele executa indefinidamente o envio de um dado para o ator *Consumer*, utilizando a versão modificada do procedimento *csp_write()*, uma vez que o ator *Consumer* utiliza um comando de comunicação condicional. O segundo parâmetro do procedimento *csp_writecond()* (linha 3) é a *flag* associada ao processo do ator *Consumer*, enquanto que o terceiro parâmetro é o bit desta *flag* associado ao processo da instância *Producer2*. O procedimento para a instância *Producer1* é igual a da instância *Producer2*.

O ator *Consumer* utiliza um comando de comunicação condicional (linhas 13 a 40) para obter indefinidamente dados de qualquer canal associado a seu porto de entrada. A primeira etapa da comunicação condicional é a determinação de quais condições são válidas (linhas 16 a 19). Em seguida, o processo aguarda que pelo menos um canal conectado esteja apto a efetuar uma troca de dados (linhas 20 e 21), isto é, pelo menos um ator conectado deseja efetuar uma comunicação. Após escolhido algum canal apto a comunicar (linhas 22 e 23), o respectivo procedimento de comunicação é executado (linhas 26 a 37).

3.3.6 Transformação a partir do modelo PN

A tradução de uma topologia controlada pelo modelo PN é muito similar a de uma controlada pelo modelo CSP. Cada instância de um ator atômico é associada a um processo. Os procedimentos de execução são implementados conforme ilustrado na Figura 27. Para toda topologia, um procedimento é responsável pela inicialização de todos os parâmetros de atores, canais da topologia, criação dos processos e iniciação do escalonador. A tradução dos métodos de comunicação de um ator atômico é igual a aplicada ao modelo CSP.

Todavia, os processos oriundos de uma topologia PN devem se comunicar respeitando as restrições impostas pela semântica do modelo, isto é, um canal entre dois processos deve implementar uma fila com leitura bloqueante. Desenvolvi duas versões para um canal no modelo PN: (1) fila de capacidade limitada e imutável; (2) fila de capacidade variável. A primeira solução não contempla a noção de uma fila infinita de dados. Ela pode ser empregada quando a capacidade de dados em cada canal nunca ultrapasse um dado limite. O desenvolvedor é obrigado a especificar a

capacidade de cada fila, procurando evitar um *deadlock* devido a filas cheias. A vantagem deste tipo de canal é sua simplicidade. A desvantagem, é a provável perda de memória de dados devido a não utilização da capacidade total de cada fila e a possibilidade de ocorrer deadlocks devido a uma escolha incorreta da capacidade de cada fila. A Figura 31 apresenta o código para a estrutura desse tipo de canal.

```

1-  typedef struct {
2-    void **buffer;
3-
4-    unsigned int write_index;
5-    unsigned int read_index;
6-    unsigned int size;
7-
8-    cyg_sem_t channel_sem;
9-    int free_elements;
10-   cyg_flag_t channel_flag;
11- } kpnchannel_t;
```

Figura 31Estrutura de dados para um canal PN com tamanho fixo.

A estrutura é composta pela região de memória utilizada para armazenar os dados (linha 2), variáveis indicando a próxima posição de escrita (linha 4) e leitura (linha 5), capacidade total da fila (linha 6) e a quantidade de posições restantes na fila (linha 9). A estrutura também possui um semáforo (linha 8) e uma *flag* (linha 10),utilizados pelos procedimentos de escrita e leitura do canal. A Figura 32 apresenta os procedimentos de leitura e escrita no canal.

```

1- void * kpn_read(kpnchannel_t *channel) {
2-     void *data;
3-
4-     cyg_flag_wait(&channel->channel_flag, 0x1,
5-                   CYG_FLAG_WAITMODE_OR);
6-
7-     data = channel->buffer[channel->read_index];
8-     channel->read_index = (channel->read_index + 1)%channel->size;
9-
10-    cyg_semaphore_peek(&channel->channel_sem,
11-                      &channel->free_elements);
12-
13-    if((unsigned int) channel->free_elements >=
14-        (channel->size - 1)) {
15-        cyg_flag_maskbits(&channel->channel_flag, ~0x01);
16-    }
17-    cyg_semaphore_post(&channel->channel_sem);
18-
19-    return data;
20- }
21-
22- void kpn_write(kpnchannel_t *channel, char *data) {
23-     cyg_semaphore_wait(&channel->channel_sem);
24-
25-     channel->buffer[channel->write_index] = data;
26-     channel->write_index =
27-         (channel->write_index + 1)%channel->size;
28-
29-     cyg_flag_setbits(&channel->channel_flag, 0x01);
30- }
```

Figura 32Procedimentos de escrita e leitura em um canal PN de capacidade fixa.

O semáforo é utilizado pelo procedimento de escrita (linha 23) a fim de determinar se há alguma posição livre no canal. Caso contrário, o processo que chamou o procedimento será suspenso. Para voltar a executar, é necessário que o processo responsável por ler os dados do canal chame o procedimento de leitura (linha 17). A *flag* é utilizada pelo procedimento de leitura a fim de determinar se há dados no canal (linha 4). Caso contrário, o processo permanecerá suspenso até que algum dado seja escrito (linha 29).

A segunda implementação do canal PN permite alterar a capacidade de um canal lotado de dados. O próprio procedimento de escrita determina se não há mais espaço na fila de dados e aumenta sua capacidade se necessário. A fim de tornar a implementação mais eficiente, a quantidade de posições de memória adicionada à fila é sempre um bloco de tamanho igual a capacidade inicial especificada. Quando

um bloco não for mais necessário, ele é desalocado da memória. Por exemplo, uma fila tem no inicio da execução capacidade para 5 dados, quando for necessário aumentar sua capacidade, esta é aumentada para 10 dados. A vantagem desta implementação é contemplar a noção de fila infinita³⁷ sem a necessidade de um monitoramento do estado de cada fila. A desvantagem é o tempo extra necessário para alocar e desalocar a memória e uma eventual ineficiência no uso da memória. A Figura 33 apresenta a estrutura de dados para a fila de tamanho variável.

```

1-  typedef struct {
2-    void **buffer;
3-    int *free_elements;
4-
5-    unsigned int size;
6-
7-    unsigned int wBuf;
8-    unsigned int wIdx;
9-
10-   unsigned int rBuf;
11-   unsigned int rIdx;
12-
13-   cyg_flag_t channel_flag;
14-   cyg_mutex_t channelMutex;
15- } kpnchannelInf_t;
```

Figura 33 Estrutura de dados para fila de capacidade variável.

Nesta estrutura, não é mais necessário o semáforo, uma vez que o procedimento de escrita não suspende o respectivo processo. A memória é alocada progressivamente, em regiões cujo tamanho é determinado por uma variável (linha 5). A matriz *free_elements* (linha 3) indica quantas posições livres existem em cada bloco alocado. Duas variáveis são utilizadas para indicar a próxima posição de memória livre a ser escrita (linhas 7 e 8) e outras duas para a próxima posição de memória a ser lida (linhas 10 e 11). A Figura 34 apresenta o procedimento de leitura para o canal de capacidade variável.

37 Limitada pela implementação física do sistema.

```

1- void kpnInf_read(kpnchannelInf_t *channel) {
2-     void data, **newbuffer;
3-     int *newfree;
4-
5-     if(channel->free_elements[channel->rBuf] == channel->size) {
6-         if(channel->wBuf != channel->rBuf) {
7-             free(channel->buffer[channel->rBuf]);
8-
9-             channel->rBuf++;
10-            channel->rIdx = 0;
11-            if(channel->rBuf == channel->wBuf) {
12-                newbuffer = (char **) malloc(sizeof(char *));
13-                newfree = (int *) malloc(sizeof(int));
14-
15-                newfree[0] = channel->free_elements[channel->rBuf];
16-                newbuffer[0] = channel->buffer[channel->rBuf];
17-
18-                free(channel->free_elements);
19-                free(channel->buffer);
20-
21-                channel->free_elements = newfree;
22-                channel->buffer = newbuffer;
23-
24-                channel->wBuf = 0;
25-                channel->rBuf = 0;
26-            }
27-        }
28-    else {
29-        cyg_flag_wait(&channel->channel_flag, 0x1,
30-                      CYG_FLAG_WAITMODE_OR |
31-                      CYG_FLAG_WAITMODE_CLR);
32-    }
33- }
34- data = channel->buffer[channel->rBuf][channel->rIdx];
35- channel->rIdx = (channel->rIdx + 1) % channel->size;
36-
37- channel->free_elements[channel->rBuf]++;
38- cyg_flag_maskbits(&channel->channel_flag, ~0x01);
39-
40- return data;
41- }

```

Figura 34Procedimento de leitura de um canal PN com capacidade variável.

Inicialmente, o procedimento determina se o bloco de memória sendo utilizado está sem nenhum dado (linha 5). Se possuir, o dado é lido (linha 34) e as variáveis do canal são atualizadas. Caso contrário, é necessário verificar se o canal não possui nenhum dado ou se há dados em outro bloco de memória pertencente ao canal (linha 6). Se não houver dados, é necessário bloquear o processo até algum dado ser escrito no canal (linhas 29 a 31). Caso haja dados, então é necessário liberar

a memória referente ao bloco vazio e atualizar as variáveis do canal (linhas 7 a 25).

Um parâmetro importante de um canal é sua capacidade inicial. Adotei o valor máximo obtido nas execuções do ambiente Ptolemy II. Desta forma, o desenvolvedor valida sua especificação no ambiente Ptolemy II e, uma vez satisfeita, é possível obter a capacidade máxima de cada fila. Quanto mais relevante for o conjunto de dados utilizado para validar a especificação, maior será a probabilidade de que este valor final seja a capacidade necessária para a fila, caso seja possível executar a topologia com quantidade limitada de memória.

Como no modelo CSP, o escalonador de processos do ambiente eCos é utilizado sem nenhuma alteração. Desta forma, não implementei a técnica desenvolvida por Parks [PAR95] que consiste em suspender um processo na tentativa de escrever em uma fila cheia. Para tal, seria necessário alterar o escalonador de processos do ambiente eCos ou adicionar um processo responsável por monitorar o estado de cada canal, e na ocorrência de um *deadlock* artificial, aumentar a capacidade das filas cheias. A principal razão para esta decisão é que ainda não está claro qual é a técnica mais adequada para escalonar processos em uma topologia PN. A técnica de Parks possui a vantagem de executar uma topologia com memória limitada, quando isto for possível, mas apresenta ineficiências [BAS01]. Desta forma, preferi utilizar uma técnica simples e eficiente, suficiente para desenvolver os estudos desta tese.

Exemplo de transformação

A Figura 35 apresenta uma topologia para ilustrar a transformação a partir do modelo PN, enquanto que a Figura mostra o código fonte simplificado obtido a partir da transformação.

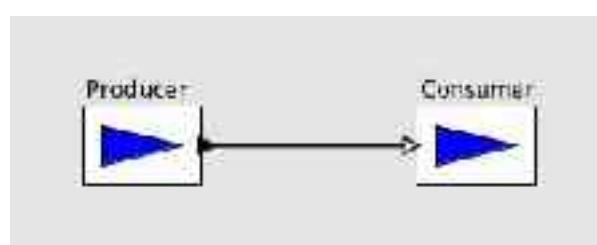


Figura 35 Exemplo de topologia no modelo PN.

```

1- static kpnchannel_t buffer;
2-
3- externC void cyg_start( void ) {
4-     cyg_thread_create(4, producer, (cyg_addrword_t) 0,
5-         "Producer", (void *) stack[0], 4096,
6-         &producerHandle, &thread_s[0]);
7-     cyg_thread_resume(producerHandle);
9-
10-    cyg_thread_create(4, consumer, (cyg_addrword_t) 1,
11-        "Consumer", (void *) stack[1], 4096,
12-        &consumerHandle, &thread_s[1]);
13-    cyg_thread_resume(consumerHandle);
14-
15-    kpn_init(&buffer, 5);
16-
17-    Cyg_Scheduler::scheduler.start();
18- }
19-
20- void producer(cyg_addrword_t data) {
21-     char texto[5];
22-     for(;;) {
23-         texto[0] = 'a';
24-         texto[1] = 'b';
25-         texto[2] = 'c';
26-         texto[3] = 'd';
27-         texto[4] = '\0';
28-         kpn_write(&buffer, (void *) texto);
29-     }
30- }
31-
32- void consumer(cyg_addrword_t data) {
33-     char *texto;
34-
35-     for(;;) {
36-         texto = (char *) kpn_read(&buffer);
37-     }
38- }

```

Figura 36 Resultado da transformação aplicada a topologia da Figura 35.

Entre as linhas 4 e 13 os processos associados aos atores *Producer* e *Consumer* são criados. O canal entre eles é iniciado na linha 15, com uma capacidade para 5 dados. A versão do canal utilizada é a com capacidade fixa. O procedimento executado pelo processo do ator *Producer* está descrito entre as linhas 20 e 30, enquanto que processo do ator *Consumer* entre as linhas 32 e 38.

CAPÍTULO 4 – VALIDAÇÃO DE ESPECIFICAÇÕES EXECUTÁVEIS

“Tudo que você precisa na vida é ignorância e confiança.

Então o sucesso será certo.”

Mark Twain

4.1 Introdução

Utilizar um ambiente baseado na interação de atores apresenta algumas vantagens. Dentre elas, a possibilidade de empregar vários modelos computacionais ao longo da criação das especificações executáveis, alguns simultaneamente em uma mesma especificação, destaca-se. Entretanto, um desenvolvedor pode fazer duas perguntas fundamentais com relação à combinação de uma especificação e um modelo computacional:

1. Que conjunto de modelos computacionais posso utilizar para capturar o sistema sendo desenvolvido ?
2. Dado vários conjuntos de modelos, qual é o melhor ?

Para responder a segunda pergunta, é necessário determinar um conjunto de parâmetros para comparação de diferentes soluções. No contexto desta tese, um modelo computacional pode ser dito melhor que outro quando a implementação obtida utilizando o primeiro modelo é mais eficiente que a gerada com o segundo modelo. Descrevi critérios para classificar a eficiência de uma implementação no capítulo 3. Desta forma, estabelecida uma maneira eficiente de se transformar uma especificação baseada na interação de atores em uma baseada na comunicação de processos, torna-se possível responder à pergunta (2).

A primeira pergunta pode ser desmembrada em duas:

1. a) A especificação é válida para um dado modelo computacional ?
1. b) Para os modelos válidos, a especificação captura o comportamento desejado para o sistema ?

Uma maneira de se responder à pergunta 1.b) é executar a especificação para diferentes casos de teste, e determinar se os resultados obtidos são os esperados.

Defino uma especificação executável como válida para um modelo computacional quando a função de cada ator e a composição de todos os atores em uma topologia respeitam as regras impostas pela semântica do modelo em questão. Por exemplo, um ator que consome uma quantidade de dados de um porto de entrada com base no valor obtido de outro porto de entrada não pode ser capturado no modelo SDF, pois viola a regra das taxas de amostragem constantes. Desta forma, a fim de responder à pergunta 1.a), é necessário estabelecer para cada modelo computacional:

- Quais são as regras impostas aos atores e as topologias;
- Como determinar se a especificação satisfaz todas essas regras.

Dentre os seis modelos computacionais que utilizei durante o desenvolvimento desta tese, escolhi os modelos SDF e SR a fim de estudar uma estratégia para responder a pergunta 1.a). Foram duas as principais razões porque escolhi estes dois modelos computacionais em particular:

1. Ambos os modelos apresentam um escalonamento estático, uma vantagem na implementação das especificações, uma vez que elimina a penalidade do escalonamento dinâmico;
2. Os modelos foram desenvolvidos para capturar tipos de sistemas distintos. O modelo SDF é adequado a aplicações orientadas ao processamento contínuo de dados e o modelo SR a aplicações com forte interação com o meio físico.

No próximo item descrevo quais são as regras impostas por estes dois modelos. Enuncio este conjunto de regras como consequência imediata da definição da semântica dos modelos SDF e SR, e que foram descritas no capítulo 2. No restante deste capítulo, apresento uma estratégia para validação e sua implementação em uma ferramenta automática de análise de especificação executáveis.

4.2 Condições para uma especificação válida

4.2.1 SDF

Para um ator atômico, deve-se garantir a seguinte condição:

1. *Um ator atômico SDF deve produzir e/ou consumir um número constante de dados durante toda a execução.*

Dada uma topologia composta somente por atores SDF válidos, deve-se assegurar que:

1. *As taxas de amostragem de todos os atores devem ser consistentes;*
2. *A topologia deve ser livre de deadlock.*

Satisfitas estas duas condições, é possível encontrar um escalonamento para a topologia, e consequentemente, executá-la.

4.2.2 SR

Para um ator atômico ser válido para o modelo SR, duas condições devem ser satisfeitas:

1. *Um ator pode produzir no máximo um valor por porto de saída em um instante;*
2. *O ator deve implementar uma função monotônica.*

A fim de ser possível encontrar um ponto fixo para um topologia, a seguinte condição deve ser satisfeita:

1. *A topologia deve ser livre de deadlock.*

Um *deadlock* em uma topologia SR ocorre quando pelo menos um valor de uma relação, que independentemente do valor dos portos de entrada da topologia, nunca apresenta um estado definido. Isto ocorre quando há um ciclo de atores onde não há nenhum do tipo *NonStrict*.

4.3 Polimorfismo

A função de um ator atômico pode ser capturada em um código que use características específicas de um modelo computacional. Métodos, estruturas sintáticas específicas (como portos próprios a um modelo), características semânticas particulares (como o modelo de tempo), são exemplos de construções particulares a um modelo computacional. Assim, tal ator não é compatível com outros modelos computacionais. Capturar a mesma função em outro modelo computacional implica em reescrever parcialmente ou totalmente o código do ator. Isto acarreta em um tempo adicional, além de possibilitar a introdução de erros no código do ator.

Uma característica importante do ambiente Ptolemy II é a presença de uma semântica abstrata³⁸. Utilizando-a, um desenvolvedor é capaz de capturar atores atômicos que podem ser utilizados em mais de um modelo computacional. Um ator que é válido para mais de um modelo computacional é dito **polimórfico**. A utilização de atores polimórficos permite explorar diferentes soluções através de diferentes combinações de modelos computacionais.

A Figura 37 apresenta o método *fire()* de um ator atômico cuja função é somar o valor dos dados presentes no porto de entrada *plus*.

```

1- public void fire() throws IllegalActionException {
2-     Token sum = plus.get(0);
3-     for(int i = 1;i < plus.getWidth();i++) {
4-         sum.add(plus.get(i));
5-     }
6-     output.send(0, sum);
7- }
```

Figura 37 Um ator específico para semânticas de fluxo de dados.

38 Implementada no pacote *Actor*. Vide Anexo A.

Embora não utilize nenhum método específico de um modelo computacional, o código da Figura 37 leva em consideração o fato de que é garantida a presença de dados nos portos de entrada. Desta forma, este ator pode ser utilizado em topologias controladas por qualquer modelo que utilize a semântica de fluxo de dados. Assim, podemos considerar o método da Figura 37 como polimórfico. A Figura 38 apresenta um método com a mesma funcionalidade do descrito na Figura 37, mas válido para um número maior de modelos computacionais.

```

1- public void fire() throws IllegalActionException {
2-
3-     Token sum = null;
4-     for(int i = 0;i < plus.getWidth();i++) {
5-         if(plus.hasToken(i)) {
6-             if(sum == null) {
7-                 sum = plus.get(i);
8-             }
9-             else {
10-                 sum.add(plus.get(i));
11-             }
12-         }
13-     }
14-     if(sum != null) {
15-         output.send(0, sum);
16-     }
17- }
```

Figura 38 Método *fire()* de um ator polimórfico.

O código da Figura 38 não assume que será executado somente quando houver dados em todos os portos de entrada. O método *hasToken()* (linha 5) é utilizado para determinar em qual canal do porto *plus* há dados. É possível que nada seja produzido após uma execução do método (linhas 14 a 16). O código da Figura 38 pode ser utilizado em modelos com semântica de fluxo de dados, e seu comportamento é idêntico ao da Figura 37. Entretanto, ele também pode ser utilizado em uma topologia controlada pelos modelos DE ou SR. O ponto fundamental a ser observado no exemplo da Figura 38, é que embora possa apresentar um comportamento diferente dependendo do modelo computacional onde tal código estiver embutido, em todos os casos trata-se de um código construído corretamente.

Existem certas funções que não são possíveis de serem implementadas através

de atores polimórficos, pois dependem diretamente de uma característica específica de um modelo computacional. Um exemplo é um ator de atraso, isto é, um ator que produz no instante $n+1$ um valor baseado em dados consumidos no instante n .

4.4 Estratégia para validação

As regras impostas por um modelo computacional podem ser infringidas em três locais:

1. Na captura da função de um ator atômico;
2. Na interação entre um grupo de atores atômicos em uma mesma topologia;
3. Na interação com um ator hierárquico opaco.

Exemplos da primeira situação são o emprego de comandos específicos de um modelo computacional quando o ator estiver contido em uma topologia controlada por outro modelo computacional, ou quando uma determinada construção de comandos viola a semântica de comunicação do modelo.

Uma topologia composta por atores válidos não necessariamente resulta em uma especificação válida. Por exemplo, um *deadlock* pode ocorrer mesmo quando todos os atores são válidos.

A terceira situação é composta por dois casos: (1) um ator hierárquico que é controlado pelo mesmo modelo computacional da topologia onde ele está contido; (2) um ator hierárquico que utiliza um modelo computacional diferente daquele associado à topologia onde ele está contido.

A partir das três situações enumeradas acima, defino a seguinte estratégia para validação de uma especificação:

1. Classificar cada ator atômico em específico ou polimórfico;
2. Determinar se uma topologia é específica a um modelo, através da presença de atores específicos;
3. Para os atores atômicos polimórficos, verificar se são válidos para os

- modelos SDF e/ou SR;
4. Determinar se uma topologia que contenha somente atores válidos para os modelos SDF e/ou SR é também válida;
 5. Verificar se todas as interações com atores hierárquicos opacos estão corretas.

4.5 A ferramenta FASI

A fim de ser possível implementar as etapas da estratégia proposta no ítem anterior, é necessário analisar o código fonte de cada ator atômico e a representação de cada topologia. Um tipo de técnica que pode ser utilizada para tal fim é a análise estática, isto é, uma análise efetuada durante a etapa de criação da especificação executável. Para um ator atômico, cria-se uma representação de seu código que é analisada por algoritmos que determinam se as regras impostas por um modelo computacional são satisfeitas. Para um ator hierárquico, utiliza-se as informações coletadas de cada ator atômico, bem como as conexões entre cada ator da topologia.

Implementei uma ferramenta automática para análise estática, denominada de FASI³⁹. Ela recebe como entrada uma especificação executável e após processá-la, gera um conjunto de mensagens ao desenvolvedor. Estas mensagens explicitam a presença de erros, apontando qual regra foi violada e onde, ou informações úteis ao desenvolvedor. Por exemplo, a ferramenta indica para cada ator, hierárquico ou atômico, se ele é específico ou não, e para quais modelos computacionais ele é válido. O apêndice C apresenta informações adicionais sobre a ferramenta FASI.

4.5.1 Etapas da análise

A execução dos algoritmos de validação para um modelo computacional constitui a última etapa do processo de análise de uma especificação. Antes, esta deve ser representada em estruturas de dados e algoritmos devem extrair informações necessárias para a validação. A Figura 39 apresenta o diagrama das etapas da análise. Elas seguem a estratégia proposta na seção 4.4.

³⁹ Ferramenta de Análise de Semânticas de Interação.

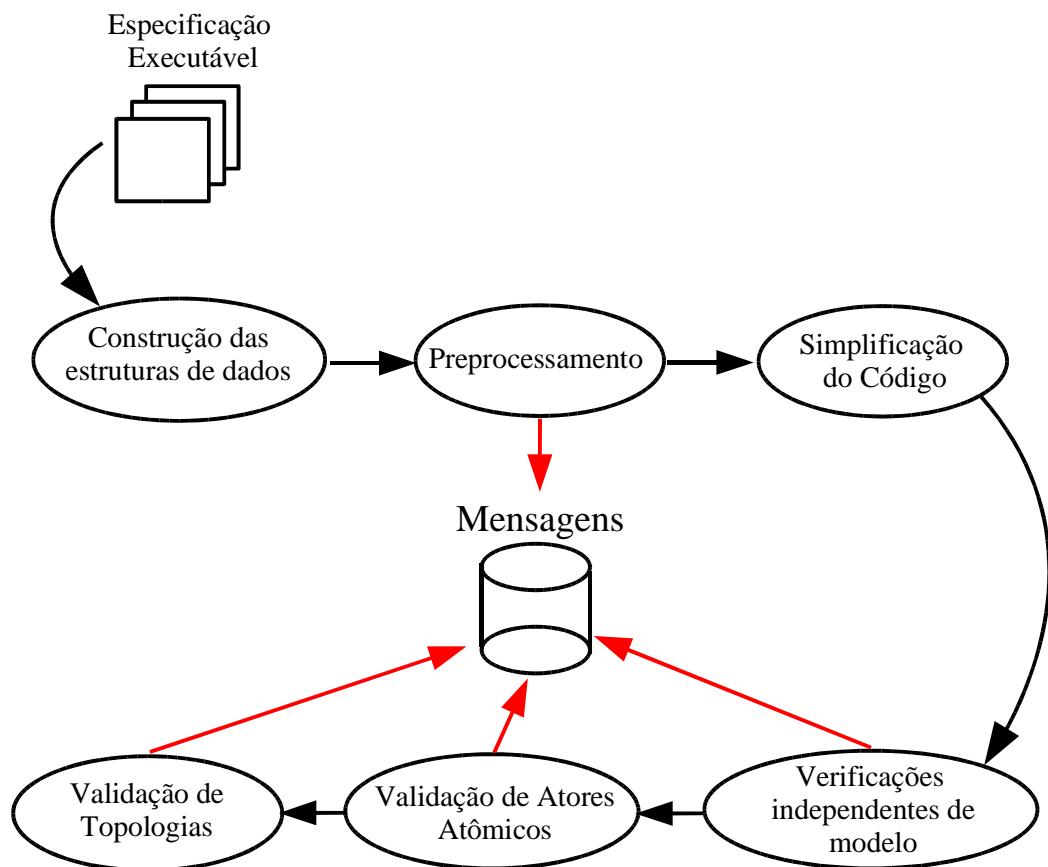


Figura 39 As etapas da análise da ferramenta FASI.

As etapas são:

1. Construção das estruturas de dados: processa o código fonte de cada ator atômico criando as estruturas de dados utilizadas subsequentemente no processo de análise. É assumido que cada ator é descrito por um código fonte correto;
2. Pré-processamento: identifica se um ator atômico foi escrito especificamente para um modelo computacional e se o projetista especificou para qual modelo computacional ele é válido ou não;
3. Simplificação do código: utiliza algoritmos conhecidos de otimização [APP98] para reduzir o código fonte de um ator atômico, eliminando comando desnecessários, e assim aumentar a precisão da análise;
4. Verificações independentes de modelo: verifica situações não válidas e não

dependentes da semântica de um modelo computacional em particular;

5. Validação de atores atômicos: executa os algoritmos de validação dos modelos SDF e SR sobre todos os atores atômicos polimórficos da especificação;
6. Validação de topologias: executa os algoritmos de validação dos modelos SDF e SR sobre todos os atores hierárquicos da especificação e sobre todas as interações com atores hierárquicos opacos.

4.5.1.1 Estrutura de dados

A estrutura de dados contém informações referentes aos atores hierárquicos, aos atores atômicos e às classes Java que implementam os atores atômicos. Dada uma especificação executável, determinam-se os atores atômicos que são instanciados em cada topologia. Para cada ator atômico, determina-se a hierarquia de classes que o implementa. Para uma classe, é efetuada a análise do respectivo código fonte, e constróem-se grafos de fluxo de controle (GFC) [APP98], tabelas de símbolos para as variáveis e métodos declarados no GFC e um grafo derivado do GFC, denominado de **grafo de métodos** (GM). A Figura 40 ilustra o GM para o método *fire()* da Figura 38.

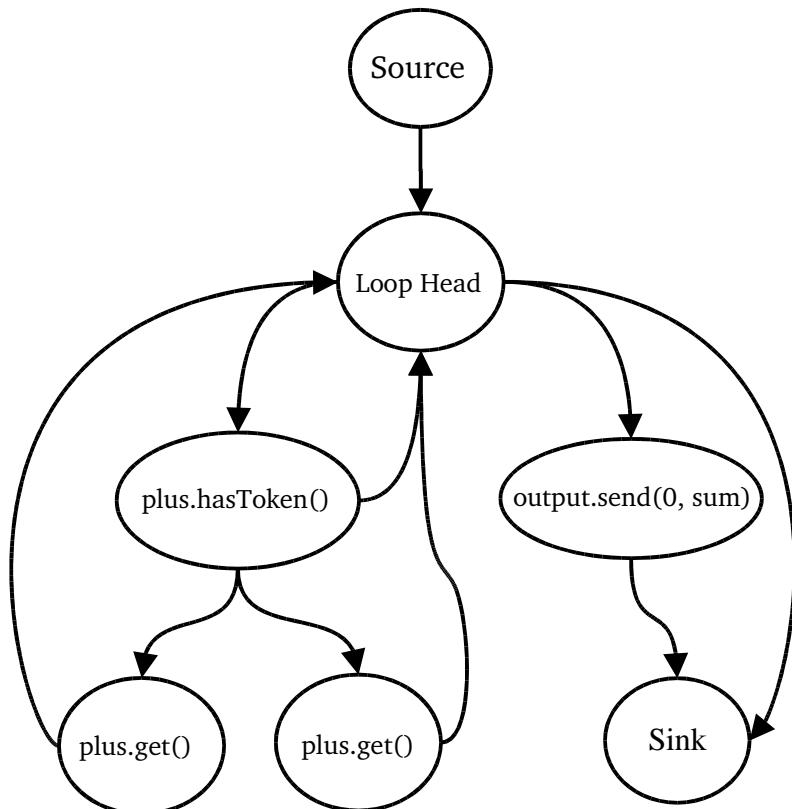


Figura 40 Grafo de métodos para o código da Figura 38.

Assim como um GFC, todo GM possui pelo menos dois nós: o nó fonte (*Source*) e no nó destino (*Sink*). Cada método do ambiente Ptolemy II utilizado é representado por um nó próprio. Na Figura 40, há quatro nós deste tipo. Para cada ocorrência de uma estrutura de iteração da linguagem Java, há um nó específico (*Loop Head*) no GM. Este nó é utilizado para determinar os nós que estão contidos no corpo da estrutura, e representar os caminhos de execução. Quando possível, o nó que representa um comando de iteração é anotado com a quantidade de repetições. Note a presença de uma aresta entre os nós *Loop Head* e *Sink*. Ela é devido ao comando *if* (linha 14 na Figura 38).

Os nós do GFC, e consequentemente do GM, são construídos a partir das árvores sintáticas do código Java, e não de uma representação interna baseada em instruções genéricas. Desta forma, cada nó apresenta uma relação direta com o código fonte de um ator. Isto é de extrema importância, pois o objetivo da ferramenta, além de determinar a validade de um ator, é gerar um conjunto de mensagens para o desenvolvedor. Desta forma, é possível produzir mensagens detalhadas e, em caso de erros, indicar precisamente quais foram as construções que

violaram as regras de um modelo computacional.

As estruturas construídas durante esta etapa são armazenadas em repositórios de dados, uma vez que uma classe, um ator atômico ou um ator hierárquico podem ser reutilizados em outras especificações.

4.5.1.2 Fase de Pré-processamento

Uma vez construídas as estruturas de dados, inicia-se o processo de análise propriamente dito. A primeira tarefa desta fase é determinar se algum ator atômico é específico a um modelo computacional⁴⁰. Em seguida, determina-se se o desenvolvedor especificou a lista de modelos computacionais válidos e inválidos para o ator, através de parâmetros associados aos atores.

Etapa 1: Busca por métodos específicos a um modelo computacional

Um ator atômico é específico a um modelo quando em seu código fonte forem utilizadas construções que dependam de classes ou interfaces particulares a um modelo computacional ou quando métodos específicos forem empregados.

Construções:

As construções que indicam o emprego de um modelo computacional em particular são:

1. Presença de um comando *getDirector()* com conversão de tipo para o diretor de um determinado modelo;
2. Alguma classe do ator implementa as interfaces *SequenceActor* e/ou *TimedActor*;
3. Alguma classe do ator implementa as interfaces *CTDynamicActor* e/ou *CTStatefulActor* ou deriva as classes *CTBaseIntegrator* ou *CTCompositeActor*;
4. Alguma classe do ator deriva a classe *DEActor*;

⁴⁰ Na fase de pré-processamento, estou considerando seis modelos computacionais: SDF, SR, DE, CT, CSP e PN.

5. Alguma classe do ator deriva a classe *CSPActor*;
6. O ator instancia objetos das classes *ConditionalBranch*, *ConditionalBranchController*, *ConditionalSend* e *ConditionalReceive*;
7. Presença do atributo *nonStrictMarker*.

O primeiro caso indica que o ator espera encontrar um diretor particular a um modelo computacional. Desta forma, não é possível o emprego de qualquer outro modelo.

O segundo caso não especifica um modelo em particular, mas sim um tipo de semântica, eliminando modelos que não a suportam. A interface *SequenceActor* indica que o ator espera que as entradas e saídas sejam sinais formados por uma seqüência de dados, onde a ordem dos dados na seqüência é relevante. A interface *TimedActor* indica que o ator opera sobre sinais onde os dados estão associados a um valor de um modelo de tempo.

Um ator que implementa as interfaces *CTDynamicActor* e/ou *CTStatefulActor*, ou deriva as classes *CTBaseIntegrator* ou *CTCompositeActor*, indica que ele é específico ao modelo CT, pois tais interfaces e classes são próprias deste modelo.

Um ator deriva a classe *DEActor* para utilizar portos de entrada e métodos específicos ao modelo DE. Da mesma forma, um ator deriva a classe *CSPActor* para utilizar comandos particulares ao modelo CSP.

As classes *ConditionalBranch*, *ConditionalBranchController*, *ConditionalSend* e *ConditionalReceive* são específicas ao modelo CSP e implementam a comunicação condicional.

O atributo *nonStrictMarker* identifica um ator como sendo do tipo *NonStrict*. Este atributo só é empregado no modelo SR.

Métodos específicos:

Os modelos SR, PN e CT não possuem métodos específicos que podem ser utilizados no código fonte de um ator atômico. Para os demais, é necessário percorrer os GFC em busca de tais métodos. São eles:

- SDF: Métodos `getTokenConsumptionRate()`, `getTokenProductionRate()`, `getTokenInitProduction()`, `setTokenConsumptionRate()`, `setTokenProductionRate()`, `setTokenInitProduction()`. Todos estes métodos pertencem a classe `SDFIOPort`;
- DE: As versões dos métodos de comunicação `broadcast()` e `send()` que utilizam um parâmetro para indicar o *timestamp* do dado produzido, e os métodos `delayTo()`, `getDelayedPorts()`. Todos estes métodos pertencem a classe `DEIOPort`;
- CSP: Métodos `getConditionalBranchController()` e `delay()`, ambos pertencentes a classe `CSPActor`.

Ao final da primeira etapa de pré-processamento, os atores atômicos são divididos em específicos e genéricos. As topologias que contenham instâncias de um ator específico, são declaradas específicas a esse modelo. A partir deste etapa, a ferramenta só manipula os atores genéricos e atores específicos aos modelos SDF e SR.

Etapa 2: Especificação da lista de modelos válidos e inválidos

A segunda etapa vasculha os parâmetros de cada ator a procura dos parâmetros `invalidFor` e `validFor`. O primeiro contem uma lista de modelos computacionais inválidos para o ator, enquanto que o segundo uma lista de modelos computacionais válidos. Utilizando estes parâmetros, o desenvolvedor pode reduzir o trabalho da ferramenta para casos onde tenha certeza da validade ou não de um ator.

4.5.1.3 Simplificação de código

Os algoritmos de validação descritos neste capítulo se baseiam na análise estática dos atores. Desta forma, para atores atômicos, a presença de regiões de código não utilizadas pode afetar o resultado da análise. Antes de qualquer verificação, deve-se simplificar o código fonte, eliminando tais regiões. Esta é uma

operação tradicional implementada em compiladores [APP98]. Mais especificamente, a etapa de simplificação é composta dos seguintes algoritmos:

1. Propagação de constantes;
2. Propagação de cópias de valores;
3. Eliminação de código morto.

Os três algoritmos são implementados sobre uma modificação do GFC denominada de *Static Single Assignment Form* (SSA) [APP98]. Através desta representação, é possível determinar as cadeias de definição-uso de cada variável. A representação SSA garante que o local da definição é único. As cadeias de definição-uso, além de utilizadas na simplificação do código, são também empregadas pelos algoritmos de validação do modelo SR.

Dependendo do modelo computacional, alguns métodos do ambiente Ptolemy II podem ser substituídos por constantes, ou simplesmente removidos. Para o modelo SR, nenhuma substituição de método pode ser feita. Para o modelo SDF, é possível executar as seguintes alterações:

- O método *hasToken()* pode ser substituído pelo valor lógico verdade, já que o escalonamento irá garantir a quantidade necessária de dados;
- O método *hasRoom()* pode ser substituído pelo valor lógico verdade, uma vez que o escalonador calcula o tamanho adequado para as filas de comunicação;
- O método *isKnown()* pode ser substituído pelo valor lógico verdade, pois o estado de um porto é sempre conhecido no modelo SDF;
- O método *sendClear()* e *broadcastClear()* podem ser removidos pois não possuem implementação do modelo SDF.

Todas as alterações sobre um GFC efetuadas durante esta etapa são transmitidas para o respectivo GM. A remoção de um nó ou uma aresta de um GFC implica em examinar se algum nó ou aresta do GM deve ser removido. Para as alterações específicas ao modelo SDF, nenhuma alteração é feita sobre o GM, mas

anotações são adicionadas. Por exemplo, no grafo da Figura 40, existem duas arestas a partir do nó com o método *hasToken()*, cada uma com destino a um nó com o método *get()*. Uma delas é anotada como não existente para o modelo SDF.

4.5.1.4 Validações independentes de um Modelo Computacional

A última etapa antes da execução dos algoritmos de validação para cada modelo computacional detecta situações inválidas, independentes de um modelo computacional. Verifico a ocorrência das seguintes construções:

- Não utilização de portos de E/S declarados e inicializados;
- Não utilização de parâmetros declarados e inicializados;
- Verificação dos campos *channelindex*⁴¹, determinando quantos canais o projetista está assumindo para um porto. Verificar se tais conexões existem.

A fim de ser possível a análise de um ator atômico, três restrições são impostas sobre seu código fonte. Atores que não satisfaçam estas restrições, são rejeitados pela ferramenta. São elas:

1. Os métodos de comunicação não podem aparecer em comandos de iteração não contáveis;
2. O parâmetro *channelindex* de um método de comunicação deve ser uma constante;
3. O parâmetro *vectorLength*⁴² de um método de comunicação que manipula vetores deve ser uma constante.

Um comando de iteração é dito contável quando for possível determinar estaticamente o número de iterações que ele executa, e não existir nenhum comando dentro do corpo do laço que termine sua execução antes de tal número. A razão para as três restrições é permitir o cálculo de quantos dados um determinado comando de

⁴¹ Trata-se de um parâmetro (número inteiro) presente nos métodos de comunicação, que identifica qual o canal do dado porto a que o comando de comunicação se refere.

⁴² Trata-se de um parâmetro (número inteiro) que para as versões dos comandos de comunicação que manipulam vetores, informa o tamanho do vetor.

comunicação produz ou consome ao ser executado.

4.5.2 Validação para o modelo SDF

4.5.2.1 Validação de atores atômicos

O objetivo da análise de um ator atômico perante as regras do modelo SDF é encontrar a taxa de amostragem para cada porto do ator. Desenvolvi um algoritmo para calcular o valor máximo e mínimo de cada taxa de amostragem. O cálculo exato é uma tarefa complexa, pois dependendo dos caminhos de execução do GFC, vários valores podem ser obtidos. Armazenar e processar estes valores resulta em um algoritmo pouco eficiente. Para determinar se o ator é válido, basta verificar se o valor máximo e mínimo para as taxas de amostragem são iguais.

Para cada porto de E/S, é associado um valor mínimo e máximo. Dependendo do tipo de porto, este valor representa consumo ou produção de dados. No caso de um porto bidirecional, dois conjuntos são criados. A entrada para o algoritmo deve ser o GM obtido conectando-se o nó *Sink* do GM do método *prefire()* ao nó *Source* do GM do método *fire()*, e efetuando a mesma conexão entre o GM do método *fire()* e *postfire()*. Temos o algoritmo para o cálculo das taxas mínima e máxima:

Algoritmo 4.1 – Taxas de amostragem.

Entrada: GM conectado seqüencialmente.

Saída: para cada porto de E/S, a taxa mínima e máxima de produção e/ou consumo.

1. Obtenha uma lista (L) dos nós do grafo, ordenados topologicamente desconsiderando as arestas de retorno⁴³ (*back edge*);
2. Obtenha o próximo nó de L. Caso não haja, vá para 6;
3. Calcule quantos dados o nó produz ou consome. Quando o nó estiver em um corpo de um laço, multiplique pelo número de iterações do laço;

⁴³ Uma aresta de retorno é aquela onde o nó origem é dominado pelo nó destino. É dito que um no A domina um nó B, quando todo caminho a partir do nó *Source* para o nó B contém o nó A.

4. Calcule o valor mínimo entre os valores mínimos obtidos de todos os nós conectados ao nó sendo processado. Não considere os nós conectados ao nó sendo processado através de arestas de retorno. Calcule o valor máximo de maneira similar;
5. Caso o nó sendo processado seja um início de laço ou uma chamada de método, execute o algoritmo recursivamente para o corpo do laço ou grafo associado ao método, respectivamente. Ao retornar, atualize os valores mínimos e máximos considerando somente as arestas de retorno.
6. Incremente os valores mínimo e máximo adicionando os dados produzidos ou consumidos pelo nó sendo processado. Vá para o passo 2;
7. Fim.

O resultado do algoritmo 4.1 é obtido observando-se os valores mínimo e máximo, para cada porto de E/S, no nó *Sink* do GM. O passo 3 do algoritmo determina qual é o porto sendo utilizado pelo nó e se os valores máximo e mínimo são de consumo ou produção de dados.

O algoritmo 4.1, bem como outras técnicas que utilizo no processo de análise, estão baseados na análise dos caminhos de um GFC⁴⁴. Como a análise é estática, considero que todos os caminhos do GFC são executáveis. Na prática, isto não ocorre. Como consequência, os resultados produzidos pelo algoritmo são conservativos.

Embora o problema de determinar se um caminho qualquer de um GFC é executável seja não computável, existem algoritmos capazes de eliminar caminhos não executáveis em determinadas situações [BOD97]. Assim, a seguinte pergunta pode ser feita: é possível que a eliminação de caminhos inexistentes do GFC altere a resposta da análise de um ator atômico? Dois cenários devem ser considerados:

Caso 1: Um ator identificado como inválido passa a ser válido;

Caso 2: Um ator identificado como válido passa a ser inválido.

No primeiro caso, o emprego de um algoritmo que eliminate caminhos não existentes apenas tornaria o resultado da análise mais precisa. Entretanto, o segundo caso é potencialmente catastrófico, pois um escalonamento inválido seria obtido.

44 Um GM é derivado do GFC.

Proposição: O caso 1 pode ocorrer e o caso 2 nunca ocorre

Considere a situação genérica em um GFC apresentando na Figura 41, onde existem N caminhos entre um nó A qualquer e um nó B qualquer, passando por N subgrafos (G_1, G_2, \dots, G_N).

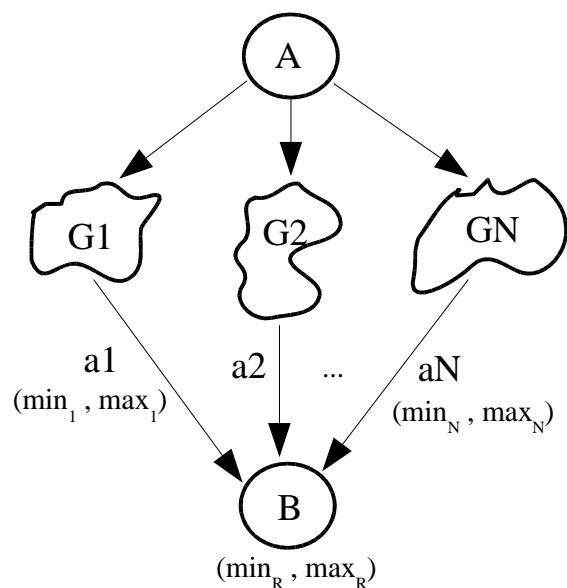


Figura 41 Conexão genérica entre nós do GFC.

A cada nó X do GFC e para cada porto P do ator está associado o par $(\min, \max)_P$, onde \min (\max) é o valor mínimo (máximo) de dados manipulados⁴⁵ pelo porto P até o nó X, obtido pelo algoritmo 4.1. No decorrer desta demonstração, a fim de torná-la mais clara, assumo que só há um porto sendo manipulado. O par (\min_R, \max_R) denota os valores obtidos para o nó B. Por definição, temos a seguinte relação: $\min \leq \max$.

Um algoritmo de eliminação de caminhos não existentes pode eliminar: (a) nenhum, (b) alguns, ou (c) todos os caminhos entre os nós A e B. O caso (a) é trivial. O caso (c) não é relevante, pois só ocorre quando o código fonte do ator não for um código Java correto.

Seja Alg41 o algoritmo 4.1 e Alg41-Elm o algoritmo que executa como

45 Manipulado significa consumido ou produzido.

primeiro passo a eliminação de caminhos inexistentes para, em seguida, executar o algoritmo 4.1.

Para o algoritmo Alg41 temos:

$\min_R = \text{menor valor entre todos os } \min_N.$

$\max_R = \text{maior valor entre todos os } \max_N.$

Para o algoritmo Alg41-Elm temos:

$\min'_R = \text{menor valor entre os restantes } \min_N.$

$\max'_R = \text{maior valor entre os restantes } \max_N.$

Pela definição do cálculo de \min_R e \max_R temos as seguintes relações:

$$(1) \min'_R \geq \min_R,$$

$$(2) \max'_R \leq \max_R.$$

Comparando a mudança das respostas entre os algoritmos Alg41 e Alg41-Elm, temos 4 possíveis situações:

A) $\min_R = \max_R$ torna-se $\min'_R < \max'_R$

B) $\min_R < \max_R$ torna-se $\min'_R < \max'_R$

C) $\min_R = \max_R$ torna-se $\min'_R = \max'_R$

D) $\min_R < \max_R$ torna-se $\min'_R = \max'_R$

Para cada caso, devemos considerar três situações: (s1) apenas o valor de \min_R mudou; (s2) apenas o valor de \max_R mudou; (s3) ou ambos os valores mudaram.

Caso A:

(s1): impossível, devido a 1.

(s2): impossível, devido a 2.

(s3): impossível, devido a **1** e **2**.

Caso B: Todas as três situações são possíveis.

Caso C:

(s1): impossível.

(s2): impossível.

(s3): impossível, devido a **1** e **2**.

Caso D: Todas as três situações são possíveis.

Obviamente, o caso A nunca ocorre. Logo um ator nunca é identificado erroneamente como válido. Pelo caso B, a eliminação de caminhos não altera o resultado final, pois o ator continua sendo classificado como inválido. O caso C só é permitido quando os valores de min e max não são alterados, ou seja, as taxas de amostragens são obtidas com precisão. No caso D, a eliminação de arestas contribui para identificar mais precisamente se um ator é válido ou não.

C.Q.D.

4.5.2.2 Validação de uma topologia

Uma vez determinadas todas as taxas de amostragem, a topologia pode ser validada. De acordo com as regras que enunciei no item 4.2.1, a topologia deve ser passível de escalonamento. O escalonador implementado no ambiente Ptolemy II é capaz de realizar esta tarefa. Quando uma topologia for inválida, ela é rejeitada pelo escalonador e uma mensagem de erro é apresentada. Isso pode ocorrer devido a duas situações: 1) a matriz de taxas de amostragem é inconsistente; 2) existem ciclos que não possuem a quantidade suficiente de dados iniciais. No primeiro caso, nada pode ser feito.

O segundo caso envolve a detecção de ciclos e o cálculo de quantos dados iniciais são necessários. Seria interessante construir um algoritmo para detectar automaticamente tais situações. Entretanto, esta tarefa não é trivial, pois em um

ciclo, atrasos podem ser introduzidos de diversas maneiras. Considere o exemplo da Figura 42.

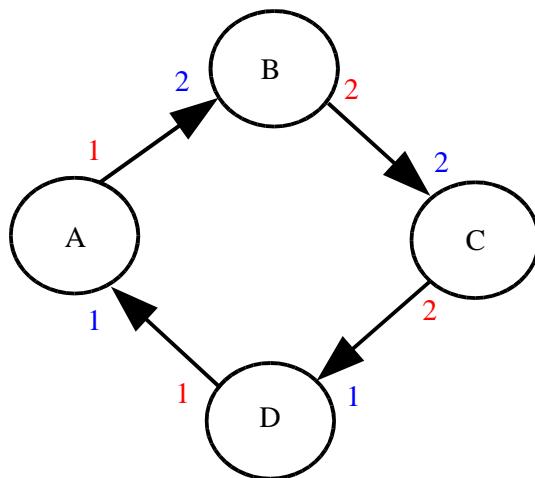


Figura 42 Exemplo de ciclo em uma topologia SDF.

O número de repetições para este grafo é: 2 A, 2 D, 1 B, 1 C. Assim, dentre outras opções, podemos distribuir elementos de atraso da seguinte maneira: 2 unidades de atraso na aresta CD, 2 unidades de atraso na aresta DA. 1 unidades de atraso na aresta CD, 1 na aresta DA, 1 unidades de atraso na aresta DA, 1 na aresta AB.

A solução mais adequada, que capture o comportamento desejado, deve ser escolhida pelo desenvolvedor. Entretanto, a ferramenta pode auxiliar detectando os ciclos com quantidade insuficiente de atraso.

Em algum momento, algum ator do ciclo deve ser ativado pela primeira vez. Isso significa que todas suas arestas devem apresentar a quantidade necessária de dados, implicando que a aresta pertencente ao ciclo deve ter algum dado inicial. Desta forma, uma condição necessária, mas não suficiente é que dado um ciclo, pelo menos uma aresta deve ter um atraso maior ou igual à taxa de consumo do respectivo porto, na respectiva aresta. Um caso que mostra que esta condição não é suficiente pode ser observado no exemplo da Figura 42: um atraso de 1 unidade na aresta CD não torna a topologia escalonável.

Utilizo o seguinte algoritmo para validação de topologias no modelo SDF:

Algoritmo 4.2 – Validação de uma topologia no modelo SDF

Entrada: Descrição de uma topologia: atores (com taxas de amostragem) e relações.

Saída: Grafo escalonado.

1. Encontre um escalonamento para a topologia, utilizando o ambiente Ptolemy II; Caso encontre, vá para 4;
2. Determine todos os ciclos da topologia da especificação;
3. Para cada ciclo, verifique se a condição de atraso está sendo satisfeita e informe o desenvolvedor em caso de falha;
4. FIM.

4.5.3 Validação para o modelo SR

4.5.3.1 Validação de atores atômicos

Um ator atômico para ser válido no modelo SR deve respeitar as duas condições enumeradas no item 4.2.2: (1) implementar uma função monotônica; (2) produzir no máximo um dado por porto de saída em um dado instante.

É possível verificar que um ator não produza mais que um dado por porto de saída através do algoritmo do cálculo das taxas de produção/consumo desenvolvido para o modelo SDF. Para o modelo SR, os pares aceitáveis de valor mínimo e máximo para um porto de saída são: (0,1) ou (1,1). A única modificação necessária com relação à versão deste algoritmo no modelo SDF é a presença dos métodos *sendClear()* e *broadcastClear()*. Estes métodos devem ser considerados como a produção de um dado, uma vez que torna definido o estado do respectivo porto. Como no modelo SDF, a resposta desse algoritmo também é afetada pela presença de caminhos não executáveis. Pela demonstração do caso SDF, é possível que um ator identificado como não SR na verdade é SR, por exemplo: uma resposta (0, 2) que na verdade é (0, 1). Novamente, uma etapa para eliminar caminhos não executáveis irá aumentar a precisão da resposta da ferramenta.

Determinar se um ator atômico implementa uma função monotônica analisando apenas seu código fonte é uma tarefa complexa. Isto acontece porque é

necessário determinar todas as condições em que um porto de saída tem seu valor definido. A definição de um porto pode depender do estado definido ou não de portos de entrada e do valor desses portos quando definidos. Para um ator do tipo *NonStrict*, deve-se levar em conta mais de um ativamento. A Figura 43 apresenta o método *fire()* do ator *NonStrictLogicFunction*, encontrado na biblioteca de atores do ambiente Ptolemy II. Este ator é específico ao modelo SR.

```

1- public void fire() throws IllegalActionException {
2-     BooleanToken value = null;
3-     BooleanToken in = null;
4-     for (int i = 0; i < input.getWidth(); i++) {
5-         if (input.isKnown(i)) {
6-             if (input.hasToken(i)) {
7-                 in = (BooleanToken)(input.get(i));
8-                 if (in != null) value = _updateFunction(in, value);
9-             }
10-        }
11-    }
12-
13-    if (value == null) {
14-        if (input.isKnown()) output.sendClear(0);
15-    } else {
16-        value = _nullifyIncompleteResults(value);
17-    }
18-    if (value != null) {
19-        if (_negate) value = value.not();
20-        output.send(0, (BooleanToken)value);
21-    }
22- }
23- private BooleanToken _nullifyIncompleteResults(
24-     BooleanToken inValue) throws IllegalActionException {
25-     BooleanToken outValue = inValue;;
26-     if (!input.isKnown()) {
27-         switch(_function) {
28-             case _AND:
29-                 if (inValue.booleanValue()) outValue = null;
30-                 break;
31-             case _OR:
32-                 if (!inValue.booleanValue()) outValue = null;
33-                 break;
34-             case _XOR:
35-                 outValue = null;
36-                 break;
37-             default:
38-         }
39-     }
40-     return outValue;
41- }
```

Figura 43 Método *fire()* do ator *NonStrictLogicFunction*.

O ator da Figura 43 implementa a função lógica, a ser determinada por dois parâmetros: `_function` (linha 27) e `_negate` (linha 19). O primeiro seleciona entre as funções lógicas **e**, **ou** ou **ou-exclusivo**. O segundo indica se o valor computado deve ser invertido ou não. Dependendo da função escolhida, não é necessário o estado definido de todos os portos de entrada, isto é, o ator é do tipo *NonStrict*. Por exemplo, se o método `_nullifyIncompleteResults()` (linha 23) receber um valor lógico falso e estiver sendo computado uma função lógica **e**, então a saída será definida. Este ator implementa uma função monotônica, pois é garantido que mesmo com a definição de novos portos de entrada o valor do porto de saída não seria alterado.

A fim de possibilitar o teste de monotonicidade de um ator automaticamente, impus a seguinte restrição a um ator atômico no modelo SR:

Um porto de saída só pode ser definido quando todos os portos de entrada da qual ele depende tiverem seus estados conhecidos.

Seja **c** um caminho do CFG com origem no nó *Source* e destino no nó onde ocorre uma definição do porto de saída. Um porto de saída **S** depende de um porto de entrada **E** em duas situações:

1. O cômputo do dado sendo produzido em **S** utiliza o valor de **E**;
2. A execução do caminho **c** contém pelo menos uma condição cuja resolução depende de um estado definido de **E**.

Na Figura 37, o cálculo do valor (*sum*) enviado pelo porto *output* utiliza o valor presente no porto *plus*. Na Figura 43, a definição do porto de saída *output* como ausente (linha 14) depende de uma condição que testa o estado do porto de entrada *input*.

Dante deste modelo SR mais restrito, todo ator é monotônico por construção, pois por definição, um porto de saída tem seu estado definido apenas quando todos os portos de entradas utilizados não mudarem de estado. Não é permitido a definição do estado de um porto de saída com base no não conhecimento do estado de um porto de entrada, isto é, uma definição que dependa do valor lógico falso do

método `isKnown()`. Assim como as taxas de amostragem, a dependência é uma característica associada a um porto, independentemente da quantidade de canais.

Note que um ator só é ativado quando for possível definir o estado de algum porto de saída. Não há mais a necessidade de classificar os atores atômicos em *Strict* e *NonStrict*. Isto é importante, pois facilita a concepção de atores polimórficos.

Um ator pode ser ativado mais de uma vez durante um instante, pois diferentes portos de saída de um ator podem depender de conjuntos distintos de portos de entrada. A fim de evitar a redefinição de um porto de saída, o método `isKnown()` deve ser adicionado antes de todo método que define um porto de saída. Esta observação induz a uma estrutura que um ator atômico deve ter no modelo SR. A Figura 44 ilustra tal estrutura.

```

1- IOPort in1, in2, in3, out1, out2;
2-
3- public void fire() throws IllegalActionException {
4-
5-     if(!out1.isKnown()) {
6-         if(in1.isKnown() && in3.isKnown() ) {
7-             ...
8-         }
9-     }
10-    if(!out2.isKnown()) {
11-        if(in2.isKnown() && in3.isKnown() ) {
12-            ...
13-        }
14-    }
15-    if(in1.isKnown() && in3.isKnown() && !out1.isKnown() ) {
16-        out1.broadcastClear();
17-    }
18-    if(in2.isKnown() && in3.isKnown() && !out2.isKnown() ) {
19-        out2.broadcastClear();
20-    }
21- }
```

Figura 44Estrutura genérica de um ator atômico no modelo SR.

A estrutura exemplificada na Figura 44 possui três portos de entrada (`in1`, `in2` e `in3`) e dois portos de saída (`out1` e `out2`). O porto `out1` depende dos portos `in1` e `in3`. O porto `out2` depende dos portos `in2` e `in3`. Entre as linhas 5 e 6, é determinado se é possível definir o valor do porto `out1`. Analogamente para o porto `out2` nas linhas 10 e 11. Ao final do código do ator (linhas 15 a 20) é determinado se as condições de dependências foram satisfeitas, mas o estado do porto de saída

permanece indefinido. Neste caso, o valor do porto de saída deve ser definido como ausente.

Desenvolvi então o seguinte algoritmo de validação:

Algoritmo 4.3 – Validação de um ator atômico para o modelo SR.

Entrada: GM e CFG dos métodos *prefire()*, *fire()* e *postfire()*, conectados em seqüência⁴⁶.

Saída: o ator é válido ou não.

1. Calcule as taxas de produção de cada porto de saída. Caso algum porto apresenta um valor diferente de (0, 1) ou (1, 1), o ator é inválido e vá para 4;
2. Determine para cada porto de saída, a lista de portos de entrada da qual ele depende. Caso ele dependa do valor falso do método *isKnown()* para algum porto de entrada, o ator é inválido e vá para 4;
3. ⁴⁷Modifique o código do ator, com base nas dependências entre portos calculadas no passo 1, a fim de adequá-lo ao modelo SR;
4. FIM.

As dependências de um porto de saída podem ser obtidas percorrendo-se as cadeias de definição-uso, que foram criadas durante a etapa de simplificação do código fonte. Implementei o seguinte algoritmo para esta função:

Algoritmo 4.4 – Cálculo da dependência de um porto de saída de um ator atômico.

Entrada: GM e CFG do método a ser analisado.

Saída: para cada porto de saída, uma lista de portos de entrada da qual ele depende.

1. Execute o pseudo-código BuscaDependências(GM, CFG);
2. Fim.

⁴⁶ Assim como no modelo SDF, o nó *Sink* de um grafo é conectado ao nó *Source* de grafo seguinte na seqüência.

⁴⁷ Este passo do algoritmo não foi implementado na ferramenta FASI.

Pseudo-código BuscaDependências(GM, CFG)

```

1-     para todo nó n do GM faça
2-         se n for uma chamada de método que define um porto de saída out então
3-             para todo uso u de uma variável v em n faça
4-                 encontre o local da definição def do uso u
5-                 execute EncontreDependenciasDeDados(out, def)
6-             para toda expressão expr no caminho do CFG que termina em n faça
7-                 para todo uso u de uma variável v em expr faça
8-                     encontre o local da definição def do uso u
9-                     execute EncontreDependenciasDeDados(out, def)

```

Pseudo-código EncontreDependenciasDeDados(out, def)

```

10-    se def é uma chamada de método de um porto de entrada in então
11-        adicione IN a lista de dependência de out
12-    senão
13-        se def é uma chamada de método mc então
14-            seja GM2 o grafo de métodos de mc
15-            seja CFG2 o grafo de fluxo de controle de mc
16-            execute BuscaDependencias(GM2, CFG2)
17-        senão
18-            para todo uso de uma variável v2 na definição def faça
19-                encontre o local da definição def2 de v2
20-                execute EncontreDependenciasDeDados(out, def2)

```

O pseudo-código *BuscaDependencias()* verifica as duas condições de dependência de um porto de saída. Entre as linhas 2 e 5, é verificado se um dado produzido depende de algum porto de entrada. Isso é feito analisando-se todas as variáveis envolvidas. De maneira similar, entre as linhas 6 e 9, é determinado se alguma expressão em um caminho do CFG que alcança uma definição de porto de saída depende de algum porto de entrada. O pseudo-código *EncontreDependenciasDeDados()* é responsável por determinar se uma dada definição depende de um porto de entrada. Isto é feito nas linhas 10 e 11, onde verifica-se se a definição é uma chamada a um dos comandos do ambiente Ptolemy II.

Exemplo de validação

Considere o código do ator *NonStrictDelay* apresentado na Figura 23. Como em todos os caminhos possíveis do comando *if* (linhas 10 e 11) o estado do porto de saída *output* é definido, a taxa de amostragem para este porto é (1,1).

A seguir, o algoritmo 4.4 é aplicado sobre o código. Existem três locais da definição do estado do porto *output*: linhas 12, 14 e 17. O comando da linha 12 não contempla a produção de um dado. A execução deste comando depende de duas condições (linhas 10 e 11), ambas relacionadas à variável *_previousToken*. Como a definição desta variável não está neste método (e efetuada no método *postfire*), o algoritmo conclui que o comando da linha 12 não depende de nenhum porto de entrada. Para o comando da linha 17, o algoritmo executa os mesmos passos. Para o comando da linha 14, há a produção de um dado. Entretanto, este dado é o valor da variável *_previousToken*. Logo, o algoritmo conclui que o porto *output* não depende de nenhum porto de entrada.

O último passo da validação é a modificação do código do ator. A Figura 45 ilustra o código alterado após o cálculo das dependências.

```

1- public void fire() throws IllegalActionException {
2-
3-     if(input.isKnown(0)) {
4-         if(input.hasToken(0)) {
5-             _currentToken = input.get(0);
6-         } else {
7-             _currentToken = AbsentToken.ABSENT;
8-         }
9-
10-    if(!output.isKnown()) {
11-        if(_previousToken != null) {
12-            if(_previousToken == AbsentToken.ABSENT) {
13-                output.sendClear(0);
14-            } else {
15-                output.send(0, _previousToken);
16-            }
17-        } else {
18-            output.sendClear(0);
19-        }
20-    }
20-}

```

Figura 45 Método *fire()* do ator *NonStrictDelay* adequado ao modelo SR.

Apenas duas linhas (10 e 20) foram incluídas, uma vez que o porto de saída não depende de nenhum porto de entrada e em todos os caminhos possíveis, o estado do porto de saída é definido.

4.5.3.2 Validação de uma topologia

Como no modelo SDF, uma topologia válida deve ser possível de ser escalonada. Dada a restrição que impus ao modelo SR, o escalonamento pode ser computado trivialmente a partir de uma ordem topológica de um grafo (denominado de grafo de dependência). Cada nó do grafo representa um porto. Existe uma aresta entre dois nós, onde um nó representa um porto de entrada e outro nó representa um porto de saída, em duas situações:

1. Existe uma relação na topologia que contenha os dois portos;
2. O porto de entrada está na lista de dependências do porto de saída.

O grafo captura a dependência entre todos os portos da topologia. A fim de ser possível escalar a topologia, não pode haver ciclos no grafo, pois isto implica que haverá portos que permanecem sempre em um estado indefinido. Quando o grafo resultante for acíclico, qualquer ordem topológica produz um escalonamento para o sistema.

Embora a versão do modelo SR que utilizei é mais restrita, ela continua sendo de grande utilidade. Em particular, a linguagem Lustre [HAL91] apresenta o mesmo tipo de restrição, isto é, resolução de dependências permitindo somente a construção de topologias acíclicas. A linguagem Lustre é utilizada no ambiente de interação de atores Scade⁴⁸. Este ambiente é utilizado para projetos em escala industrial.

4.5.4 Implementação da estratégia de validação

Uma vez validados todos os atores atômicos e cada topologia separadamente, o próximo passo da estratégia de validação implica em determinar se a interação entre atores hierárquicos opacos é possível. A estratégia que propus no item 4.4

⁴⁸ www.estrel-technologies.com

induz naturalmente a uma análise do tipo *bottom-up*, isto é, processar inicialmente a topologias que não contenham nenhum ator hierárquico, para em seguida processar as topologias que contenham as já processadas, e assim sucessivamente, até a topologia de nível zero.

O resultado dessa análise é uma árvore, onde cada nó é um ator hierárquico, e existe uma conexão entre um nó pai a um nó filho quando a respectiva topologia do nó pai conter a do nó filho. A cada nó da árvore está associada uma lista de modelos computacionais que podem controlar a respectiva topologia.

Existem quatro combinações de modelos a serem validadas:

1. Ator hierárquico SDF contido em topologia SR;
2. Ator hierárquico SDF contido em topologia SDF;
3. Ator hierárquico SR contido em topologia SR;
4. Ator hierárquico SR contido em topologia SDF.

No primeiro caso, o ator hierárquico SDF deve apresentar taxas de amostragem igual a 1 em todos os seus portos. As taxas de amostragem de um ator hierárquico são obtidas propagando-se as taxas do portos de atores contidos na topologia desse ator, e conectados a seus portos de E/S. Entretanto, nem sempre isto é possível, pois um porto de E/S de uma topologia pode estar conectado a portos de atores com taxas de amostragem diferentes. Neste caso, a interação não é válida. Todos os portos de saída do ator hierárquico SDF dependem de todos seus portos de entrada.

O segundo caso procura inicialmente considerar o ator hierárquico como um ator atômico para a topologia que o contém. Entretanto, o modelo SDF não possui a propriedade de composição, isto é, embora uma topologia possa ser válida para o modelo SDF, isto não implica que o ator hierárquico obtido seja válido em outra topologia SDF. Desta forma, primeiramente a ferramenta procura determinar se a topologia que contém o ator hierárquico é escalonável, considerando-o como um ator atômico. Caso não seja possível, a ferramenta procura escalar a topologia, substituindo-se o ator hierárquico pela sua topologia. A Figura 46 apresenta um par de topologias para ilustrar este caso.

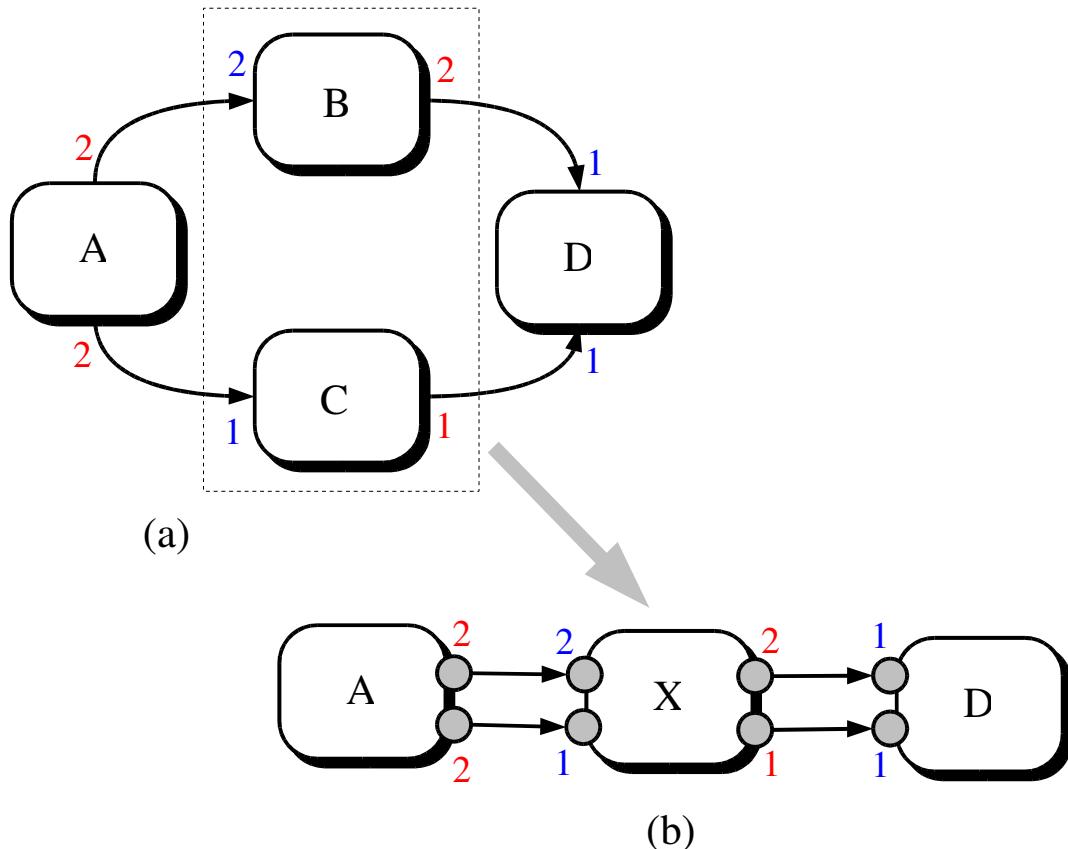


Figura 46 Exemplo de composição no modelo SDF.

A Figura 46 (a) possui duas topologias, uma composta por um ator hierárquico que contém os atores B e C, e outra que contém os atores atômicos A e D e o ator hierárquico. Na Figura 46 (b), o ator hierárquico é encarado como um ator atômico. Nesta topologia, as taxas de amostragem são inconsistentes. Já na Figura 46 (a), a topologia obtida substituindo o ator hierárquico pela sua topologia é escalonável (ABCCDD por exemplo).

No terceiro caso de combinação, o ator hierárquico SR é considerado como um ator atômico para a topologia que o contém. Ao contrário do modelo SDF, o modelo SR possui a propriedade de composição. As taxas de amostragem dos portos de E/S são obtidas a partir das taxas dos portos conectados na topologia do ator hierárquico. Como o conceito de dependência apresenta a propriedade de transitividade, a lista de dependência de um porto de saída do ator hierárquico é obtida determinando-se todos os portos de entrada do ator hierárquico que possuem um caminho no grafo de dependência, até o respectivo porto de saída.

No último caso de combinação de modelos, a única restrição é que o ator

hierárquico SR deve possuir taxas de amostragem iguais a 1 em todos os seus portos, tanto de entrada quanto de saída.

4.5.5 Resultados Experimentais

O primeiro experimento que conduzi verificou a capacidade da ferramenta FASI de analisar atores atômicos. O ambiente Ptolemy II possui uma biblioteca de atores, sendo que vários são polimórficos. Selecionei um conjunto destes atores, e verifiquei a validade de cada um para o modelo SDF e SR. A tabela 3 apresenta o resultado obtido. A coluna sucesso indica a quantidade de atores que a ferramenta foi capaz de analisar corretamente, indicando a validade ou não para os modelos em questão. A coluna falha indica atores que poderiam ser analisados, mas que a ferramenta foi incapaz de produzir uma solução precisa. A coluna impossível indica a quantidade de atores que não são possíveis de serem analisados estaticamente. Os dados são apresentados no formato SDF\SR, onde SDF indica o número para o modelo SDF, e SR o número para o modelo SR. Quando apenas um número for apresentado, isto significa que o resultado foi igual para os dois modelos.

Tabela 3 – Resultados da análise efetuada pela ferramenta FASI em atores atômicos.

Biblioteca	Total de Atores	Sucesso	Falha	Impossível
Array	6	4	0	2
Conversion	17	16	1	0
Flow Control	16	5\7	5\4	6\5
Logic	5	3\5	2\0	0
Math	16	12\16	4\0	0
Random	4	4	0	0
Signal Processing	19	8\17	10\2	1\0
Sink	12	12	0	0
Source	11	10\11	1\0	0
Total:	106	74\92	23\7	9\7

Após analisar os atores classificados como falha da ferramenta FASI,

determinei que ocorreram apenas duas situações para esta classificação:

1. Métodos de comunicação que dependem de um valor de um parâmetro do ator;
2. Laços cuja iteração é condicionada ao número de conexões de um porto.

Em alguns atores, a quantidade de dados produzidos ou consumidos eram especificadas ou deduzidas a partir de valores de parâmetros do ator. A Figura 47 apresenta um trecho de código do ator *SequenceToArray* da biblioteca Array que exemplifica este caso.

```

1- public Parameter arrayLength;
2- ...
3- public void fire() throws IllegalActionException {
4-     super.fire();
5-     int length = ((IntToken)arrayLength.getToken()).intValue();
6-     Token[] valueArray = input.get(0, length);
7-
8-     output.send(0, new ArrayToken(valueArray));
9- }
```

Figura 47Trecho de código do ator SequenceToArray.

Na linha 8, um dado é produzido. Este dado é um vetor contendo todos os dados lidos pelo ator, no presente ativamento. A quantidade de dados lidos pelo método *get()* (linha 6) depende do valor do parâmetro *arrayLength* (linha 1 e 5).

Um exemplo do segundo caso pode ser observado na linha 4 da Figura 38. O código da Figura 38 viola a restrição de um valor constante para o parâmetro *channelindex*.

Uma solução para estas duas situações onde houve falha da ferramenta é fazer a análise para cada instância do ator, isto é, particularizar o código fonte do ator para cada uso em uma especificação executável. Para tal, uma cópia do conjunto de estruturas de dados criadas a partir do código fonte deve ser feita para cada instância do ator.

No primeiro caso, os parâmetros poderiam ser classificados em dois tipos: estáticos ou dinâmicos. Um parâmetro estático não teria seu valor alterado ao longo

da execução. Neste caso, seu valor pode ser considerado uma constante. Para cada instância do código do ator, os parâmetros estáticos poderiam ser substituídos por seus valores e a etapa de simplificação de código poderia propagar estes valores.

Da mesma forma que um parâmetro estático, o valor retornado pelo método *getWidth()* pode ser considerado uma constante⁴⁹. Para cada instância de um ator, este método indica quantas conexões e, consequentemente, canais existem em um porto. Assim, o valor dos campos *channelindex* dos métodos de comunicação e o número de interações dos laços podem ser substituídos por constantes.

Embora não implementei na ferramenta FASI a cópia das estruturas de dados para cada instância de um ator, nas análises de especificações que efetuei a fim de testar a ferramenta, quando necessário, efetuei a substituição de parâmetros e do método *getWidth()* por constantes manualmente.

Para os atores que foram classificados como de análise impossível, detectei apenas duas situações que acarretaram em tal conclusão:

1. O código do ator não respeita as restrições da ferramenta;
2. Os portos de E/S eram parâmetros.

No primeiro caso, o código do ator contém construções que não respeitam as restrições da ferramenta descritas no item 4.5.1.4.

O segundo caso foi identificado em alguns atores da biblioteca Flow Control. Neles, portos de E/S são criados através de parâmetros ao invés de declarados diretamente no código fonte do ator. A Figura 48 apresenta um trecho de código do ator *RecordAssembler* que exemplifica essa situação.

⁴⁹ A estratégia que proponho neste capítulo não contempla especificações que alteram topologias ao longo da execução.

```

1- public void fire() throws IllegalActionException {
2-     Object[] portArray = inputPortList().toArray();
3-     int size = portArray.length;
4-
5-     String[] labels = new String[size];
6-     Token[] values = new Token[size];
7-
8-     for (int i = 0; i < size; i++) {
9-         IOPort port = (IOPort)portArray[i];
10-        labels[i] = port.getName();
11-        values[i] = port.get(0);
12-    }
13-
14-    RecordToken result = new RecordToken(labels, values);
15-
16-    output.send(0, result);
17- }

```

Figura 48Trecho de código do ator RecordAssembler.

Os portos de entrada deste ator são obtidos na linha 2, através do método *inputPortList()*. Os dados dos portos são obtidos na linha 11.

Além de testar a ferramenta FASI analisando somente atores atômicos, efetuei alguns testes de especificações de pequeno porte. Testei a capacidade da ferramenta de detectar topologias específicas a um modelo utilizando exemplos disponíveis no ambiente Ptolemy II. Também analisei especificações compostas por atores polimórficos, mas desenvolvidas considerando-se um modelo computacional em particular. Por exemplo, a especificação das Figuras 21 e 22 foi criada para testar as características do modelo SR. A ferramenta FASI foi capaz de determinar que tal especificação é válida para o modelo SR (a presença dos atores *NonStrictDelay* evitam ciclos de dependência).

4.6 Considerações

Neste capítulo descrevi uma estratégia para análise automática de especificações executáveis a fim de determinar sua validade com respeito a um modelo computacional. Implementei essa estratégia em uma ferramenta de software para dois modelos de grande importância para o projeto de software embarcado.

A estratégia que propus apresenta algumas dificuldades para a sua implementação. O principal impecilho é que a validação de um modelo computacional envolve a resolução de problemas não computáveis ou intratáveis.

Por exemplo, determinar os caminhos executáveis de um código ou determinar a monotonicidade de um ator atômico. Além destes problemas, a implementação da ferramenta impõe algumas restrições ao código de um ator. Entretanto, os resultados obtidos no item 4.5.5 indicam a utilidade da ferramenta. De um total de 106 atores analisados, 74 foram corretamente analisados para o modelo SDF e 92 para o modelo SR. Este número poderia subir facilmente para 97 e 99 respectivamente, isto é, 92% dos atores analisados corretamente. Além disto, o tempo de execução necessário para efetuar as análises é bastante pequeno, em geral, menor que um minuto. No capítulo 5, demonstro a utilidade da estratégia em dois estudos de caso.

Embora os resultados que obtive sejam satisfatórios, é importante lembrar que várias melhorias podem ser efetuadas a ferramenta. Dentre elas:

- Implementar as modificações descritas no item 4.5.5;
- Contemplar outros modelos computacionais, como por exemplo PN e CSP;
- Associar a ferramenta de análise a uma de geração de código, tal que seja possível produzir, a partir da especificação dada para a ferramenta de análise, especificações específicas aos conjuntos de modelos computacionais válidos;
- Introdução de algoritmos para tornar a análise mais precisa, como por exemplo eliminação de caminhos não executáveis, algoritmos de propagação de constantes mais poderosos que o que utilizei e outras técnicas de simplificação de código.

CAPÍTULO 5 - ESTUDO DE CASOS

“Poucas coisas são mais difíceis de se conseguir que um bom exemplo.”

Mark Twain

5.1 Codificador de voz padrão GSM

A codificação de voz é empregada no campo da telefonia celular devido à limitação da banda disponível para a comunicação sem fio. Este tipo de técnica procura atingir um alto grau de compressão sem perder demasiadamente a qualidade do sinal de voz. Algoritmos eficientes para codificação de voz humana foram obtidos através do modelamento do processo de formação sonora nas cordas vocais humanas. Esses algoritmos foram inicialmente pesquisados pela companhia telefônica Nokia em conjunto com a universidade de Sherbrooke [JAR97].

O padrão de codificação de voz que utilizei neste estudo de caso é o *Enhanced Full Rate (EFR) speech transcoding* do instituto de padrões europeu (ETSI), sob o código GSM 06.60 [ETS96]. Estudei e implementei uma das etapas do processo de codificação, denominada de análise LPC. Nesta etapa, uma nova amostra do sinal de voz, denominada de pacote, composta por um conjunto de palavras de 13 bits⁵⁰, é analisada conjuntamente com uma fração do pacote recebido na amostragem anterior, a fim de produzir 5 dos 57 parâmetros de saída do codificador. A etapa de análise LPC também produz sinais internos que são usados em etapas posteriores do processo de codificação.

O padrão GSM 06.60 impõe uma restrição na taxa de transmissão do sinal codificado. O padrão especifica que o codificador deve processar um pacote a cada 20 ms. Desta forma, a taxa de entrada é de 104 kbits/s. O resultado da codificação produz um total de 244 bits por pacote, a uma taxa de 12.2 kbits/s.

5.1.1 Especificação

⁵⁰ Cada pacote é composto por 160 palavras de 13 bits.

O padrão GSM 06.60 dispõe de uma especificação escrita em língua inglesa, detalhando o processo de codificação e decodificação. Além deste documento, está disponível um código em linguagem de programação C (modelo de referência) que implementa os algoritmos para codificação e decodificação, onde as operações aritméticas apresentam precisão de bits [ET96b]. Utilizei este modelo de referência como base para o desenvolvimento e validação de uma especificação executável no ambiente Ptolemy II.

Uma análise detalhada do código do modelo de referência revelou a utilização de procedimentos para implementar os diversos blocos funcionais. A comunicação entre cada bloco é feita através de parâmetros e variáveis globais. Como o código é seqüencial, não existe necessidade de sincronização entre os procedimentos.

Embora o modelo de referência implemente corretamente os algoritmos de codificação e decodificação, o uso de variáveis globais para a comunicação entre procedimentos é uma técnica reconhecidamente desvantajosa, uma vez que torna a compreensão e modificação do modelo mais complicada, facilitando a introdução de erros. A Figura 49 apresenta de maneira resumida o código fonte para o procedimento de nível 0 do codificador.

```

1-      txdtx_ctrl = 3;
2-
3-      reset_flag = encoder_homing_frame_test (new_speech);
4-
5-      for (i = 0; i < L_FRAME; i++)
6-      {
7-          new_speech[i] = new_speech[i] & (Word16)0xffff8;
8-      }
9-      Pre_Process(new_speech, L_FRAME);
10-
11-      Coder_12k2(prm, syn, A_t, Aq_t);

```

Figura 49 Código fonte de nível 0 do codificador.

Cada novo pacote é armazenado em um bloco de memória determinado pelo ponteiro *new_speech*. Antes da análise LPC (linha 11), cada pacote é pré-processado (linhas 3 até 9) pelos procedimentos *encoder_homing_frame_test()* e *Pre_Process()*. Os vários sub-procedimentos da análise LPC estão implementados no procedimento

Coder_12k2(). Note que o ponteiro *new_speech* não é passado como um parâmetro do procedimento *Coder_12k2()*, pois a codificação é feita sobre uma janela composta por um pacote e meio. Desta forma, cada novo pacote recebido é armazenado na região indicada pelo ponteiro *new_speech*, mas o processamento é efetuado sobre uma região maior, que também inclui parte do pacote anterior. A Figura 50 mostra a atualização de ponteiros globais responsáveis por indicar tais regiões.

```

1- static Word16 old_speech[L_TOTAL];
2- static Word16 *p_window;
3- Word16 *new_speech;

4- new_speech = old_speech + L_TOTAL - L_FRAME;
5- p_window = old_speech + L_TOTAL - L_WINDOW;

```

Figura 50 Ponteiros para os pacotes do codificador.

Neste código, três constantes são empregadas: *L_TOTAL* é equivalente a dois pacotes (320 palavras); *L_FRAME* é equivalente a um pacote (160 palavras); *L_WINDOW* é equivalente a janela de processamento (240 palavras). A variável *old_speech* possui espaço de armazenamento para dois pacotes. O ponteiro *new_speech* aponta para a segunda metade desta variável (linha 4). O ponteiro *p_window* aponta para a janela onde a computação é efetuada (linha 5).

A etapa da codificação que implementei neste estudo de caso é a análise LPC. Ela é a primeira etapa da codificação. A Figura 51 apresenta o código fonte resumido desta etapa, contendo os procedimentos que compõe a análise LPC.

```

1- void
2- LPAnalysis(Word16 *A_t, Word16 *Aq_t, Word16 *txdtx_ctrl,
3-             Word16 dtx_mode, Word16 ana[]) {
4-     scal_acf = Autocorr(p_window_mid, M, r_h, r_l, window_160_80, 1);
5-     Lag_window(M, r_h, r_l);
6-     Levinson(r_h, r_l, &A_t[MP1], rc);
7-     Az_lsp(&A_t[MP1], lsp_mid, lsp_old);
8-
9-     scal_acf = Autocorr(p_window, M, r_h, r_l, window_232_8, 2);
10-    Lag_window(M, r_h, r_l);
11-    Levinson(r_h, r_l, &A_t[MP1 * 3], rc);
12-    Az_lsp(&A_t[MP1 * 3], lsp_new, lsp_mid);
13-
14-    if(dtx_mode == 1) {
15-        VAD_flag = vad_computation(r_h, r_l, scal_acf, rc, 1);
16-        tx_dtx(VAD_flag, txdtx_ctrl);
17-    }
18-    else {
19-        VAD_flag = 1;
20-        *txdtx_ctrl = TX_VAD_FLAG | TX_SP_FLAG;
21-    }
22-
23-    Q_plsf_5(lsp_mid, lsp_new, lsp_mid_q, lsp_new_q, ana,
24-              *txdtx_ctrl);
25-
26-    ana += 5;
27-
28-    Int_lpc2(lsp_old, lsp_mid, lsp_new, A_t);
29-    if((*txdtx_ctrl & TX_SP_FLAG) != 0) {
30-        Int_lpc(lsp_old_q, lsp_mid_q, lsp_new_q, Aq_t);
31-
32-        for(i = 0; i < M; i++) {
33-            lsp_old[i] = lsp_new[i];
34-            lsp_old_q[i] = lsp_new_q[i];
35-        }
36-    }
37-    else {
38-        for(i = 0; i < MP1; i++) {
39-            Aq_t[i] = A_t[i];
40-            Aq_t[i + MP1] = A_t[i + MP1];
41-            Aq_t[i + MP1 * 2] = A_t[i + MP1 * 2];
42-            Aq_t[i + MP1 * 3] = A_t[i + MP1 * 3];
43-        }
44-        for (i = 0; i < M; i++) {
45-            lsp_old[i] = lsp_new[i];
46-            lsp_old_q[i] = lsp_new[i];
47-        }
48-    }
}

```

Figura 51O procedimento implementando a análise LPC.

O procedimento produz como resultado os cinco primeiros parâmetros do codificador, que são armazenados no vetor *ana* (linha 3). Note na linha 26 o uso de aritmética de ponteiros para modificar a posição de memória que tal vetor aponta,

após o computo dos cinco parâmetros. O procedimento também calcula os vetores A_t e Aq_t (linha 2) que são utilizados em outras etapas da codificação e a palavra de controle $txdtx_ctrl$ (linha 2). O procedimento tem como valores de entrada a janela de processamento (p_window e p_window_mid) e uma variável de controle (dtx_mode).

5.1.2 Especificação executável

A tarefa de criar uma especificação baseada na interação de atores a partir do modelo de referência envolveu associar os diferentes procedimentos a atores, traduzindo a comunicação através de variáveis e parâmetros para uma baseada em troca de mensagens. Os atores obtidos são interconectados formando uma estrutura hierárquica. Essa estrutura foi determinada com base nas diferentes etapas de codificação, que pode ser decomposta em pré-processamento e análise LPC⁵¹.

A Figura 52 apresenta o ator hierárquico de nível 0 da especificação executável.

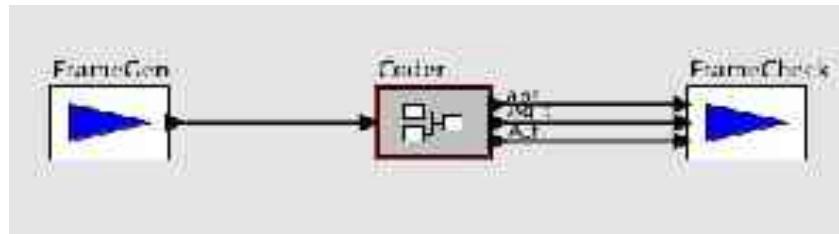


Figura 52 Ator hierárquico de nível 0 do codificador.

A topologia da Figura 52 é composta por dois atores atômicos, *FrameGen* e *FrameCheck* e um ator hierárquico, *Coder*. O ator *FrameGen* é responsável por produzir os pacotes do sinal de voz. O ator *Coder* implementa o codificador. O ator *FrameCheck* é utilizado para verificar se os valores produzidos pelo codificador são corretos.

Cada pacote é composto por 160 palavras de 13 bits, representando uma amostragem de 20ms do sinal de voz. Cada ator atômico que manipula um pacote,

⁵¹ O codificador apresenta outras etapas. Entretanto, elas não foram implementadas neste estudo de caso.

ou parte dele, representa-o através de um vetor do tipo *Short*. No ambiente Ptolemy II, esse tipo de dado pode ser encapsulado em um *token* do tipo *ObjectToken*. A Figura 53 mostra o método *fire()* do ator *FrameGen* da Figura 52. Note a construção de um novo *ObjectToken* na linha 6, tendo como valor associado um vetor de dimensão *L_FRAME*.

```

1- Short [] frame = new Short[Cnst.L_FRAME];
2- for(int i = 0;i < Cnst.L_FRAME;i++) {
3-   frame[i] = new Short((short)({_frames[_counter][i] & 0xffff8}));
4- }
5-
6- ObjectToken obj = new ObjectToken(frame);
7- output.broadcast(obj);

```

Figura 53 Método *fire()* do ator *FrameGen*.

A Figura 54 apresenta a topologia do ator *Coder*.

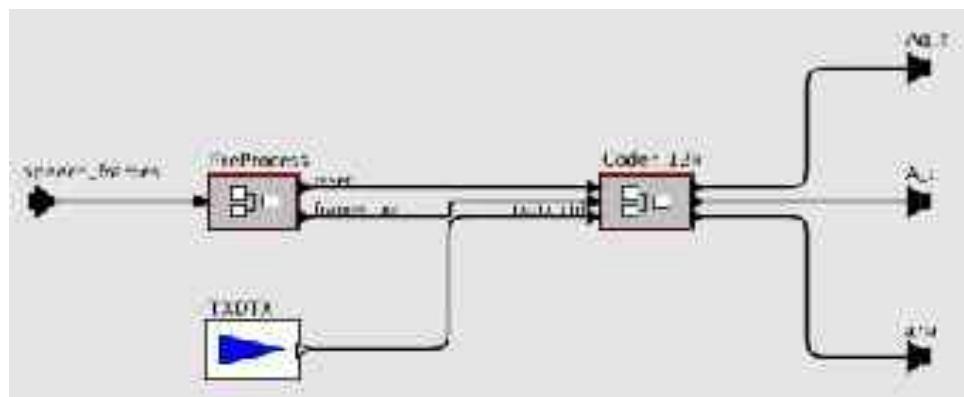


Figura 54 Topologia do ator *Coder*.

O ator *Coder* é composto pelos atores hierárquicos *PreProcess* e *Coder_12k*, e pelo ator atômico *TXDTX*. O ator *TXDTX* é uma instância do ator *Const* da biblioteca de atores polimórficos do ambiente Ptolemy II. Sua finalidade é produzir um valor constante a cada ativação. Na topologia da Figura 54, o ator *TXDTX* é responsável por produzir uma palavra de controle.

O ator *PreProcess* implementa o pré-processamento sobre um pacote. A Figura 55 apresenta sua topologia.

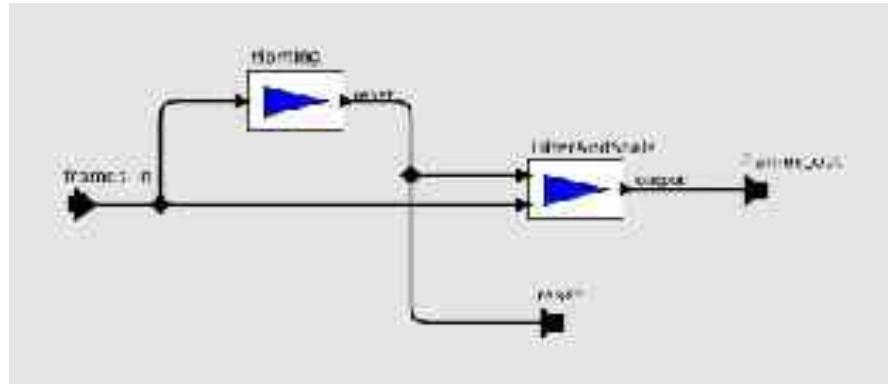


Figura 55 Topologia do ator *PreProcess*.

Todo novo pacote é submetido a dois atores atômicos na topologia *PreProcess*: *Homing* e *FilterAndScale*. O ator *Homing* é responsável por detectar uma condição de iniciação do codificador. Isto ocorre quando um pacote com um valor específico for recebido. O resultado é um sinal (*reset*) indicando a necessidade de reiniciação do codificador. O ator *Homing* implementa o procedimento da linha 3 da Figura 49.

O ator *FilterAndScale* implementa o procedimento da linha 9 da Figura 49. Trata-se de uma transformação operada sobre um pacote. Desta forma, este ator caracteriza-se por ser transformacional. A Figura 56 apresenta o método *fire()* desse ator.

```

1- if(reset.hasToken(0)) {
2-     BooleanToken bt = (BooleanToken) reset.get(0);
3-     if(bt.booleanValue()) {
4-         _init();
5-     }
6- }
7-
8- ObjectToken obj = (ObjectToken) input.get(0);
9- Short [] frame = (Short []) obj.getValue();
10-
11- frame = _filter_and_scale(frame);
12- output.broadcast(new ObjectToken(frame));

```

Figura 56 Método *fire()* do ator *FilterAndScale*.

Entre as linhas 1 e 6, é determinado a presença do sinal *reset*, e caso seu valor seja verdade, é efetuada uma reiniciação do estado do ator. Entre as linhas 8 e 9, um novo pacote é obtido. Na linha 11, o método que transforma o pacote é chamado. Na linha 12, o pacote transformado é enviado pelo porto *output*.

O ator *Coder_12k2* da Figura 54 é responsável por todas as demais etapas de codificação, em particular, a análise LPC. A Figura 57 apresenta sua topologia.

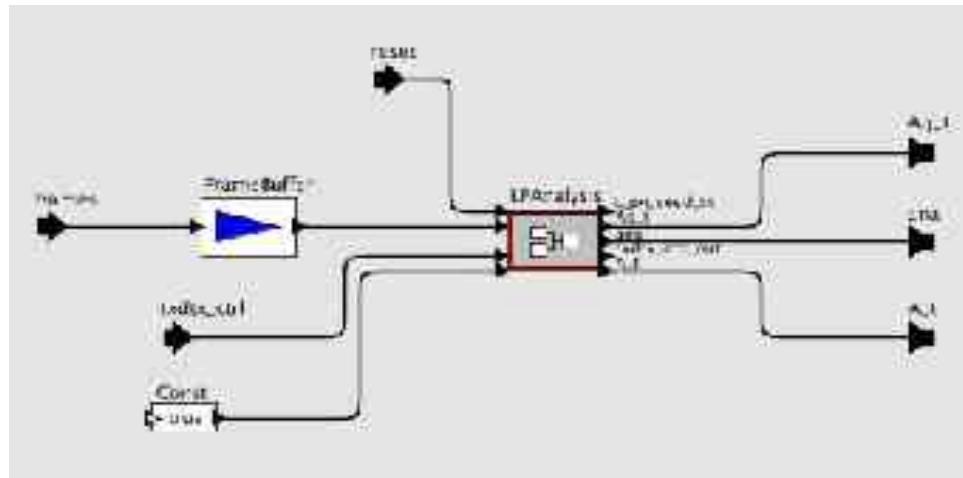


Figura 57 Topologia do ator *Coder_12k2*.

O ator *Coder_12k2* é composto por dois atores atômicos e um hierárquico. O ator *Const* é outra instância do ator homônimo da biblioteca de atores polimórficos. O ator *FrameBuffer* é capaz de armazenar dois pacotes, implementando o conceito de janela de pacotes (vide Figura 50). Para cada novo pacote recebido, este ator gera uma nova janela. Assim, seu comportamento também é transformacional. O ator *LPAanalysis* implementa a etapa de análise LPC. A Figura 58 apresenta sua topologia.

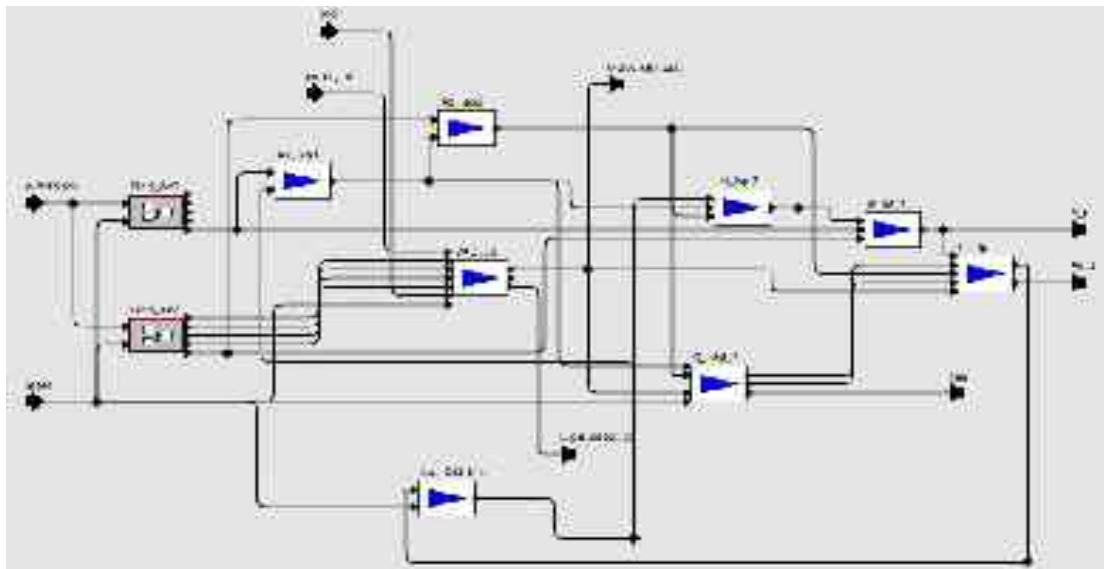


Figura 58 Topologia do ator LPAnalysis.

Os atores *Az_lsp1* e *Az_lsp2* são duas instâncias de um mesmo ator hierárquico, implementando as linhas 7 e 12 da Figura 51, respectivamente. O ator *VAD_Lp* corresponde as linhas 14 a 21. O ator *Q_plsf_5* corresponde ao procedimento da linha 23 e 24. O ator *Int_lpc2* implementa o procedimento da linha 28. O ator *JoinAt* é utilizado para construir o vetor *A_t* a partir de outros três vetores criados durante a análise LPC. O ator *Int_lpc* implementa as linhas 29 a 48. Este ator calcula o vetor *Aq_t*, outro resultado da análise LPC. O ator *Lsp_Old_Init* é responsável por armazenar o valor do sinal *lsp_old* calculado pelo ator *Int_lpc*. Os atores *Az_lsp1* e *Int_lpc2* necessitam do vetor *lsp_old* para produzir seus resultados. Isto provoca uma dependência cíclica acerca do valor do vetor *lsp_old*. Para resolver tal dependência, o ator *Lsp_Old_Init* gera um valor inicial para o vetor *lsp_old* independentemente do ator *Int_lpc*.

A Figura 59 apresenta a topologia do ator *Find_Az1*.

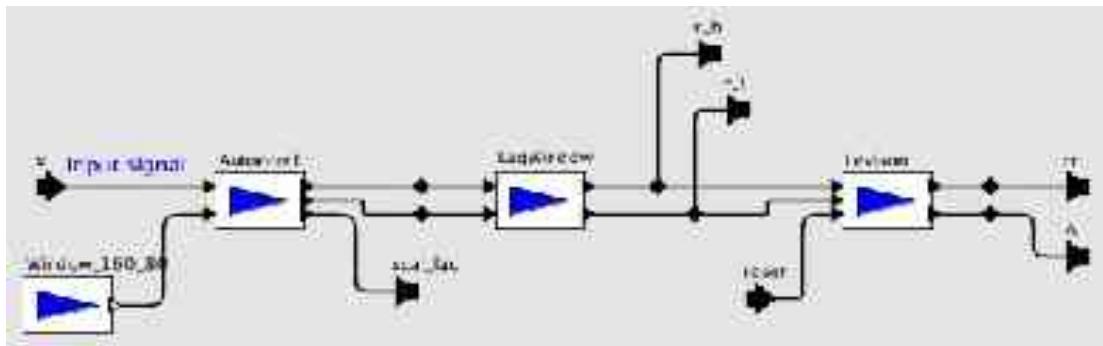


Figura 59 Topologia do ator *Find_Az*.

O ator *Find_Az1* é composto por quatro atores atômicos. O ator *Window_160_80* é outra instância do ator *Const* da biblioteca de atores polimórficos. Os atores *Autocorr1*, *LagWindow* e *Levison* implementam os procedimentos da linhas 4, 5 e 6 da Figura 51.

O ator *Find_Az2* da Figura 58 é igual ao ator *Find_Az1*, apenas substituindo o valor da constante do ator *Window_160_80*.

5.1.3 Análise

Observando-se o código do modelo de referência, nota-se que o codificador é um exemplo de aplicação orientada ao fluxo de dados, isto é, sua funcionalidade é centrada na obtenção e transformação de um grande conjunto de dados. Isto pode ser concluído devido à presença de vários atores transformacionais. Assim, pode-se imaginar, a priori, que um modelo de fluxo de dados (PN, SDF) é adequado para controlar a especificação executável do codificador.

Foi possível implementar uma especificação executável independente de um modelo computacional em particular, devido a presença de um grande conjunto de atores transformacionais. Desta forma, foi possível separar o código referente a comunicação de dados do código que efetua a computação sobre os dados (por exemplo, vide Figura 56). Isto é alcançado encapsulando toda a computação em métodos auxiliares. Entretanto, o ator *Lsp_Old_Init* é um exemplo de ator com atraso. Conforme mencionado no capítulo 4, tal ator não pode ser escrito de maneira polimórfica.

A especificação do codificar foi submetida a ferramenta FASI. Os resultados obtidos para os atores atômicos foram:

Tabela 4 – Resultado da análise dos atores atômicos do codificador de voz.

Ator	SDF	SR
<i>FrameGen</i>	Válido	Válido
<i>FrameCheck</i>	Válido	Válido
Constantes (TXDTX)	Válido	Válido
<i>Homing</i>	Válido	Válido
<i>FilterAndScale</i>	Válido	Válido
<i>FrameBuffer</i>	Válido	Válido
<i>Az_Lsp1</i>	Válido	Válido
<i>Az_Lsp2</i>		
<i>VAD_Lp</i>	Inválido	Válido
<i>Lsp_Old_Init</i>	Específico	Específico
<i>Int_lpc2</i>	Válido	Válido
<i>Q_plsf_5</i>	Válido	Válido
<i>JoinAt</i>	Válido	Válido
<i>Int_lpc</i>	Inválido	Válido
<i>Autocorr1</i>	Válido	Válido
<i>Autocorr2</i>		
<i>LagWindow1</i>	Válido	Válido
<i>LagWindow2</i>		
<i>Levison1</i>	Válido	Válido
<i>Levison2</i>		

Para o modelo SDF, dois atores atômicos não apresentam uma descrição válida. Todos os demais atores foram determinados como homogêneos. Para o modelo SR, todos os atores atômicos estão bem definidos.

O ator *VAD_Lp* foi determinado como inválido para o modelo SDF pois alguns portos de entrada tem uma taxa de amostragem variando entre 0 e 1. A Figura 60 apresenta um trecho simplificado do método *fire()* deste ator.

```

1-   ...
2-   if(_dtx_mode) {
3-       ObjectToken obj = (ObjectToken) r_h.get(0);
4-       ...
5-       obj = (ObjectToken) r_l.get(0);
6-       ...
7-       obj = (ObjectToken) scal_acf.get(0);
8-       ...
9-       obj = (ObjectToken) rc.get(0);
10-      ...
11-      BooleanToken bt = (BooleanToken) ptch.get(0);
12-      ...
13-      boolean VAD_flag = _vad_computation();
14-      ...
15-  }
16-  else {
17-      txdtx_ctrl.broadcast();
18-  }
19-  ...

```

Figura 60 Trecho do método *fire()* do ator *VAD_Lp*.

Dependendo do valor de um parâmetro (*dtx_mode*, linha 2), alguns portos de entrada podem ser lidos (linhas 3 a 11) ou não (linhas 16 a 18). Desta forma, esse ator não apresenta uma taxa de amostragem constante. A mesma situação ocorre com o ator *Int_lpc*.

A Figura 61 apresenta os resultados da análise para os atores hierárquicos da especificação do codificador.

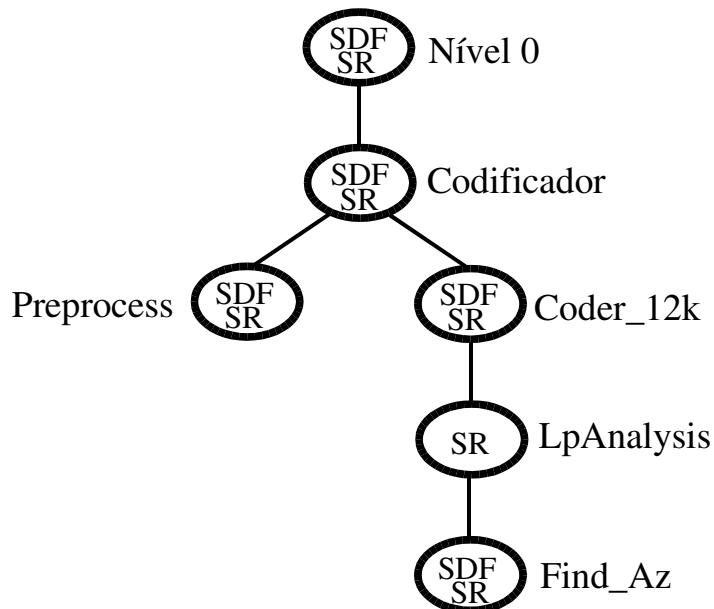


Figura 61 Análise dos atores hierárquicos do codificador.

O modelo SR foi identificado como válido para todos os atores hierárquicos. O modelo SDF foi identificado como válido para todos menos o ator *LP_Analysis*. Isto ocorre pois este ator hierárquico contém os atores *VAD_Lp* e *Int_lpc*, ambos inválidos para o modelo SDF. Entretanto, o ator *LP_Analysis* quando controlado pelo modelo SR, pode ser visto como um ator atômico SDF para a topologia do ator *Coder_12k*, uma vez que ele é homogêneo.

O modelo SDF não pode ser empregado em toda especificação do codificador pois dois atores atômicos, em certas circunstâncias, não estão lendo alguns valores de entrada. Entretanto, estes valores estão sendo produzidos independentemente de serem lidos. É possível fazer uma alteração ao código destes dois atores, a fim de tornar toda a especificação válida para o modelo SDF. Para tal, basta acrescentar um comando de leitura para todos os dados ignorados em ambos os atores. O resultado da leitura é então descartado. Por exemplo, para o código da Figura 60, deve-se adicionar a leitura dos sinais *r_h*, *r_l*, *scal_acf*, *rc* e *ptch* entre as linhas 16 e 18.

5.1.4 Implementação

Duas implementações para o codificador foram geradas, uma para a combinação que utiliza a versão original dos atores *VAD_Lp* e *Int_lpc*, e outra que utiliza a versão modificada, tornando o ator *LP_Analysis* válido para modelos de fluxo de dados. Assim, a primeira versão empregou uma combinação dos modelos SDF e SR, e a segunda apenas um modelo. Ao invés de utilizar o modelo SDF para a segunda versão, foi empregado no lugar o modelo PN. Desta forma, pode-se construir uma implementação com escalonamento estático e outra com escalonamento dinâmico, para um exemplo de sistema real.

5.1.4.1 A versão SDF + SR

Nesta versão, o ator *Lp_Analysis* é controlado pelo modelo SR enquanto que os demais atores hierárquicos são controlados pelo modelo SDF.

A topologia de nível 0

Os atores *FrameGen* e *FrameCheck* da topologia de nível 0 são utilizados para gerar pacotes de teste e verificar se os resultados produzidos pelo codificador são corretos, respectivamente. Cada um destes atores pode ser mapeado em um processo na arquitetura alvo. De acordo com a metodologia do capítulo 3, o ator *Coder* é associado a um processo. A topologia de nível 0 simula o ambiente de operação do codificador, uma vez que é possível implementar a taxa de recepção de pacotes especificada para o codificador. A Figura 62 apresenta o código simplificado para o processo referente ao ator *FrameGen*.

```

1- for(i = 0; i < N; i++) {
2-     cyg_mutex_lock(&frameProcessedLock);
3-     if(frameProcessed == 0) {
4-         diag_printf("FRAME OVERLAP!\n");
5-     }
6-     frameProcessed = 0;
7-     cyg_mutex_unlock(&frameProcessedLock);
8-
9-     for(k = 0; k < L_FRAME; k++) {
10-         frames[i][k] = frames[i][k] & (Word16) 0xffff8;
11-     }
12-     new_frame = frames[i];
13-
14-     cyg_flag_setbits(&rxFlag, 1);
15-     cyg_thread_delay(2);
16- }
```

Figura 62 Trecho de código do processo referente ao ator *FrameGen*.

O processo utiliza a flag *rxFlag* (linha 14) para indicar ao processo do ator *Coder* que um novo pacote está disponível. A variável *frameProcessed* (linhas 3 a 5) indica quando o processamento sobre um novo pacote foi concluído pelo codificador. Caso um novo pacote seja enviado quando o processamento do pacote anterior ainda não tenha sido concluído, uma mensagem de erro é gerada. O processo do ator *FrameGen* se interrompe a cada 20 ms (linha 15), emulando assim a taxa de processamento determinada na especificação do codificador.

O sistema a ser implementado corresponde ao ator *Coder*, que também é associado a um processo. Um trecho do código desse processo pode ser observado na Figura 63.

```

1- initCoder();
2- while(1) {
3-   cyg_flag_wait(&rxFlag, 1,
4-     CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
5-
6-   coder(new_frame);
7-
8-   cyg_flag_setbits(&txFlag, 1);
9-
10-  cyg_mutex_lock(&frameProcessedLock);
11-  frameProcessed = 1;
12-  cyg_mutex_unlock(&frameProcessedLock);
13- }

```

Figura 63Código simplificado para o processo associado ao ator Coder.

Na linha 1, toda a iniciação do codificador é efetuada. Nas linhas 3 e 4, o processo espera por um novo pacote de dados monitorando a flag *rxFlag*. O procedimento *coder()* (linha 6) corresponde ao escalonamento do ator *Coder* por uma iteração. Uma vez concluído o processamento de um pacote, o processo associado ao ator *FrameCheck* é sinalizado através da flag *txflag* (linha 8).

A implementação do codificador (ator *Coder*)

Foi determinado na etapa de análise que todos os atores atômicos controlados pelo modelo SDF são homogêneos. Assim, o escalonamento das topologias controladas pelo modelo SDF corresponde a alguma ordem topológica das mesmas. O mesmo ocorre para o escalonamento do ator *LP_Analysis*, uma vez que seu grafo de dependências é acíclico. A Figura 64 mostra um possível escalonamento para a topologia a partir do ator *Coder*.

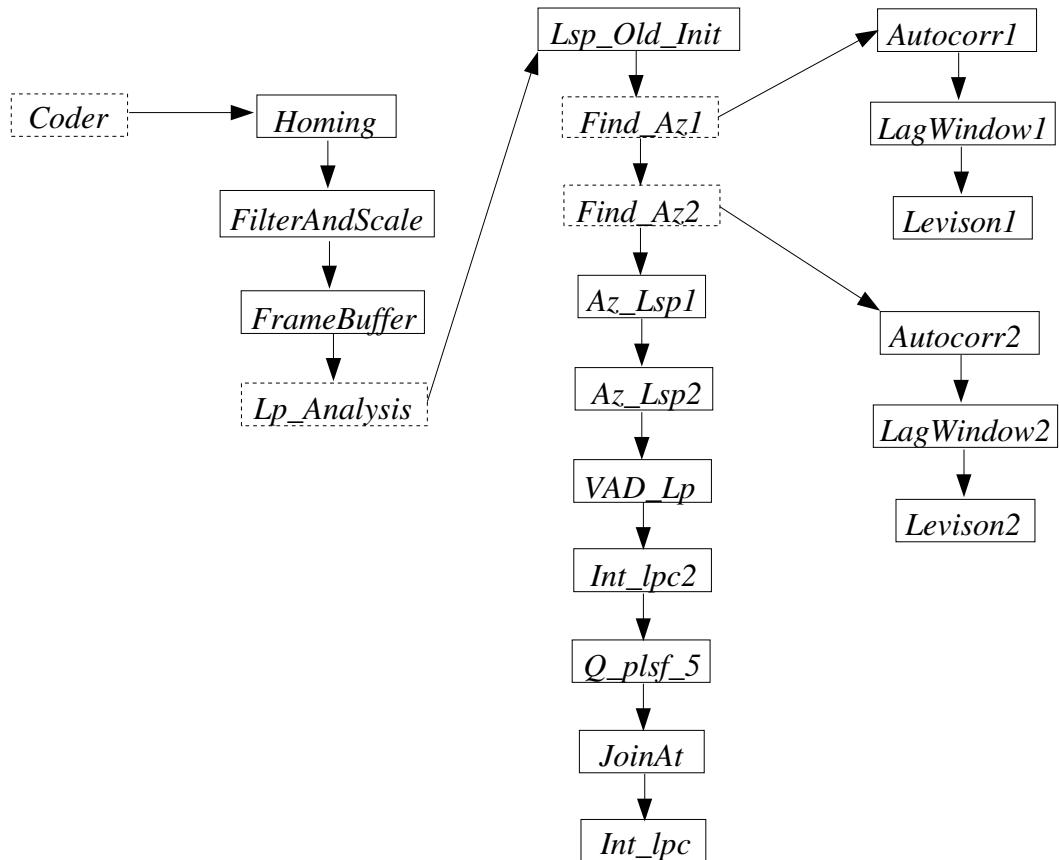


Figura 64 Um possível escalonamento para o ator Coder.

Após o escalonamento, deve-se alocar as posições de memória referentes a cada fila, conforme descrito no capítulo 3. Como todos os atores atômicos controlados pelo modelo SDF são homogêneos, cada fila pode ser implementada por um vetor estático de tamanho igual a 1. A única exceção é o ator *Lsp_Old_Init*, cuja fila associada ao porto de saída possui tamanho igual a 2.

O tipo de dados transmitido através de cada fila depende de cada ator. Na especificação do codificador, as conexões transmitem vetores ou valores escalares. Por exemplo, o sinal *reset* ou uma palavra de controle são representados por valores escalares. Um pacote do sinal de voz é um exemplo de tipo de dado vetor. Uma característica da especificação do codificador é que todos os vetores transmitidos apresentam tamanho constante. Assim, não é necessário utilizar alocação dinâmica de memória. A Figura 65 exemplifica a alocação de memória através da versão implementada do *fire()* para o ator *JoinAt*.

```

1- void
2- JoinAtFire(Word16 *input1, Word16 *input2, Word16 *input3,
3-             Word16 *output) {
4-     int i;
5-
6-     for(i = 0;i < 4 * MP1;i++) {
7-         output[i] = input1[i];
8-     }
9-
10-    for(i = 0;i < MP1;i++) {
11-        output[MP1 + i] = input2[i];
12-    }
13-
14-    for(i = 0;i < MP1;i++) {
15-        output[MP1*3 + i] = input3[i];
16-    }
17- }
```

Figura 65Método fire() do ator JoinAt.

O ator *JoinAt* é responsável por agregar três sinais calculados por outros atores pertencentes a topologia *Lp_Analysis*. Todos os sinais de entrada (*input1*, *input2* e *input3*) e de saída (*output*) são vetores de tamanho constante. Esses parâmetros são associados aos respectivos *buffers* de comunicação na chamada do procedimento *JoinAtFire()*.

5.1.4.2 A versão PN

Nesta versão, apenas o modelo PN é utilizado para controlar os atores.

A topologia de nível 0

Como no caso da implementação SDF + SR, os atores *FrameGen* e *FrameCheck* são associados a processos. Entretanto, nessa versão a comunicação se dá através dos canais PN, o que torna desnecessário a presença das *flags* para indicar a chegada e o processamento de um pacote, uma vez que os processos bloqueiam na falta de dados.

A implementação do codificador (ator Coder)

Nesta versão, o procedimento que implementa o ator *Coder* contém a criação dos processos associados aos atores atômicos, ao invés de chamadas de procedimentos implementando um escalonamento.

Esta versão da especificação também é válida para o modelo SDF. Em particular, a especificação é composta apenas por atores homogêneos. Assim, o tamanho necessário para cada fila de comunicação no modelo PN é igual a 1, e a versão estática do canal PN pode ser empregada.

5.1.5 Resultados

A tabela 5 apresenta a caracterização das duas versões da implementação do codificador e do modelo de referência com relação ao tempo de processamento de um pacote. Para cada implementação, duas diretivas de otimização do compilador foram utilizadas: -O2 executa um conjunto básico de algoritmos sobre o código fonte; -O3 adiciona 6 novos algoritmos além dos presentes na diretiva -O2. Para cada caso de teste, a tabela apresenta a média e o desvio padrão do tempo de processamento.

Tabela 5 – Resultados de tempo de resposta das implementações do codificador de voz.

	<i>Mod. de Referência</i>		<i>SDF+SR</i>		<i>PN</i>	
Compilado com -O3	637 µs	4	639 µs	5	671 µs	12
Compilado com -O2	923 µs	4	929 µs	5	942 µs	12

A versão SDF + SR apresenta dados de desempenho minimamente maiores que a versão do modelo de referência. Desta forma, a transformação da comunicação através de variáveis compartilhadas para troca de mensagens não resultou em penalidade de desempenho do sistema.

Os valores obtidos para a implementação no modelo PN são os maiores das três versões. Isto já era esperado, devido ao escalonamento dinâmico empregado na versão PN. Para a versão compilada com -O2, são 13 µs com relação a versão

SDF+SR, e quando compilado com a diretiva -O3, são 32 μ s. Ambos os valores não são significativos. O aumento da penalidade pode ser explicado pela melhor otimização da implementação SDF + SR com relação a da implementação PN, pois no primeiro caso, é possível fazer *inlining* das chamadas de métodos, e talvez outras otimizações. Assim, embora ambas as implementações melhoraram com relação à diretiva -O2, a implementação SDF + SR foi mais beneficiada.

Os dados da versão PN apresentam um desvio padrão maior que os das outras versões. Isto pode ser explicado devido ao escalonamento dinâmico e a necessidade de sincronização, que tornam a execução mais aleatória.

A observação mais importante a ser feita sobre os dados obtidos é que a penalidade de se utilizar uma diretiva de otimização mais fraca é muito maior que a de utilizar um modelo computacional teoricamente com execução menos eficiente. A penalidade entre as duas versões da implementação SDF + SR é de 290 μ s, nove vezes maior que a penalidade entre as versões PN e SDF + SR.

Os resultados da tabela 5 foram obtidos em um computador pessoal com processador de AMD Athlon de 1.6GHz. Mesmo implementando apenas a primeira etapa da codificação, observa-se que com a utilização deste processador o tempo de processamento de um pacote fica bastante abaixo do limite de 20 ms. Entretanto, tal configuração é pouco comum em um sistema embarcado, devido ao seu custo. Por exemplo, um telefone celular utiliza a codificação de voz. Somente o custo deste processador ultrapassa o valor de um telefone celular convencional.

Pode-se fazer uma estimativa grosseira dos tempos de processamento de um pacote para um processador tipicamente encontrado em aplicações embarcadas considerando-se uma freqüência de operação menor, como por exemplo 100 Mhz. Neste caso, o sistema seria aproximadamente 16 vezes mais lento. Assim, o tempo de processamento para a especificação SDF+SR compilada com a diretiva -O3 sobe de 639 μ s para 10244 μ s, ou seja um pouco mais de 10 ms. A penalidade da utilização do modelo PN em relação ao modelo SDF + SR subiria de 32 μ s para 512 μ s, ou seja, aproximadamente 5% do tempo de processamento total.

A tabela 6 apresenta o tamanho do código de instruções obtido para as três versões de implementação do codificador, compiladas com as diretivas -O2 e -O3. Para cada situação, a tabela discrimina quantos kbytes são necessários para as

instruções e dados, respectivamente.

Tabela 6 – Resultados de quantidade de memória das implementações do codificador de voz.

	<i>Mod. de Referência</i>	<i>SDF+SR</i>		<i>PN</i>	
Compilado com -O3	30k	3k	27k	3k	33k
Compilado com -O2	28k	3k	25k	3k	30k

Comparando os dados da tabela 6 com relação as diretivas de otimização do compilador, nota-se uma penalidade no tamanho das instruções da diretiva -O3 com relação a diretiva -O2. Essa penalidade é pequena e aceitável, considerando-se o ganho de tempo de processamento da diretiva -O3.

O implementação SDF + SR se mostrou praticamente igual a do modelo de referência, isto é, não houve uma penalidade na tradução da especificação de referência para uma baseada em atores.

A implementação PN utiliza mais instruções que a implementação SDF + SR. Isto se deve ao código necessário para criar os vários processos associados aos atores atômicos, além do código para cada canal de comunicação. Entretanto, nota-se uma penalidade significativa no espaço de armazenamento para dados nessa implementação. É necessário sete vezes mais espaço para a versão PN com relação a versão SDF + SR. Analisando o tamanho de cada módulo da implementação, determinei que o responsável por esse aumento da quantidade necessária de dados são as pilhas associadas a cada processo. Experimentando alguns valores, e utilizando o comando do sistema *eCos* para monitorar a variação do tamanho da pilha de um processo, percebi que um tamanho que permite a execução é em torno de 1kbyte por pilha. São 19 o número de processos na versão PN, contra 1 na versão SDF+SR.

5.2 Controle de trenó

O segundo estudo de caso que desenvolvi é um sistema de controle para um trenó de carga. Este exemplo foi proposto em uma conferência científica [MOS99], com o propósito de testar diferentes linguagens para especificação de sistemas digitais [GOR00].

5.2.1 Especificação

A especificação para o sistema do trenó é composta exclusivamente de um texto em língua inglesa, detalhando o sistema, incluindo um modelo para a planta, os sensores, os atuadores e o controle do trenó. Também são descritos casos de teste para o sistema.

O modelo da Planta

A Figura 66 apresenta um diagrama do sistema, incluindo as variáveis modeladas.

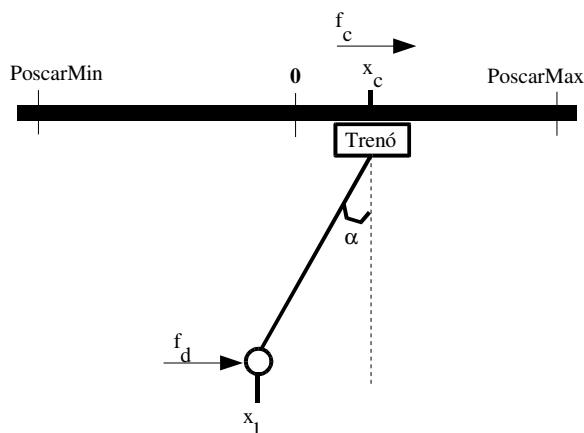


Figura 66 Modelo do sistema do trenó.

O trenó se desloca por um trilho entre uma posição mínima (PoscarMin) e máxima (PoscarMax). São cinco as variáveis modeladas:

- x_c : representa a posição atual do trenó;
- x_l : representa a posição atual da carga transportada;
- α : ângulo entre o cabo que liga o trenó a carga e a perpendicular;
- f_d : força que atua sobre a carga (modela o vento, p.ex)
- f_c : força que atua sobre o tremó.

O modelo para a planta de operação é então descrito por três equações:

$$\ddot{x}_c = \frac{f_c}{m_c} + g \frac{m_l}{m_c} \alpha - \frac{d_c}{m_c} \dot{x}_c \quad (7)$$

$$r \ddot{\alpha} = -g \left(1 + \frac{m_l}{m_c}\right) \alpha + \left(\frac{d_c}{m_c} - \frac{d_l}{m_l}\right) \dot{x}_c - r \frac{d_l}{m_l} \dot{\alpha} - \frac{f_c}{m_c} + \frac{f_l}{d_l} \quad (8)$$

$$x_l = x_c + r \alpha \quad (9)$$

, onde

$$m_c = 10.0 \text{ kg}, m_l = 100.0 \text{ kg}, g = 9.81 \text{ m/s}^2, d_c = 0.5 \text{ s}, r = 5.0 \text{ m e } d_l = 0.01 \text{ s}.$$

Desta forma, dado f_d e f_c é possível calcular x_l , x_c e o ângulo α .

Os sensores

A especificação determina a presença de cinco sensores:

- Poscar : a posição atual do trenó (x_c);
- SwPoscarMin : valor booleano que identifica quando a posição do trenó for menor que a mínima permitida;
- SwPoscarMax : valor booleano que identifica quando a posição do trenó for maior que a máxima permitida;
- Alpha : o valor do ângulo α ;
- ShutDown : identifica que o usuário do sistema deseja seu desligamento;
- PosDesired : posição desejada para a carga.

É especificado que o intervalo mínimo entre a ocorrência de dois eventos em qualquer um destes sensores é de 2 ms.

Os atuadores

O sistema de controle do trenó deve controlar dois valores de atuação:

- VC : voltagem do motor DC que impulsiona o trenó;
- Brake : freio do trenó.

A força sobre o trenó (f_c) pode ser obtida a partir da voltagem VC segundo a seguinte equação:

$$t_m \dot{f}_c + f_c = k_m V C, \text{ onde } t_m = 1 \text{ seg e } k_m = 4.0 \text{ N/volt.} \quad (10)$$

O sistema de controle

O algoritmo de operação para o sistema de controle é fornecido:

1. Iniciação: acione o freio, inicialize todos os componentes do sistema e execute a verificação dos sensores;
2. Ative o bloco de diagnósticos;
3. Execute em iteração, até o usuário desligar o sistema;
 1. Espere por um novo valor de posição desejada para a carga;
 2. Libere o freio do trenó;
 3. Execute o algoritmo de controle.

A verificação dos sensores da etapa 1 consiste em observar restrições impostas aos valores dos sensores. O trenó é movimentado até ambas as extremidades do trilho, enquanto os valores dos sensores são monitorados. Em particular, as seguintes condições são testadas:

1. Com o trenó parado, verifique se $(Poscar < PoscarMin - \Delta Poscar) \vee (Poscar > PoscarMax + \Delta Poscar) \vee (SwPoscarMin \wedge SwPoscarMax)$, onde $\Delta Poscar = 0.1$, $PoscarMin = -5.0$ e $PoscarMax = 5.0$;
2. Mova o trenó para a esquerda ($VC = -0.002$) até $Poscar \leq PoscarMin - \Delta Poscar \vee SwPoscarMin$. Verifique se $Poscar > PoscarMin + \Delta Poscar \vee \neg SwPoscarMin$;
3. Mova o trenó para a direita ($VC = 0.002$) até $Poscar \geq PoscarMax + \Delta Poscar \vee \neg SwPoscarMax$.

$\Delta\text{Poscar} \vee \text{SwPoscarMax}$. Verifique se $\text{Poscar} < \text{PoscarMax} - \Delta\text{Poscar} \vee \neg\text{SwPoscarMax}$;

4. Mova o trenó para a esquerda até $\text{Poscar} \leq \text{PoscarMax} - \Delta\text{Poscar} \vee \text{SwPoscarMin}$. Verifique se $\text{SwPoscarMin} \vee \text{SwPoscarMax}$.

Caso alguma das condições acima resultar em um valor verdade, um procedimento de parada de emergência (*EmergencyStop*) deve ser executado. Neste procedimento, o trenó é imediatamente parado ($\text{VC} = 0$ e $\text{Brake} = \text{verdade}$), e o sistema interrompe seu funcionamento. Para retornar a operação após uma parada de emergência, o usuário deve desligar e religar o sistema.

Os diagnósticos da etapa 2 são executados concorrentemente com o algoritmo de controle. São três os diagnósticos efetuados:

1. Posição desejada para a carga (PosDesired) deve estar dentro do intervalo PoscarMin e PoscarMax ;
2. SwPoscarMin ou SwPoscarMax não podem ser válidos por mais que 20ms;
3. O ângulo α não pode ter um valor absoluto maior que 0.1 radianos por mais de 50ms.

Os diagnósticos 1 e 2 fazem com que uma parada de emergência seja executada. O diagnóstico 3 altera o comportamento do algoritmo de controle, que não mais utiliza o valor do ângulo α , mas sim uma estimativa de seu valor.

O algoritmo de controle é baseado em um ciclo de 10ms. Sua finalidade é produzir um valor de voltagem VC do motor do trenó. A especificação do sistema de trenó detalha seus passos.

5.2.2 Especificação executável

A especificação executável consiste em um modelo da planta e do sistema de controle. O modelo da planta é utilizado para obter os valores da posição da carga, do trenó e do ângulo α . Além destes três valores de entrada, o sistema de controle recebe comandos do usuário, através dos sinais de desligamento (Shutdown),

iniciação (PowerOn) e posição desejada para a carga (PosDesired). A Figura 67 apresenta a topologia de nível 0 para a especificação.

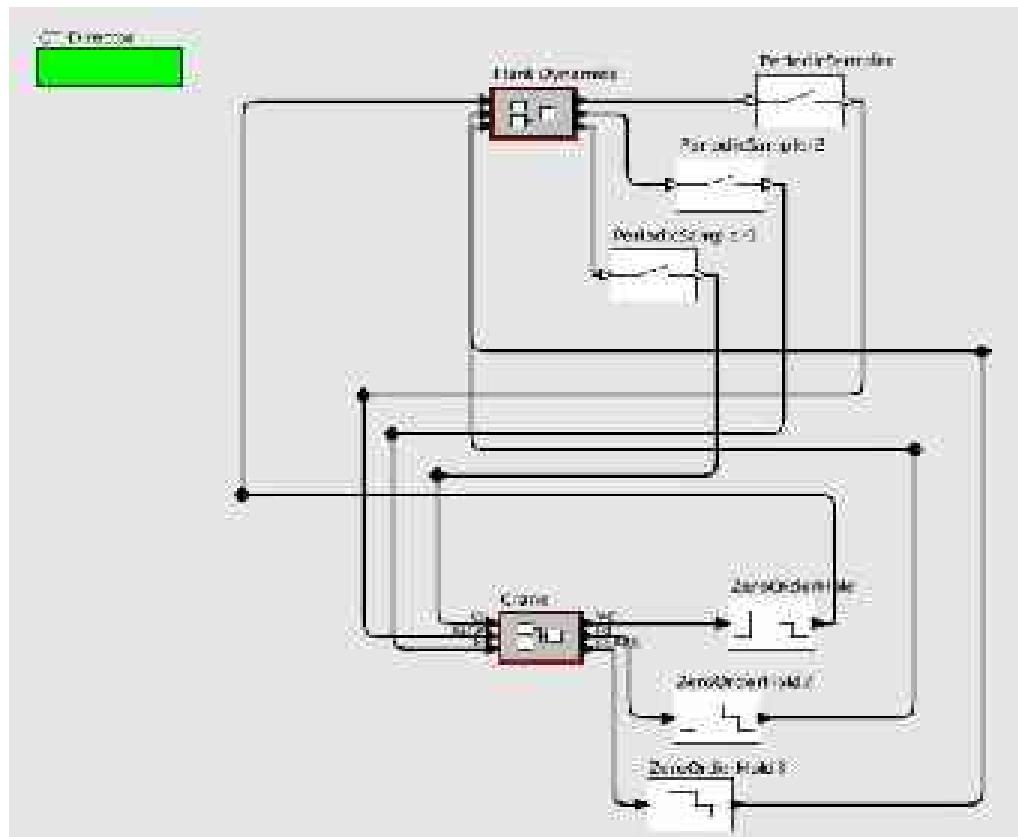


Figura 67 Topologia de nível 0 para o sistema do trenó.

A topologia é específica para o modelo CT. Este modelo computacional é o adequado para capturar as equações diferenciais da planta. O ator *PlantDynamics* contém a implementação destas equações, enquanto que o ator *Crane* contém o sistema de controle. Os atores atômicos *PeriodicSampler* e *ZeroOrderHold* são utilizados para converter um sinal contínuo em um evento discreto e vice-versa, respectivamente. O ator *PeriodicSampler* possui um parâmetro que especifica a taxa de amostragem do sinal contínuo. Este valor foi configurado para 2 ms, uma vez que este é o intervalo mínimo entre dois eventos dos sinais produzidos pelo ator *PlantDynamics*. A Figura 68 apresenta a topologia desse ator.

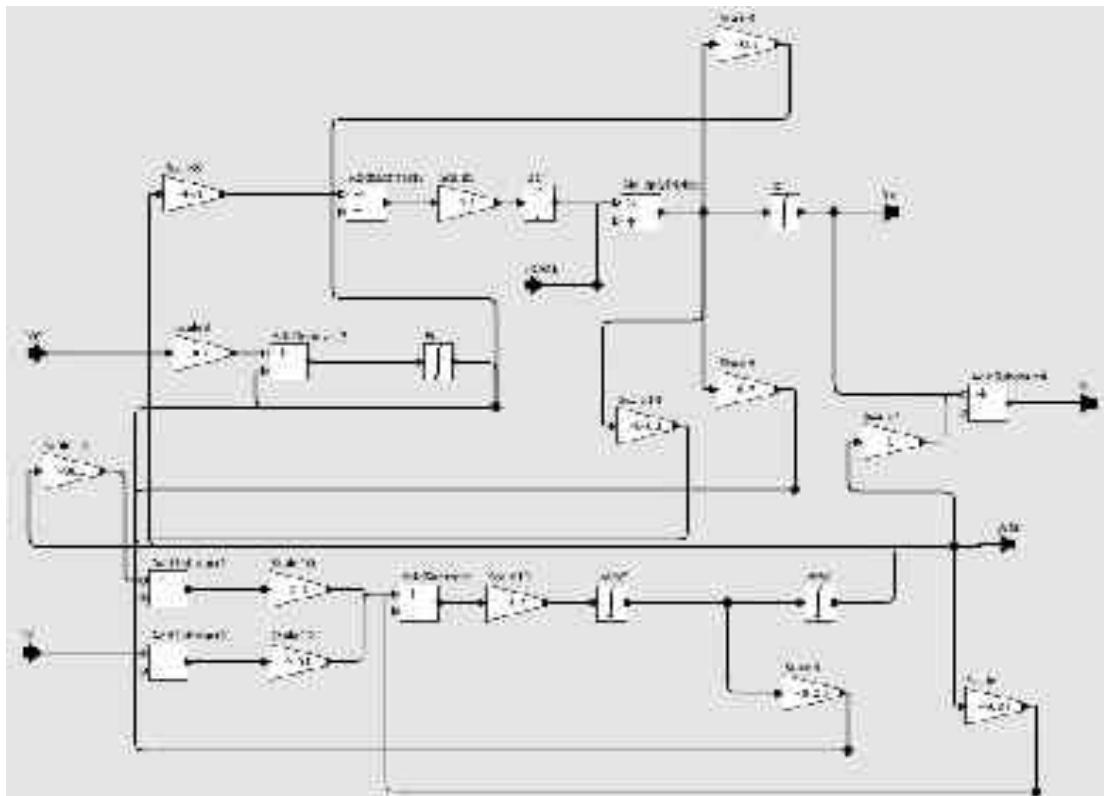


Figura 68 Topologia do ator PlantDynamics.

A entrada Brake foi adicionada ao ator *PlantDynamics* a fim de modelar a frenagem do trenó. Quando este valor for igual a zero, a velocidade do trenó (x_c) irá para zero.

A Figura 69 apresenta a topologia para o ator *Crane*.

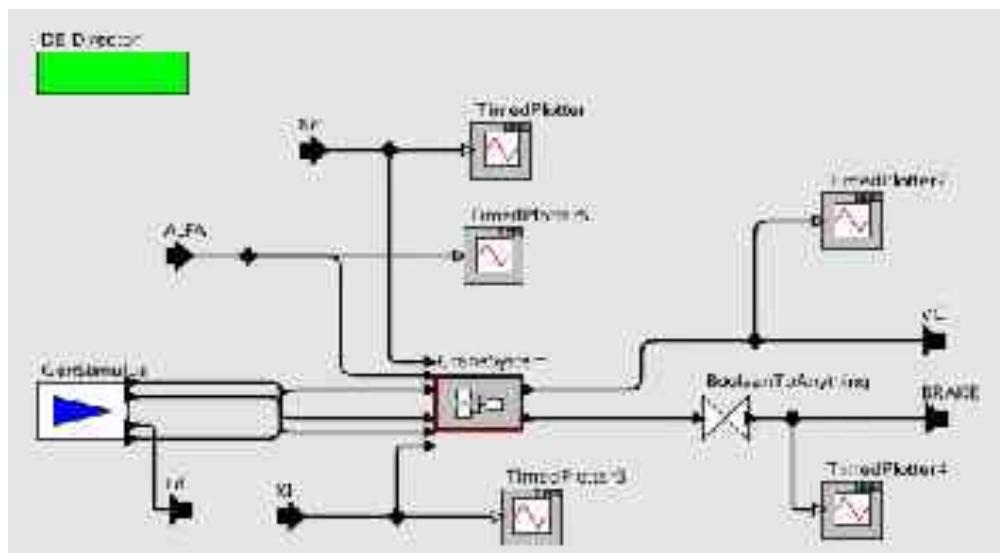


Figura 69 Topologia do ator hierárquico Crane.

O ator *Crane* é composto por cinco instâncias (*TimedPlotter*, *TimedPlotter2*, *TimedPlotter3*, *TimedPlotter4* e *TimedPlotter5*) do ator atômico *TimedPlotter*, uma do ator atômico *BooleanToAnything*, uma do ator atômico *GenStimulus* e uma do ator hierárquico *CraneSystem*. Os atores *TimedPlotter* e *BooleanToAnything* pertencem a biblioteca de atores polimórficos, e são utilizados para desenhar um gráfico em duas dimensões e converter um valor booleano em outro (neste caso, um valor em ponto flutuante), respectivamente.

O ator *GenStimulus* é utilizado para capturar os casos de teste propostos na especificação do sistema. Além dos valores dos sinais, uma temporização é especificada. Por exemplo, um teste determina a posição desejada para carga como -3.5. Após 4 segundos, deve-se modificar o valor da força f_d para 400 N. Depois de 1 segundo do novo valor de f_d , deve-se modificar seu valor para 0.0 N. A fim de capturar corretamente as temporizações especificadas, o ator *GenStimulus* foi implementado especificamente para operar no modelo computacional DE. Logo, a topologia da Figura 69 é específica para tal modelo.

O ator *CraneSystem* contém a implementação do sistema de controle. É importante notar a natureza dos sinais de entrada e de saída deste ator. Podemos dividí-los em dois grupos: periódicos e aperiódicos. Os sinais de entrada periódicos (XL, Poscar, Alpha) são oriundos do ator *PlantDynamics*, e estão sempre presentes a cada 2 ms. Os sinais de entrada aperiódicos (PosDesired, PowerOn e Shutdown) são produzidos pelo ator *GenStimulus*, e sua presença é aperiódica. Todos os sinais de saída (Brake e VC) são periódicos.

O algoritmo de operação fornecido pela especificação determina claramente três blocos para o sistema: verificação dos sensores, diagnósticos e controle do trenó.

A verificação dos sensores é uma tarefa seqüencial composta de quatro etapas. Após uma verificação da posição inicial, o trenó é deslocado ao longo do trilho. Implementei este comportamento através de um ator atômico com estados, onde cada estado representa uma etapa da verificação. A Figura 70 apresenta um trecho do método *fire()* deste ator.

```

1- case CHECK1: {
2-     if(poscar.isKnown() && swposcarmin.isKnown() &&
3-         swposcarmax.isKnown()) {
4-         if(poscar.hasToken(0)) {
5-             poscarTk = (DoubleToken) poscar.get(0);
6-             double posc = poscarTk.doubleValue();
7-             boolean minposc =
8-                 ((BooleanToken) swposcarmin.get(0)).booleanValue();
9-             boolean maxposc =
10-                 ((BooleanToken) swposcarmax.get(0)).booleanValue();
11-
12-             if(posc <= -5.1 || minposc) {
13-                 if(posc > -4.9 || !minposc) {
14-                     emstop.broadcast(new Token());
15-                     _failedCheck = true;
16-
17-                     brake.broadcast(new BooleanToken(true));
18-                     vc.broadcast(new DoubleToken(0.0));
19-                 }
20-                 else {
21-                     brake.broadcast(new BooleanToken(false));
22-                     vc.broadcast(new DoubleToken(VC_CHECK));
23-                 }
24-                 _endedCheck = true;
25-             }
26-             else {
27-                 brake.broadcast(new BooleanToken(false));
28-                 vc.broadcast(new DoubleToken(-VC_CHECK));
29-             }
30-         }
31-     }
32- } break;

```

Figura 70 Trecho do método *fire()* para a verificação dos sensores.

O trecho exemplificado refere-se a segunda verificação, isto é, deslocar o trenó para a extremidade esquerda (-5.0). Quando o trenó atingir a extremidade esquerda, o código entre as linhas 13 e 24 é executado. Caso contrário, o trenó continua a ser deslocado na mesma direção (linhas 26 a 29). A variável *_failedCheck* (linha 15) indica que a condição da verificação falhou. A variável *_endedCheck* (linha 24) indica que a verificação foi concluída. A Figura 71 apresenta o método *postfire()* desse ator.

```

1-case CHECK1: {
2-  if(_endedCheck) {
3-    if(_failedCheck) {
4-      _reset();
5-    }
6-    else {
7-      _state = CHECK2;
8-      _endedCheck = false;
9-    }
10-  }
11-} break;

```

Figura 71 Trecho do método `postfire()` da verificação dos sensores.

Quando uma etapa for concluída corretamente, o estado do ator é modificado para efetuar a próxima etapa (linha 7). Caso contrário, o ator é reinicializado (linha 4) e uma nova verificação dos sensores só é efetuada após o religamento do sistema através do sinal `PowerUp`. Quando a verificação dos sensores for concluída com sucesso, o ator emite um sinal que indica que o sistema pode ativar os blocos de diagnóstico e controle do trenó.

Os diagnósticos são responsáveis por sinalizar uma parada de emergência e indicar a validade do ângulo α . A Figura 72 apresenta a topologia do ator hierárquico que implementa os três diagnósticos.

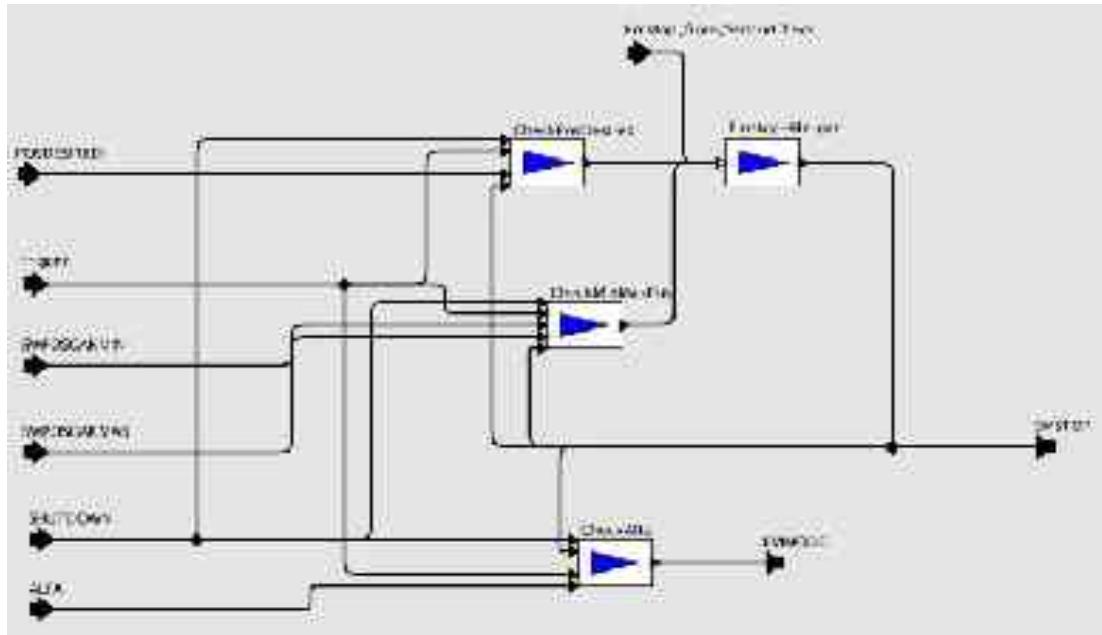


Figura 72 Topologia do ator *Diagnostics*.

O ator atômico `CheckPosDesired` é responsável por determinar se a posição

especificada pelo usuário para a carga é válida. O ator atômico *CheckMinMaxPos* verifica se a posição do trenó é válida. O ator atômico *CheckAlfa* determina a validade do ângulo α . O resultado dos atores *CheckPosDesired* e *CheckMinMaxPos* é sinalizar a necessidade de uma parada de emergência. Assim, quando qualquer um destes dois atores emitir um sinal, o ator atômico *Emstop-Merger* irá sinalizar para o resto do sistema essa condição. Como uma parada de emergência também pode ser emitida durante a etapa de verificação dos sensores, o porto *Emstop_from_SensorCheck* é utilizado para conectar o ator *Emstop-Merger* com o ator que implementa a verificação dos sensores.

Conforme descrito na especificação do sistema, os atores *CheckMinMaxPos* e *CheckAlfa* devem sinalizar uma parada de emergência após detectar uma dada condição, por um determinado intervalo de tempo. Como os sinais sendo monitorados (*Poscar* e *Alfa*) são sinais periódicos, utilizei o artifício de contar a presença de um valor a fim de determinar a passagem de tempo. Desta forma, a implementação destes atores é mais genérica que uma que utilizaria métodos que manipulam diretamente um valor temporal.

Os três atores de diagnósticos iniciam os respectivos monitoramentos após a conclusão da etapa de verificação e interrompem-os quando ocorrer uma parada de emergência ou o usuário desligar o sistema. Para implementar essa funcionalidade, estes atores também utilizam estados para determinar suas ações. A Figura 73 apresenta o método *postfire()* comum aos três atores, mostrando a transição de estados.

```

1-   switch(_state) {
2-
3-     case READY : {
4-       if(trigger.hasToken(0) && !shutdown.hasToken(0)) {
5-         _state = RUNNING;
6-       }
7-     } break;
8-
9-     case RUNNING: {
10-       if(shutdown.isKnown()) {
11-         if(shutdown.hasToken(0)) {
12-           _reset();
13-           break;
14-         }
15-       }
16-
17-       if(emstop_input.isKnown()) {
18-         if(emstop_input.hasToken(0)) {
19-           _reset();
20-           break;
21-         }
22-       }
23-     } break;
24-   }

```

Figura 73Método postfire() comum aos atores de diagnóstico.

Dois estados são utilizados nos atores de diagnóstico : READY (linhas 3 a 7) e RUNNING (linhas 9 a 23). O estado inicial é o estado READY, e indica que o ator está esperado para iniciar o monitoramento do diagnóstico. O estado RUNNING é onde o ator efetua o diagnóstico. A transição do estado READY para RUNNING acontece quando um sinal (*trigger*) que indica o início da operação do sistema for emitido. A transição do estado RUNNING para READY ocorre quando o usuário desligar o sistema ou uma parada de emergência for detectada.

O algoritmo de controle opera em paralelo com os diagnósticos após a etapa de verificação dos sensores. A Figura 74 apresenta a topologia do ator hierárquico que implementa o controle do trenó.

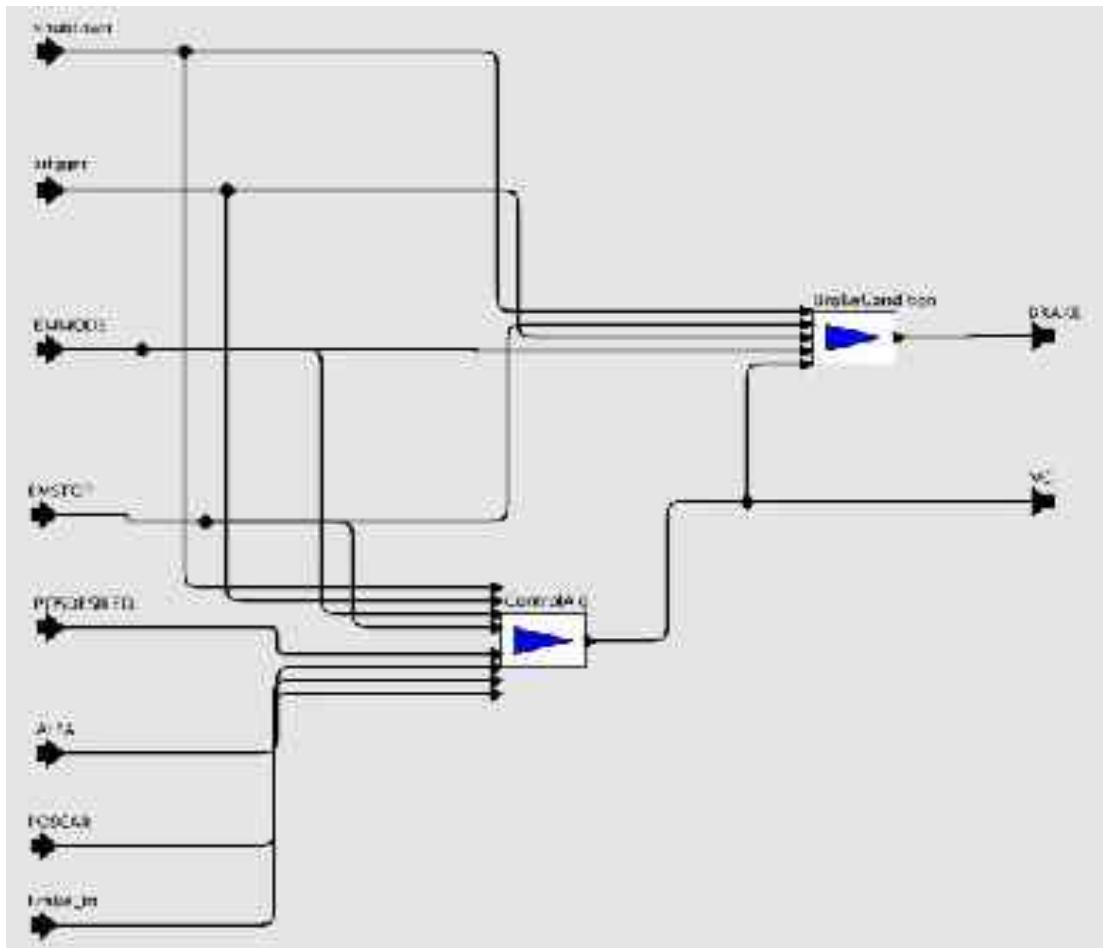


Figura 74Topologia do ator hierárquico Control.

O ator atômico *ControlAlg* implementa o algoritmo de controle. Ele utiliza os valores do ângulo α (ALFA), da posição desejada para a carga (POSDESIRED) e da posição atual do trenó (POSCAR). A cada 10 ms, o ator produz um valor para VC. Novamente, o artifício de contar a presença dos sinais de entrada foi utilizado para determinar a passagem do tempo. O ator atômico *BrakeCondition* implementa a condição de frenagem do trenó. Isto ocorre quando o valor de VC for menor que .01 Volts por um certo intervalo de tempo. Quando a condição for satisfeita, o ator *BrakeCondition* emite um evento no sinal BRAKE.

Ambos os atores também devem implementar máquinas de estado, pois suas funcionalidades dependem de valores emitidos nos sinais aperiódicos (Brake_in, POSDESIRED, EMSTOP, EMMODE, trigger e shutdown). A Figura 75 mostra o diagrama de transição para o ator *ControlAlg*.

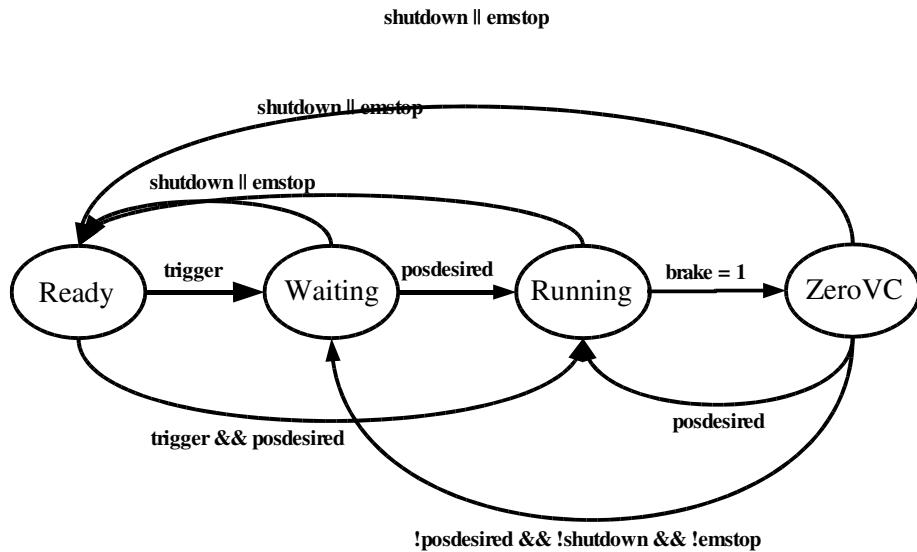


Figura 75 Máquina de estados do ator *ControlAlg*.

A Figura 76 apresenta a topologia do ator *CraneSystem* da Figura 69, que implementa o controle do trenó.

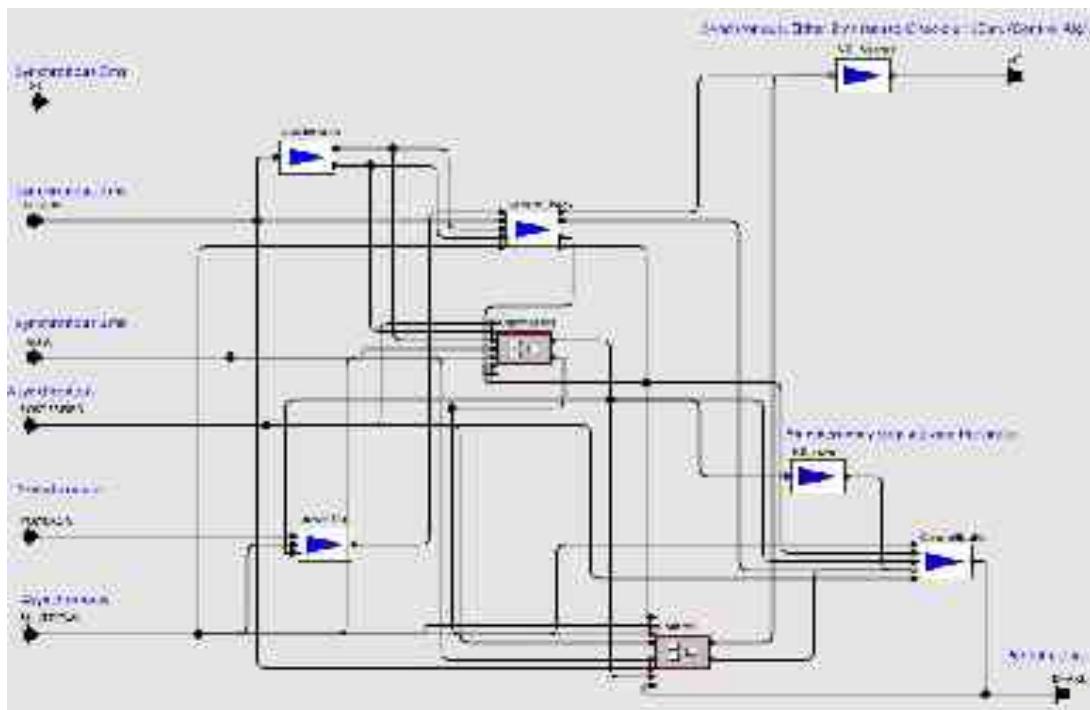


Figura 76 Topologia do ator *CraneSystem*.

Os atores *SensorCheck*, *Diagnostics* e *Control* implementam a verificação dos sensores, diagnósticos e o algoritmo de controle respectivamente. Além destes três

atores, outros cinco atores atômicos são utilizados na topologia do ator *CraneSystem*. O ator atômico *posSensors* é utilizado para determinar os valores dos sensores *SwPoscarMin* e *SwPoscarMax* para cada novo valor da posição do trenó (POSCAR). O ator atômico *PowerUp* implementa uma máquina de estado que sinaliza quando o sistema for ligado pelo usuário, ativando a etapa de verificação de sensores. O ator atômico *VC-Merger* é utilizado para concatenar o valor de VC produzido pelos atores *SensorCheck* e *Control*. O ator atômico *isKnown* determina quando há um evento em um sinal de entrada. No caso da topologia da Figura 76, esse ator irá indicar a presença de uma parada de emergência. O ator atômico *ControlBrake* controla o freio do trenó, concatenando todas as possíveis situações de frenagem, isto é, uma parada de emergência, desligamento do sistema e a condição de frenagem satisfeita.

5.2.3 Análise

A especificação executável do ítem 5.2.2 possui dois atores hierárquicos específicos (*PlantDynamics* e *Crane*). Estes atores são utilizados para simular o ambiente onde o sistema será instalado e empregam os modelos computacionais adequados para tal.

O ator *CraneSystem* captura o sistema de controle a ser implementado. Alguns de seus atores atômicos implementam máquinas de estado que reagem a sinais aperiódicos. Em geral, modelos de fluxo de dados não são capazes de capturar tal comportamento da maneira adequada. A tabela 7 mostra o resultado obtido pela ferramenta FASI referente aos atores atômicos contidos a partir do ator *CraneSystem*.

Tabela 7 – Resultado da análise dos atores atômicos do sistema de Trenó.

Ator	SDF	SR
<i>posSensors</i>	Válido	Válido
<i>powerUp</i>	Inválido	Válido
<i>SensorCheck</i>	Inválido	Válido
<i>isKnown</i>	Válido	Válido
<i>ControlBrake</i>	Inválido	Válido

Ator	SDF	SR
<i>CheckPosDesired</i>	Inválido	Válido
<i>CheckMinMaxPos</i>	Inválido	Válido
<i>CheckAlfa</i>	Inválido	Válido
<i>Emstop-Merger e VC-Merger</i>	Válido	Válido
<i>ControlAlg</i>	Inválido	Válido
<i>BrakeCondition</i>	Inválido	Válido

Para o modelo SDF, apenas três atores são válidos. Todos os demais apresentam uma taxa de amostragem não constante, uma vez que implementam máquinas de estado sobre sinais aperiódicos. Todos os atores atômicos são válidos para o modelo SR.

O resultado para a análise dos atores hierárquicos é apresentado na Figura 77.

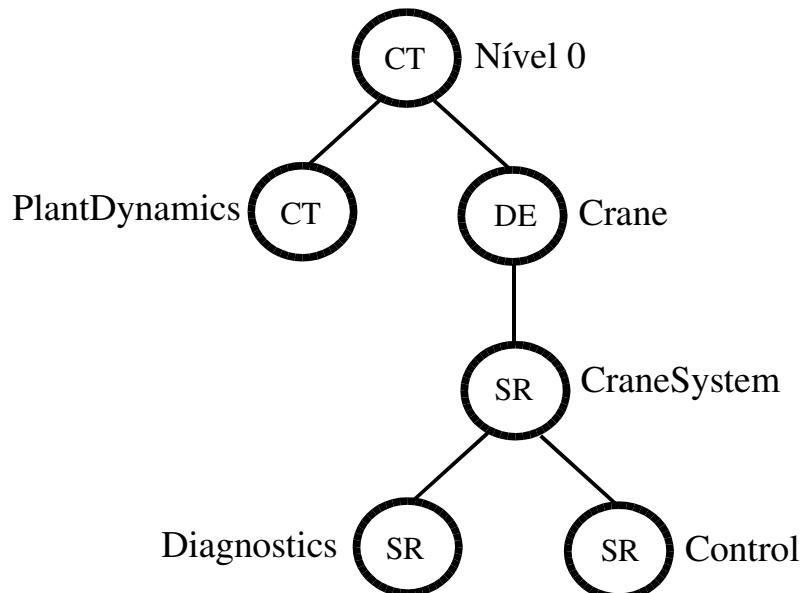


Figura 77 Resultado da análise para os atores hierárquicos do sistema de controle.

Os atores *PlantDynamics* e a topologia de nível zero são específicos para o modelo CT, enquanto que o ator *Crane* é específico para o modelo DE. O sistema de controle pode ser controlado pelo modelo SR, pois além de possuir todos os atores atômicos válidos para esse modelo, não apresenta nenhum ciclo com dependência.

5.2.4 Implementação

A topologia de nível 0

O ator *CraneSystem* é o objeto a ser implementado na arquitetura alvo. Entretanto, a fim de testar e caracterizar a implementação, os atores *PlantDynamics* e *GenStimulus* também foram mapeados em processos na arquitetura alvo. Os processos referentes aos atores *PlantDynamics* e *GenStimulus* utilizam o procedimento *cyg_thread_delay()* a fim de modelar o comportamento dos sinais periódicos e aperiódicos.

A implementação do ator *CraneSystem*

O ator *CraneSystem* é controlado apenas pelo modelo computacional SR. O processo associado a este ator aguarda pela presença de um evento para então executar uma iteração do ator. A Figura 78 apresenta um trecho do código fonte deste processo.

```

1- char Brake;
2- double VC;
3-
4- double Alfa;
5- double XL;
6- double XC;
7-
8- double posdesired;
9-
10- static void
11- craneSystem(cyg_addrword_t p) {
12-
13-     initCrane();
14-
15-     for(;;) {
16-         cyg_flag_wait(&hasEvent, 1,
17-             CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
18-         crane();
19-         hasEventFlags = 0;
20-     }
21- }
```

Figura 78Trecho de código do processo que implementa o ator *CraneSystem*.

Na linha 13, o procedimento responsável por inicializar o estado dos diversos atores atômicos contidos no ator *CraneSystem* é chamado. Nas linhas 16 e 17, o processo aguarda pela ocorrência de algum evento, observando a flag *hasEvent*. Esta flag é modificada tanto pelo processo associado ao ator *GenStimulus* quanto ao processo associado ao ator *PlantDynamics*. Na linha 18, o procedimento *crane()* executa uma iteração do sistema. Entre as linhas 1 e 8 estão declaradas as variáveis que implementam o armazenamento para os sinais de entrada e saída do ator *CraneSystem*.

A ferramenta FASI determinou a validade do ator *CraneSystem*. Assim, um escalonamento para este ator é: (*posSensors*, *powerUp*, *SensorCheck*, *CheckPosDesired*, *CheckMinMaxPos*, *Emstop-Merge*, *CheckAlfa*, *ControlAlg*, *brakeCondition*, *isKnown*, *ControlBrake*, *VC-Merger*). Um total de 14 bits foram utilizados em cada um dos vetores de estado e presença de dados.

5.2.5 Resultados

Caracterizei a implementação no modelo SR do sistema de trenó,

determinando o tempo médio de resposta do sistema para qualquer conjunto de eventos de entrada. Obteve 11 μ s como valor médio, com 1 de desvio padrão, utilizando tanto a diretiva de otimização -O2 quanto a diretiva -O3. Como no estudo do codificador de voz, considerando um processador 16 vezes mais lento, teríamos 176 μ s como tempo de resposta aproximado.

A implementação do sistema necessitou de 3 kbytes de espaço de armazenamento para as instruções e 510 bytes de espaço de dados.

5.3 Considerações

O primeiro estudo de caso apresentou a captura de um sistema dominado pelo processamento de dados. Foi possível comparar uma implementação tradicional com outras baseadas em interação de atores. A primeira observação sobre este estudo de caso é que a penalidade do emprego do modelo PN com relação ao modelo SDF foi dominada pelo espaço necessário de armazenamento ao invés do tempo de execução. Note que este exemplo é uma situação particular do uso do modelo PN, pois trata-se de um sistema que pode ser capturado com escalonamento estático. Desta forma, para sistemas onde o modelo SDF não possui expressividade suficiente, a comparação do uso do modelo PN com um modelo de fluxo de dados com escalonamento dinâmico torna-se interessante.

A segunda observação sobre o estudo de caso da codificação de voz é que a otimização realizada sobre o código dos atores atômicos se mostrou muito mais importante na qualidade final da implementação que a escolha de um modelo computacional. Desta forma, pode-se concluir que a importância de se disponibilizar diferentes linguagens e técnicas para a elaboração e implementação dos atores atômicos é tão importante quanto a presença de vários modelos computacionais.

O segundo estudo de caso é um exemplo típico de sistema de controle. Sua implementação foi dominada por um conjunto de máquinas de estado, que reagem concorrentemente a eventos externos. O modelo SR se mostrou adequado para a captura deste exemplo. A implementação de máquinas de estado através de atores atômicos se mostrou adequada, pois algoritmos podem ser capturados nos estados e na transição entre estados.

A observação fundamental que se pode fazer sobre o segundo estudo de caso, é que o fator determinante na escolha de um modelo computacional foi a natureza da operação dos atores e sinais entre eles. Neste estudo de caso, os atores são máquinas de estado, isto é, atores com uma característica passiva que reagem a estímulos. Os sinais deste exemplo foram claramente classificados em periódicos e aperiódicos. Desta forma, antes mesmo de escrever qualquer linha de código, é possível realizar uma primeira separação de modelos computacionais em adequados ou não. Por exemplo, o modelo PN é certamente inapropriado para este estudo de caso, pois não contempla eficientemente atores passivos e sinais aperiódicos. Desta forma, pode-se concluir que é necessário estudar se um novo nível de abstração é útil entre os níveis de concepção e de interação de atores. A princípio, dois tipos de entidades abstratas devem estar presentes: funções e valores. Associado a estas entidades estão propriedades semânticas. A especificação neste nível deveria então ser analisada para se descobrir quais modelos computacionais são capazes de instanciar concretamente tais entidades abstratas, baseadas nas propriedades das entidades e características do modelo computacional.

CAPÍTULO 6 - CONCLUSÕES E TRABALHOS FUTUROS

“Tudo o que eu quero na vida é um punhado de livros, sonhos e vulvas.”

Henry Miller

6.1 Considerações Finais

Nesta tese abordei o projeto de software embarcado. Este tema é de importância fundamental, devido ao volume de sistemas que estão classificados nessa área. Embora a tecnologia eletrônica teve um desenvolvimento significativo e constante ao longo do tempo, as técnicas de projeto de software embarcado mais comumente empregadas utilizam ferramental cuja base foi desenvolvida a mais de 30 anos.

Elegi uma metodologia baseada na interação de atores como ponto de partida para o projeto de software. O ponto fundamental desta metodologia é elevar o nível de abstração de processos e primitivas de sincronização de baixo nível para um fundamentado sobre modelos computacionais precisamente definidos e determinísticos. Estes modelos capturam conceitos como concorrência e modelo de tempo de maneira mais abstrata. Dentre as vantagens desta abordagem, a facilidade na criação de soluções devido ao maior grau de expressividade disponível ao desenvolvedor é notória. Embora promissora e com sistemas comerciais sendo utilizados para desenvolvimento industrial, o projeto baseado em atores ainda requer vários aprimoramentos.

No capítulo 3, detalhei como produzir e caracterizar implementações de sistemas embarcados a partir de especificações baseadas em atores. Elegi uma arquitetura composta por um microprocessador executando um sistema operacional embarcado. Desenvolvi ou utilizei trabalhos de outros pesquisadores a fim de estabelecer uma técnica de tradução para quatro importantes modelos computacionais.

No capítulo 4, abordei um problema até hoje esquecido, mas de imensa relevância. Estudei, elaborei e implementei uma estratégia para determinar como

permitir que um desenvolvedor crie uma especificação baseada em atores, sem necessariamente torná-la específica a um modelo computacional. Através desta estratégia, é possível explorar diferentes soluções para uma especificação executável. Também é possível encontrar falhas na concepção de uma especificação executável, facilitando assim o processo de desenvolvimento.

No capítulo 5, apresentei resultados iniciais do emprego da metodologia de projeto baseada em atores sobre dois estudos de caso. Apliquei as técnicas desenvolvidas nos capítulos anteriores a fim de obter implementações. Analisei os resultados obtidos.

Na introdução deste texto, destaquei duas perguntas fundamentais que qualquer metodologia para auxiliar um projetista na escolha de modelos computacionais deveria responder:

1. Quais modelos computacionais posso utilizar ?
2. Qual é o melhor (conjunto de) modelo(s) ?

Mostrei nesta tese como responder a segunda pergunta: introduza o ambiente de atores em uma metodologia de projeto. Para tal, detalhei como traduzir uma especificação de atores em uma implementação em software embarcado e, como caracterizar a implementação a fim de ser possível a comparação de implementações distintas.

A primeira pergunta foi respondida através de uma estratégia e ferramenta de análise de especificações executáveis. Entretanto, a partir dos testes que realizei, percebi que a pergunta é respondida parcialmente. Por exemplo, no estudo do controle de trenó, embora os atores estivessem bem descritos para o modelo PN por exemplo, eles não podem ser utilizados conjuntamente com estes modelos pois não iriam capturar a funcionalidade desejada. Desta forma, fica claro que mais informações, não necessariamente ligadas ao código fonte de cada ator, devem ser recebidas e consideradas para determinar a validade entre um modelo computacional e uma especificação executável.

Finalmente, destaco os produtos obtidos através do desenvolvimento deste trabalho:

- Domínio de 6 modelos computacionais;
- Estudo e utilização de um SOE completo;
- Caracterização de implementações;
- Análise léxica, sintática e construção de estruturas de dados para representação de programas na linguagem Java. Ferramenta para análise destas estruturas;
- Proposta de uma estratégia de validação de especificações;
- Implementação da validação para 2 modelos computacionais;
- Especificação detalhada da tradução de 4 modelos computacionais;
- Implementação de 4 modelos computacionais em uma arquitetura mono-processada;
- Dois estudos de caso de natureza distinta, com especificações e respectivas implementações.

6.2 Trabalhos Futuros

A lista de trabalhos futuros é extensa e certamente irei omitir trabalhos relevantes. Divido-os em dois grupos de trabalhos: 1) captura; 2) implementação. O grupo de captura refere-se ao processo da transformação de especificações no nível de concepção para o nível de atores. O grupo de implementação refere-se a transformação da especificação baseada em atores em uma implementação em alguma arquitetura alvo.

No grupo de captura, destaco os seguintes trabalhos:

- Desenvolvimento de um nível de abstração baseado em entidades abstratas, conforme descrito no ítem 5.3 (1 doutorado);
- Contemplar o desenvolvimento de especificações executáveis com múltiplas linguagens (e ferramentas associadas) para descrição de atores e múltiplos modelos computacionais (n mestrados);
- Desenvolvimento de um arcabouço para precisamente descrever a semântica da interação entre modelos computacionais (1 belo doutorado);

- Implementação de ferramentas de validação para especificação executáveis, que levem em conta informações da execução. Por exemplo, uma ferramenta de depuração detalhada (1 doutorado muito trabalhoso);
- Estudo de modelos computacionais que permitem que uma topologia modifique sua topologia durante a execução (n doutorados).

Trabalhos relevantes ao grupo de implementação são:

- Considerar arquiteturas com múltiplas unidades de processamento. A granularidade pode ser tanto grossa quanto fina. Como consequência, técnicas para implementar modelos computacionais em arquiteturas distribuídas devem ser estudadas (n doutorados e mestrados);
- Caracterização da implementação de uma especificação com relação a potência consumida. Este é um parâmetro de grande importância para determinadas classes de sistemas embarcados (n mestrados);
- Implementações mais eficientes de modelos computacionais que não possuem escalonamento estático (n mestrados);
- Implementação de especificações cuja topologia varia durante a execução (n mestrados).

ANEXO A - O AMBIENTE PTOLEMY II

A.1 Introdução

O ambiente Ptolemy II [DAV01] é um sistema para a criação de especificações executáveis baseado no paradigma de interação entre atores. Este projeto está sendo desenvolvido na Universidade da Califórnia em Berkeley desde 1990. Inicialmente voltado para o desenvolvimento de sistemas (hardware e software) para processamento digital de sinais, atualmente seu principal objetivo é possibilitar o estudo de diferentes modelos computacionais e suas interações.

A linguagem de programação Java [JOY00] é utilizada na implementação do ambiente. Desta forma, ele é constituído por um conjunto de classes organizadas em pacotes. O núcleo do ambiente é composto de dois pacotes: *Kernel* e *Actor*. Estes dois pacotes implementam respectivamente uma sintaxe e semântica abstrata. Nenhum destes pacotes implementa uma semântica concreta para a execução de uma especificação. Um modelo computacional é implementado estendendo-se o pacote *Actor*. O ambiente também contém uma série de pacotes auxiliares, como um sistema de expressões, tipos de dados, biblioteca de componentes e interface gráfica.

A.2 O pacote *Kernel*

O pacote *Kernel* é composto por um conjunto de classes para implementar e manipular topologias. Uma topologia é constituída por **entidades**⁵² e **relações**⁵³. A Figura 79 apresenta um exemplo de topologia não-hierárquica.

⁵² Classe *Entity*, *ComponentEntity* e *CompositeEntity* do ambiente.

⁵³ Classe *Relation* e *ComponentRelation* do ambiente.

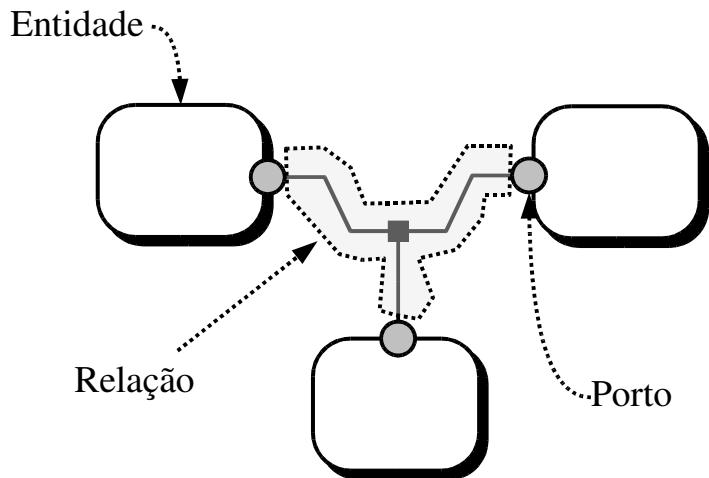


Figura 79 Topologia não hierárquica.

Cada entidade pode conectar vários portos entre si. Um grafo matemático é um grafo que pode conectar entre si os elementos. Já em uma topologia matemática, cada entidade possuiria vários portos, que representam as entradas e saídas do grafo e cada uma possuindo exatamente um porto. No caso de um grafo direcionado, cada entidade possuiria dois portos, uma para arcos de entrada e outro para arcos de saída.

Uma topologia pode ser hierárquica, isto é, uma entidade⁵⁴ pode conter uma sub-topologia. Uma entidade que não contenha uma sub-topologia é denominada de **atômica**.

A conexão entre as hierarquias se dá através de portos associados à entidade hierárquica. O nível contido na entidade se conecta aos seus portos, e a entidade hierárquica pode ser utilizada em uma topologia qualquer. A Figura 80 apresenta um exemplo de topologia hierárquica, onde E0 e E1 são entidades hierárquicas.

54 Classe *Port* e *ComponentPort* do ambiente

55 Uma entidade hierárquica é implementada através da classe *CompositeEntity* do ambiente.

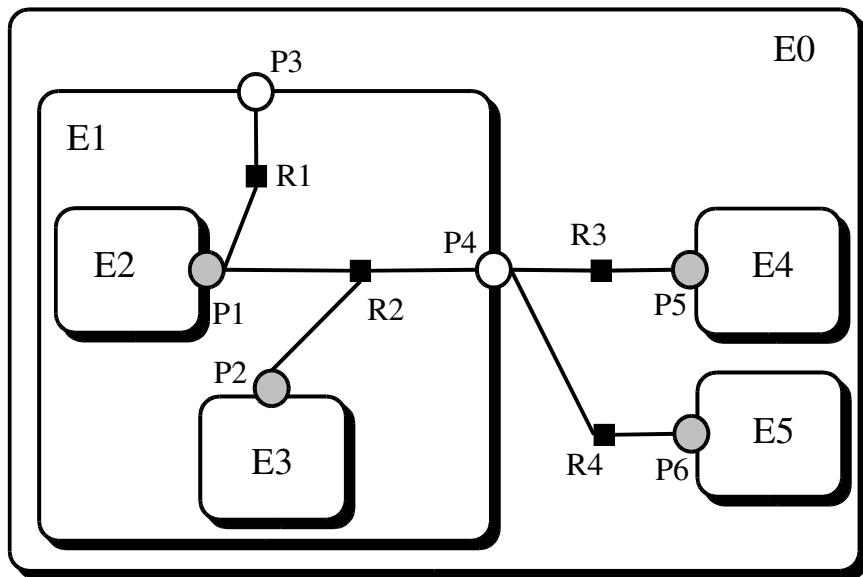


Figura 80 Exemplo de topologia hierárquica.

Os portos P3 e P4 da Figura 80 são denominados de **transparentes**, enquanto que os demais são denominados de **opacos**. Esse conceito também se aplica a entidades, como será mostrado mais adiante.

Um porto é opaco quando a entidade que o contém for opaca, e é transparente quando a entidade que o contém for transparente. Quando um porto for opaco, conexões com uma possível sub-topologia são invisíveis. Já um porto transparente permite que algoritmos percorram todas as conexões associadas a ele. Por exemplo, existe um procedimento no ambiente que retorna uma lista de todos os portos conectados, independentemente do nível de hierarquia. No caso do porto P1 da Figura 80, esse procedimento retornaria os portos P2, P5 e P6. Portos transparentes não são retornados. Quando o procedimento atinge o porto P4, através da relação R2, o procedimento continua pesquisando, através das relações R3 e R4.

A.3 O pacote *Actor*

O pacote *Actor* é composto por um conjunto de classes que implementam uma semântica abstrata. Neste pacote, entidade e porto são estendidos e passam a

ser chamados de **ator**⁵⁶ e **porto de entrada/saída**⁵⁷ (ES) respectivamente. A razão dessa extensão é prover suporte a duas características necessárias para a implementação de um modelo computacional: comunicação de dados entre os atores e controle da execução.

Comunicação de Dados

A Figura 81 ilustra o processo de troca de dados.

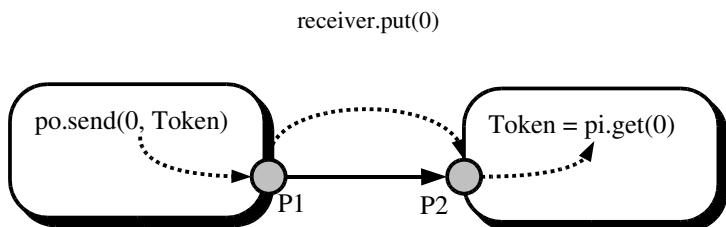


Figura 81 Exemplo de troca de dados entre atores.

Dois portos de E/S são utilizados: P1 e P2, onde P1 é um porto de saída (envio de dados) e P2 um porto de entrada (recebimento de dados). Os portos direcionais são dotados de métodos para troca de dados. Entre eles o método *send()* é utilizado por um porto de saída para enviar dados aos portos conectados, e o método *get()* é utilizado por um porto de entrada para receber dados. No ambiente Ptolemy II, os dados são transmitidos encapsulados em um objeto denominado de *Token*.

A função do método *send()* é transferir os dados para um elemento denominado de **receptor** (*receiver*). Cada conexão com um porto de entrada possui associado um receptor. Na Figura 81, temos uma conexão entre portos. Cada conexão também é denominada de **canal**. Quando um porto possui mais de um canal, deve-se especificar no método *send()* e *get()* por qual canal o dado deve ser enviado ou recebido. Uma vez que os dados estejam presentes no respectivo receptor, o ator de destino pode executar o método *get()* e obter os dados.

No pacote *Actor*, o receptor não é uma classe concreta, mas sim uma

56 Classes *AtomicActor* (atômico), *CompositeActor* (hierárquico).

57 Classe *IOPort*.

interface. O receptor é implementado como parte do modelo computacional, pois diferentes modelos definem diferentemente a natureza da comunicação entre atores. Por exemplo, entidades em um modelo de fluxo de dados comunicam-se através de filas. As filas são implementadas em um receptor. Já no modelo Eventos Discretos, o receptor deve enviar os dados para uma fila global.

Controle da Execução

Um elemento denominado de **diretor** é responsável por controlar a execução de uma ou um conjunto de topologias. Um diretor contém cinco métodos, também chamados de métodos de execução, que são ativados por uma duração e ordem pré-determinada. São eles:

- *initialize()*: chamado uma única vez durante a execução, antes de todos os outros métodos;
- *prefire()*: primeiro a ser chamado, uma única vez, durante uma iteração da execução;
- *fire()*: pode ser chamado várias vezes em uma mesma iteração, após o método *prefire()*;
- *postfire()* : similar ao método *prefire()*. Ele é o último método a ser chamado em uma iteração;
- *wrapup()*: chamado uma única vez ao final da execução.

Assim, uma execução é composta por um número finito de iterações compostas dos métodos *prefire()*, *fire()* e *postfire()*. A funcionalidade de cada um desses métodos depende de cada modelo computacional. Um ator também possui os cinco métodos de execução. O desenvolvedor da especificação deve utilizá-los a fim de implementar a função do ator.

Um diretor deve estar sempre associado a um ator hierárquico. Quando isto acontece, esse ator passa a ser opaco⁵⁸ (entidade opaca). Caso contrário ele é um ator hierárquico transparente. Em uma topologia com mais de um ator hierárquico, é

58 Um ator atômico é sempre considerado opaco.

possível a utilização de mais de um diretor. A Figura 82 ilustra esta situação.

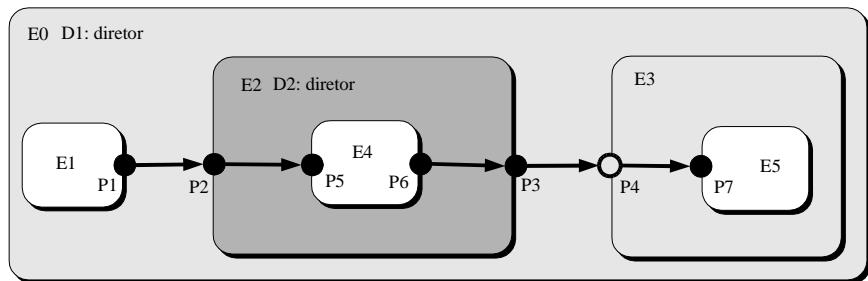


Figura 82 Exemplo de topologia hierárquica para ilustrar múltiplos diretores.

O diretor D1 está associado ao ator E0, que representa o nível mais alto da hierarquia⁵⁹. Nesse nível, existem três atores: E1, E2, E3. O ator E1 é atômico. O ator E3 é hierárquico e transparente. Quem controla a execução da sub-topologia do ator E3 é o diretor D1. Já o ator E2 é hierárquico e opaco, pois tem o diretor D2 associado. Para o diretor D1, o ator E2 é atômico.

A.4 Anatomia de Atores

A.4.1 Atores Atômicos

Uma das tarefas de um desenvolvedor de especificações executáveis é a implementação de atores atômicos. Para tal, utiliza-se os comandos da linguagem Java e os métodos das classes do ambiente Ptolemy II. Os principais métodos das principais classes do pacote *Actor* são:

Classe *Director* (diretor)

- *double getCurrentTime()* : retorna o valor do instante tempo;
- *double getNextIterationTime()* : retorna o valor do próximo instante de tempo;
- *void fireAt(Actor actor, double time)* : escalona o próximo ativamento de um

⁵⁹ Também chamado de nível 0.

ator para um dado instante de tempo. O parâmetro *actor* é o ator a ser executado e o parâmetro *time* é o valor do instante de tempo do ativamento do ator;

- *void fireAtCurrentTime(Actor actor)* : escalona o próximo ativamento de um ator para o atual instante de tempo;
- *void fireAtRelativeTime(Actor actor, double time)* : escalona o próximo ativamento de um ator com base no valor do instante de tempo. O parâmetro *actor* é o ator a ser executado. O parâmetro *time* especifica um incremento a partir do instante de tempo da especificação.

Classe *IOPort* (porto de E/S)

- *void broadcast(Token token)*: envia um token para todos os receptores conectados a este porto de E/S;
- *void broadcast(Token[] tokenArray, int vectorLength)*: envia um vetor de tokens para todos os receptores conectados. O primeiro parâmetro é um vetor de tokens e o segundo a quantidade de tokens deste vetor a ser enviada;
- *void broadcastClear()*: indica para todos os receptores conectados a não existência de um token;
- *Token get(int channelIndex)*: recebe um token do canal especificado pelo parâmetro;
- *Token[] get(int channelIndex, int vectorLength)*: recebe um vetor de tokens do canal especificado pelo primeiro parâmetro. O segundo parâmetro indica a quantidade de tokens a ser obtida;
- *double getCurrentTime(int channelIndex)*: retorna o valor do instante de tempo associado ao canal especificado pelo parâmetro;
- *int getWidth()*: retorna a largura do porto, isto é, quantos canais estão conectados a ele;
- *boolean hasRoom(int channelIndex)*: retorna verdade se for possível armazenar um token no canal especificado pelo parâmetro *channelIndex*;
- *boolean hasToken(int channelIndex)*: retorna verdade se existir um token no

canal especificado pelo parâmetro *channelIndex*;

- *boolean hasToken(int channelIndex, int tokens)*: retorna verdade se existir a quantidade de tokens especificada pelo parâmetro *tokens* no canal indicado pelo parâmetro *channelIndex*;
- *boolean isKnown()*: retorna verdade se para todos os canais conectados ao porto, o estado da presença ou não de um token for conhecido;
- *boolean isKnown(int channelIndex)*: retorna verdade se, para o canal especificado pelo parâmetro *channelIndex*, o estado da presença ou não de um token for conhecido;
- *void send(int channelIndex, Token token)*: envia o token especificado pelo segundo parâmetro para o receptor conectado ao canal especificado pelo primeiro parâmetro;
- *void send(int channelIndex, Token[] tokenArray, int vectorLength)*: envia um vetor de tokens especificado pelo segundo parâmetro para o receptor conectado ao canal especificado pelo primeiro parâmetro. O terceiro parâmetro indica a quantidade de tokens do vetor a ser enviada;
- *void sendClear(int channelIndex)*: modifica o estado de todos os receptores conectados para indicar a ausência de tokens.

Classe *AtomicActor* (ator atômico)

- *Director getDirector()*: retorna o diretor que controla a execução do ator;

Interface *Receiver*

A interface *Receiver*, embora não utilizada diretamente pelo desenvolvedor, possui grande importância. A semântica de comunicação de um modelo computacional é criada através da sua implementação. Seus principais métodos são:

- *Token get()*: remove um token desse receptor;
- *Token[] getArray (int numberoftokens)*: remove um vetor de tokens desse receptor. A dimensão do vetor é determinada pelo primeiro parâmetro;

- *boolean hasRoom()*: retorna verdade se for possível armazenar um token nesse receptor;
- *boolean hasRoom(int numberOfTokens)*: retorna verdade se for possível armazenar nesse receptor a quantidade de tokens especificados pelo parâmetro;
- *boolean hasToken()*: retorna verdade se esse receptor conter pelo menos um token;
- *boolean hasToken(int numberOfTokens)*: retorna verdade se esse receptor conter pelo menos a quantidade de tokens especificados pelo parâmetro *numberOfTokens*;
- *boolean isKnown()*: retorna o estado do conhecimento da presença de tokens nesse receptor;
- *void put(Token token)*: armazena um token nesse receptor;
- *void putArray(Token[] tokenArray, int numberOfTokens)*: armazena um vetor de tokens nesse receptor. O segundo parâmetro especifica a quantidade de tokens a serem armazenadas;
- *void setAbsent()*: indica que esse receptor não contém nenhum token.

A Figura 83 apresenta um código fonte simplificado de um ator atômico, ilustrando seus métodos de execução.

```

1- public class AtorExemplo extends TypedAtomicActor {
2-
3-     public TypedIOPort entrada;
4-     public TypedIOPort saida;
5-
6-     public Parameter par;
7-
8-     public AtorExemplo(TypedAtomicActor container, String name)
9-         throws NameDuplicationException, IllegalActionException {
10-        super(container, name);
11-
12-        entrada = new TypedIOPort(this, "entrada", true, false);
13-        entrada.setTypeEquals(BaseType.INT);
14-
15-        saida = new TypedIOPort(this, "saida", false, true);
16-        saida.setTypeEquals(BaseType.DOUBLE);
17-
18-        par = new Parameter(this, "parametro1", new IntToken(1));
19-    }
20-
21-    public void initialize() throws IllegalActionException {
22-        IntToken it = (IntToken) par.getToken();
23-        _var = it.intValue();
24-    }
25-
26-    public boolean prefire() throws IllegalActionException {
27-        ....
28-        return super.prefire();
29-    }
30-
31-    public void fire() throws IllegalActionException {
32-        IntToken it = (IntToken) entrada.get(0);39-
33-        saida.broadcast(DoubleToken.convert(it));
34-    }
35-
36-    public boolean postfire() throws IllegalActionException {
37-        ....
38-        return super.postfire();
39-    }
40-
41-    public void wrapup() throws IllegalActionException {
42-        super.wrapup();
43-        ....
44-    }
45-    private int _var;
46- }
```

Figura 83 Exemplo de código fonte de um ator atômico.

Na linha 1, a classe *TypedAtomicActor* é derivada. Esta classe define os métodos básicos para um ator atômico. Em certos modelos computacionais, novas classes de atores atômicos são criadas estendendo-se a classe *TypedAtomicActor*.

Entre as linhas 3 e 6, dois portos de E/S e um parâmetro são declarados como variáveis públicas. Entre as linhas 8 e 19 está implementado o construtor da classe. Na linha 12, um porto de entrada⁶⁰ é criado. Em seguida, o tipo do porto é determinado como sendo um inteiro. O porto de saída é criado de maneira similar nas linhas 15 e 16. Na linha 18, o parâmetro do ator é criado e inicializado como tendo tipo inteiro com valor inicial igual à 1 (terceiro parâmetro). Os métodos de execução implementados são:

- *initialize()* (linha 21 à 24): utilizado para obter e armazenar o valor do parâmetro do ator em uma variável interna;
- *prefire()* (linha 26 à 29): nenhuma função em particular. Chama o método da classe pai, retornando o devido valor. No método *prefire()*, um valor *false* de retorno indica que o ator não está apto a ser executado na atual iteração;
- *fire()* (linha 31 à 34): Na linha 38, um *token* é lido do porto de entrada. Na linha 40, o valor do *token* é convertido de inteiro para real, e enviado utilizando o método *broadcast()*;
- *postfire()* (linha 36 à 39): igual ao método *prefire()*;
- *wrapup()* (linha 41 à 44): nenhuma utilidade neste exemplo.

A.4.2 Atores hierárquicos

As Figuras 84 e 85 apresentam capturas de tela da interface gráfica do ambiente Ptolemy II. Elas apresentam um exemplo de especificação contendo atores hierárquicos.

⁶⁰O terceiro parâmetro indica se é um porto de entrada. O quarto parâmetro indica se é um porto de saída.

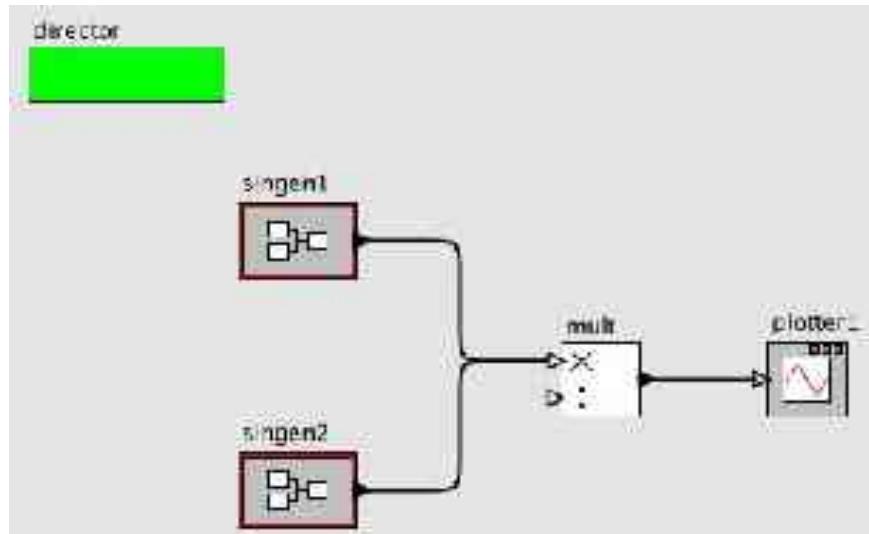


Figura 84 Nível 0 de uma especificação executável.

Diz-se que a topologia da Figura 84 é composta quatro instâncias de atores: duas instâncias de um mesmo ator hierárquico *Singen* (*singen1* e *singen2*), uma instância do ator atômico *MultiplyDivide* (*mult*) e uma instância do ator atômico *SequencePlotter* (*plotter1*). Uma instância está para um objeto assim como um ator está para uma classe. Por exemplo, diferentes instâncias de um mesmo ator podem apresentar valores distintos para parâmetros. Na interface gráfica do ambiente Ptolemy II, um ator hierárquico é distinguido dos demais por um ícone específico. A topologia da Figura 84 apresenta um modelo computacional associado, através da presença de um diretor. Esta topologia não é contida em nenhum ator hierárquico, isto é, é uma topologia de nível 0. A Figura 85 apresenta a topologia do ator hierárquico *Singen*.

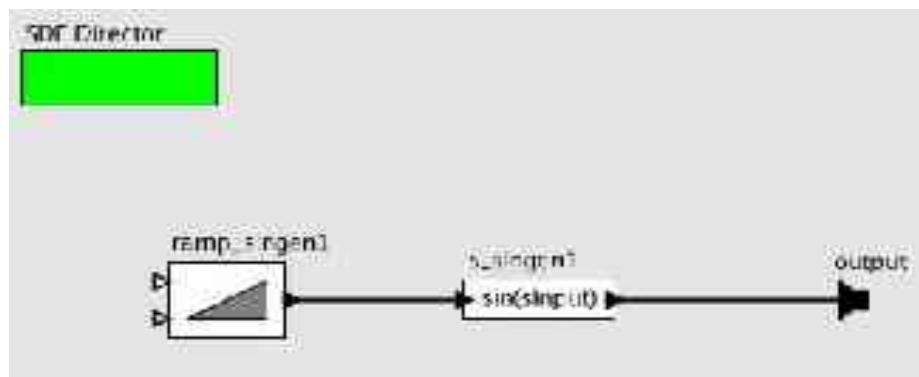


Figura 85 O ator hierárquico Singen.

A topologia da Figura 85 contém duas instâncias de atores atômicos. Um modelo computacional é associado a esta topologia. Assim, as duas instâncias do ator *Singen* na topologia da Figura 84 são consideradas opacas.

ANEXO B – RESUMO SOBRE TEORIA DE ORDENS

Neste anexo descrevo brevemente algumas definições e resultados sobre teoria de ordens. Um texto introdutório sobre tal teoria encontra-se em [DAV90].

Um **conjunto parcialmente ordenado** (*poset*) é um conjunto **S** com uma relação parcial de ordem \subseteq . Para quaisquer x, y , e z pertencentes a **S**, \subseteq satisfaz:

- $x \subseteq x$ (reflexiva);
- $x \subseteq y$ e $y \subseteq x$ implica que $x = y$ (antisimétrica);
- $x \subseteq y$ e $y \subseteq z$ implica que $x \subseteq z$ (transitiva);

O **limite inferior** de um *poset*, representado por \perp , é um membro de **S** tal que $\perp \subseteq s$ para todo $s \in S$. Um *poset* com um limite inferior é denominado de *poset pontual*.

Um **limite superior** de um conjunto **T** é um elemento **u** tal que $t \subseteq u$ para todos $t \in T$. O menor limite superior de um conjunto **T**, representado por ∇T , é um elemento **l** tal que $l \subseteq u$ para todos limites superiores **u**. O menor limite superior de um conjunto, se existir, é único.

Uma **cadeia** é um conjunto totalmente ordenado **C**, isto é, para todo $x, y \in C$, ou $x \subseteq y$ ou $y \subseteq x$. Um *poset*, onde toda cadeia em **S** possui um menor limite superior em **S**, é denominado de **conjunto parcialmente ordenado completo (CPO)**. O menor limite superior de uma cadeia finita sempre existe e é seu maior elemento. Um *poset* apenas com cadeias finitas é um CPO.

Se D_1 e D_2 são CPOs, então $D_1 \times D_2$ é um CPO sob a ordem:

$$(x_1, x_2) \subseteq (y_1, y_2) \text{ se e somente se } x_1 \subseteq y_1 \text{ e } x_2 \subseteq y_2$$

$$\text{e se } x^1 = (x^1_1, x^1_2) \text{ } x^2 = (x^2_1, x^2_2), \dots$$

$$\nabla\{x^1, x^2, \dots\} = (\nabla\{x^1_1, x^2_1, \dots\}, \nabla\{x^1_2, x^2_2, \dots\}).$$

Uma função $f:D \rightarrow E$ entre posets D e E é **monotônica** se para todo $x, y \in D$ tal que $x \leq y$, $f(x) \leq f(y)$.

Uma função $f:D \rightarrow E$ entre os CPOs D e E é **contínua** se para todas as cadeias $C \in D$, $f(\nabla C) = \nabla f(C)$.

Uma função contínua é monotônica. Uma função monotônica cujo domínio é um CPO com apenas cadeias finitas é contínua. A composição de duas funções contínuas também é contínua. A composição de duas funções monotônicas também é monotônica.

Seja D, E e F CPOs. Se $f:D \rightarrow E$ e $g:E \rightarrow F$ contínuas, então $f \times g$ também é contínua.

Seja D um poset, $f:D \rightarrow D$ uma função e $x \in D$:

- se $f(x) \leq x$, então x é um **ponto prefixo**;
- se $f(x) = x$, então x é um **ponto fixo**;
- se $f(x) = x$, então x é um **ponto fixo**; x é um ponto prefixo e $x \leq p$, para todo ponto prefixo p, então x é o **menor ponto prefixo**.
- se x é um ponto fixo e $x \leq y$ para todo ponto fixo y, então x é o **menor ponto fixo**.

REFERÊNCIAS BIBLIOGRÁFICAS

- [AGH86] G. A. Agha, **ACTORS: A Model of Concurrent Computation in Distributed Systems**, The MIT Press Series in Artificial Intelligence, MIT Press, 1986.
- [APP98] A. W. Appel, **Modern Compiler Implementation in ML**, ISBN 19980-531-58274-1, 1998.
- [BAS01] T. Basten, J. Hoogerbrugge, **Efficient Execution of Process Networks**, Proceedings of the Conference on Communicating Process Architectures, IOS Press, 2001.
- [BEN03] A. Benveniste, et al, **The Synchronous Languages 12 Years Later**, Proceedings of the IEEE, 91(1), 2003.
- [BEN96] J. Banks, et al, **Discrete event System Simulation**, ISBN 0131446797, 1996.
- [BER84] G. Berry, L. Cosserat, **The ESTEREL synchronous programming Language and its mathematical semantics**, TR 327, INRIA, 1984.
- [BHA02] B. Bhattacharya, S. S. Bhattacharyya, **Parameterized dataflow modeling for DSP systems**, IEEE Transactions on Signal Processing, 49(10), 2001.
- [BHA94] S. S. Bhattacharyya, **Compiling Dataflow Programs for Digital Signal Processing**, UCB/ERL Technical Memorandum M94/52, Dept. EECS, University of California at Berkeley, 1994.
- [BOD97] R. Bodík, R. Gupta, M. L. Soffa, **Refining data flow information using infeasible paths**, Proceedings 5th ACM SIGSOFT international symposium on Foundations of software engineering, 1997.

[DAV01] J. Davis II, et al, **Ptolemy II – Heterogeneous concurrent modeling and design in Java**, Technical Memorandum M01/12, Dept. EECS, University of California at Berkeley, 2001.

[DAV90] B. A. Davey, H. A. Priestley, **Introduction to Lattices and Order**, ISBN 19900521365848, 1990.

[DEN74] J. B. Dennis, **First version of a dataflow procedure language**, Lecture Notes in Computer Science, 19, 1974.

[EDW00] S. A. Edwards, **Languages for Digital Embedded Systems**, ISBN 2000079237925X, 2000.

[EDW94] S. A. Edwards, **An Esterel compiler for a synchronous/reactive development system**, Report ERL M94/43, Dept. EECS, University of California at Berkeley, 1994.

[EDW97] S. A. Edwards, **The Specification and Execution of Synchronous Reactive Systems**, Report ERL M97/31, Dept. EECS, University of California at Berkeley, 1997

[EKE03] J. Eker, J. W. Janneck, **CAL Language Report: specification of the CAL actor language**, ERL Technical Memo UCB/ERL M03/48, Dept. EECS, University of California at Berkeley, 2003.

[ETS96] European Telecommunication Standards Institute, **Digital Cellular Telecommunications System: Enhanced Full Rate (EFR) spec**, 1996.

[ETS96b] European Telecommunication Standards Institute, **12200 bits/s Speech Coded for Enhanced Full Rate Speech Traffic Channels**, 1996.

[FRI01] L. F. Friedrich, et al, **A Survey of Configurable, Component-Based**

Operating Systems for Embedded Applications, IEEE Micro, 21(3), 2001.

[GOR00] G. Gorla, et al, **System Specification Experiments on a Common Benchmark**, IEEE Design & Test of Computers, 17(3), 2000.

[GOR94] M. J. C. Gordon, **Programming Language Theory and Implementation**, ISBN 19940137304099, 1994.

[HAL91] N. Halbwachs, et al, **The synchronous data flow programming language LUSTRE**, Proceedings of the IEEE, 79(9), 1991.

[HEW77] C. Hewitt, H. Baker, **Actors and continuous functionals**, Proceedings of Working Conference on Formal Description of Programming Concepts, 1977.

[HOA78] C. A. R. Hoare, **Communicating Sequential Processes**, Communications of the ACM, 21(8), 1978.

[HOP79] J. E. Hopcroft, J. D. Ullman, **Introduction to automata theory, languages and computation**, ISBN 19790-201-02988-X, 1979.

[JAN03] A. Jantsch, **Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation**, ISBN 20031558609253, 2003.

[JAR97] K. Järvinen, **Enhanced Full Rate Speech Coded**, Proceedings ICASSP '97, 1997.

[JEU00] I. Jeukens, **Um Estudo sobre a Utilização de Modelos Computacionais para a Representação de Sistemas Digitais**, Dissertação de Mestrado, Escola Politécnica, Universidade de São Paulo, 2000.

[JOY00] B. Joy, et al, **Java (TM) Language Specification (2nd Edition)**, ISBN 20000201310082, 2000.

- [KAH74] G. Kahn, **The semantics of a simple language for parallel programming**, Information Processing, 1974.
- [KAH77] G. Kahn, D. B. MacQueen, **Coroutines and networks of parallel processes**, Information Processing, 1977.
- [KER88] B. Kernighan, D. Ritchie, **C Programming Language (2nd Edition)**, 19880131103628, 1988.
- [LEA96] D. Lea, **Concurrent Programming in Java: Design Principles and Patterns**, 19960201695812, 1996.
- [LEE01] E. A. Lee, **Computing for Embedded Systems**, IEEE Instrumentation and Measurement Technology Conference, 2001.
- [LEE87] E. A. Lee, D. G. Messerschmitt, **Static Scheduling of synchronous data flow programs for digital signal processing**, IEEE Transactions on Computer, 36 (1), 1987.
- [LEE95] E. A. Lee, T. M. Parks, **Dataflow process networks**, Proceedings of the IEEE, 75(9), 1995.
- [LEW95] B. Lewis, D. J. Berg, **Thread Primer: A Guide to Multithreaded Programming**, ISBN 19950134436989, 1995.
- [LI03] Q. Li, C. Yao, **Real-Time Concepts for Embedded Systems**, ISBN 20031578201241, 2003.
- [LIU01] J. Liu, **Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems**, Technical Memorandum UCB/ERL M01/41, University of California at Berkeley, 2001.
- [MAR96] J. M. R. Martin, **The Design and Construction of Deadlock-free**

Concurrent Systems, PhD Dissertation, School of Sciences, University of Buckingham, 1996.

[MAS03] A. J. Massa, **Embedded software development with eCos**, 20030-13-035473-2, 2003.

[MOS99] E. Moser, W. Nebel, **Case Study: System Model of Crane and Embedded Control**, Design, Automation and Test in Europe (DATE '99), 1999.

[MUL99] L. Muliadi, **Discrete Event Modeling in Ptolemy II**, MS Report, Dept. EECS, University of California at Berkeley, 1999.

[NEU04] S. A. Neuendorffer, **Actor-Oriented Metaprogramming**, Phd Report, Dept. EECS, University of California at Berkeley, 2004.

[NIE00] M. A. Nielsen, I. L. Chuang, **Quantum Computation and Quantum Information**, ISBN 20000521635039, 2000.

[PAR95] T. M. Parks, **Bounded Scheduling of Process Networks**, Phd. Dissertation, Dept. EECS, University of California at Berkeley, 1995.

[PIN95] J. L. Pino, S. S. Bhattacharyya, E. A. Lee, **A Hierarchical Multiprocessor Scheduling System for DSP Applications**, Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, vol.1, 1995.

[STR] <http://cag.csail.mit.edu/streamit/>, **StreamIt Language Specification : Version 2.0**, Massachusetts Institute of Technology.

[SYM98] N. Symth, **Communicating Sequential Processes Domain in Ptolemy II**, Report M98/70, Dept EECS, University of California at Berkeley, 1998.

[TAN01] A. Tanenbaum, **Modern Operating Systems**, ISBN 20010130313580, 2001.

[TEN00] D. Tennenhouse, **Proactive Computing**, Communications of the ACM, 43 (5), 2000.

[TEN91] R. D. Tennet, **Semantics of programming languages**, ISBN 19910138056072, 1991.

[WAN03] E. Wandeler, **Static Analysis of Actor Networks**, Technical Memorandum UCB/ERL M03/7, Swiss Federal Institute of Technology (Diploma Thesis), 2003.

[XIO02] Y. Xiong, **An extensible type system for Component-Based Design**, Technical Memorandum UCB/ERL M02/13, University of California at Berkeley, 2002.

APÊNDICE A – CARACTERIZAÇÃO DOS PROCEDIMENTOS DO AMBIENTE ECOS

Na tabela 8 está caracterizado, com relação ao tempo de execução, os principais procedimentos do sistema eCos para manipulação de processos e primitivas de sincronização. Todos os tempos são apresentados em micro-segundos.

Processos

Tabela 8 – Caracterização de processos.

Procedimento	Média	Valor Mínimo	Valor Máximo
Criação	6.87	0	19
Suspensão	0.1	0	1
Reativamento da execução	0.29	0	1
Atribuição de uma prioridade	0.13	0	1
Destrução	0.19	0	1
Troca de processo a ser executado	0.08	0	1

Mutex

Tabela 9 – Caracterização da primitiva mutex.

Procedimento	Média	Valor Mínimo	Valor Máximo
Iniciação	0.31	0	8
Obtenção	0.03	0	1
Liberação	0.31	0	2
Destrução	0	0	0
Liberação e Obtenção	1.09	0	4

Semáforo

Tabela 10 – Caracterização da primitiva semáforo.

Procedimento	Média	Valor Mínimo	Valor Máximo
Iniciação	0.41	0	12
Incremento	0.06	0	1
Decremento	0.09	0	2

<i>Procedimento</i>	<i>Média</i>	<i>Valor Mínimo</i>	<i>Valor Máximo</i>
Teste do valor	0	0	0
Destrução	0	0	0

Flag

Tabela 11 – Caracterização da primitiva flag.

<i>Procedimento</i>	<i>Média</i>	<i>Valor Mínimo</i>	<i>Valor Máximo</i>
Iniciação	0.63	0	19
Destrução	0.06	0	1
Atribuição de um valor	0.03	0	1
Leitura e bloqueio	0.06	0	1

APÊNDICE B – EXEMPLO DE RELATÓRIO DE OCUPAÇÃO E ESCALONAMENTO DE PROCESSOS

Abaixo apresento o relatório obtido da execução do exemplo da Figura 15. Para cada tarefa, está mostrado a seqüência de estados, indicando o momento de início e fim, bem como a duração do estado. Ao final, é indicado para cada intervalo de tempo, qual tarefa estava sendo executada.

```
Total clock ticks: 700

THREAD : Network Transmitter Thread
Blocked      5    -> 702        Dur 697
Running     702    -> 722        Dur 20
Blocked     722    -> 7010005   Dur 7009283
2.853067E-4% Running, 0.0% Ready, 99.99972% Blocked.
Average time between runs : NaN

THREAD : Estorva o processador
Blocked      5    -> 10023      Dur 10018
Running    10023 -> 21094      Dur 11071
Blocked    21094 -> 30023      Dur 8929
Running    30023 -> 41623      Dur 11600
Blocked    41623 -> 50022      Dur 8399
Running    50022 -> 61121      Dur 11099
Blocked    61121 -> 70023      Dur 8902
Running    70023 -> 82086      Dur 12063
Blocked    82086 -> 90023      Dur 7937
.....
Running    6990025    -> 7000022  Dur 9997
Ready 7000022    -> 7002347  Dur 2325
Blocked    7002347    -> 7010005  Dur 7658
59.515205% Running, 0.34121254% Ready, 40.14358% Blocked.
Average time between runs : 8097.652298850575

THREAD : Instrument Thread
Blocked      5    -> 688        Dur 683
Blocked    688    -> 7002350   Dur 7001662
Running    7002350    -> 7010005  Dur 7655
0.10920114% Running, 0.0% Ready, 99.8908% Blocked.
Average time between runs : NaN

THREAD : Thread B
Blocked      5    -> 2251348   Dur 2251343
Running    2251348    -> 2254094   Dur 2746
Blocked    2254094    -> 4363943   Dur 2109849
Running    4363943    -> 4370020   Dur 6077
Ready 4370020    -> 4384094   Dur 14074
Running    4384094    -> 4384099   Dur 5
Blocked    4384099    -> 6840979   Dur 2456880
Running    6840979    -> 6848223   Dur 7244
Blocked    6848223    -> 7010005   Dur 161782
0.22927247% Running, 0.20077033% Ready, 99.56996% Blocked.
Average time between runs : 1526934.3333333333
```

THREAD : Thread A

```

Blocked      5    -> 2420743 Dur 2420738
Running     2420743    -> 2422107 Dur 1364
Blocked     2422107    -> 4500683 Dur 2078576
Running     4500683    -> 4510020 Dur 9337
Ready 4510020    -> 4522121 Dur 12101
Running     4522121    -> 4522125 Dur 4
Blocked     4522125    -> 6681081 Dur 2158956
Running     6681081    -> 6690020 Dur 8939
Ready 6690020    -> 6703886 Dur 13866
Running     6703886    -> 6703890 Dur 4
Blocked     6703890    -> 7010005 Dur 306115
0.2802853% Running, 0.37042797% Ready, 99.34929% Blocked.
Average time between runs : 1065874.75

```

THREAD : pthread.00000800

```

Blocked      5    -> 728      Dur 723
Running     728    -> 735      Dur 7
Blocked     735    -> 7010005 Dur 7009270
9.9857345E-5% Running, 0.0% Ready, 99.9999% Blocked.
Average time between runs : NaN

```

THREAD : Network support

```

Blocked      5    -> 724      Dur 719
Running     724    -> 726      Dur 2
Blocked     726    -> 617327   Dur 616601
Running     617327   -> 617339   Dur 12
Blocked     617339   -> 1611990  Dur 994651
Running     1611990  -> 1612002  Dur 12
Blocked     1612002  -> 2644580  Dur 1032578
Running     2644580  -> 2644593  Dur 13
Blocked     2644593  -> 4702226  Dur 2057633
Running     4702226  -> 4702240  Dur 14
Blocked     4702240  -> 7010005  Dur 2307765
7.5606274E-4% Running, 0.0% Ready, 99.999245% Blocked.
Average time between runs : 1175365.75

```

THREAD : Network alarm support

```

Blocked      5    -> 689      Dur 684
Running     689    -> 700      Dur 11
Blocked     700    -> 511271   Dur 510571
Running     511271   -> 511280   Dur 9

```

.....

```

Running     6982150    -> 6982162 Dur 12
Blocked     6982162    -> 7010005 Dur 27843
0.020556347% Running, 2.853067E-4% Ready, 99.97916% Blocked.
Average time between runs : 387779.555555555556

```

THREAD : Idle Thread

```

Blocked      5    -> 737      Dur 732
Running     737    -> 10020    Dur 9283
Ready 10020 -> 21097    Dur 11077
Running     21097   -> 30020    Dur 8923

```

.....

```

Ready 6970020    -> 6982176 Dur 12156
Running     6982176    -> 6990022 Dur 7846
Ready 6990022    -> 7010005 Dur 19983
39.46498% Running, 60.524582% Ready, 0.010442225% Blocked.
Average time between runs : 12204.595375722543

```

IDLE Percentage : 39.61526

Running Intervals

- 689 Network alarm support 700- Dur 11
- 702 Network Transmitter Thread 722- Dur 20
- 724 Network support 726- Dur 2
- 728 pthread.00000800 735- Dur 7
- 737 Idle Thread 10020- Dur 9283
- 10023 Estorva o processador 21094- Dur 11071
- 21097 Idle Thread 30020- Dur 8923
- 30023 Estorva o processador 41623- Dur 11600
- 41625 Idle Thread 50020- Dur 8395
- 50022 Estorva o processador 61121- Dur 11099
- 61123 Idle Thread 70020- Dur 8897

-
- 6881133 Idle Thread 6890010- Dur 8877
- 6890012 Estorva o processador 6902189- Dur 12177
- 6902192 Idle Thread 6910014- Dur 7822
- 6910016 Estorva o processador 6921180- Dur 11164
- 6921183 Idle Thread 6930011- Dur 8828
- 6930014 Estorva o processador 6941987- Dur 11973
- 6941993 Idle Thread 6950013- Dur 8020
- 6950015 Estorva o processador 6961273- Dur 11258
- 6961276 Idle Thread 6970020- Dur 8744
- 6970022 Estorva o processador 6980022- Dur 10000
- 6982150 Network alarm support 6982162- Dur 12
- 6982176 Idle Thread 6990022- Dur 7846
- 6990025 Estorva o processador 7000022- Dur 9997
- 7002350 Instrument Thread 7010005- Dur 7655

As tarefas “Network Alarm Support”, “pthread.00000800” e “Network Support” são criadas para executar funcionalidades básicas do sistema eCos. A tarefa “Idle Thread” não efetua nenhum cálculo e é empregada quando não há nenhuma tarefa a ser executada.

APÊNDICE C - A FERRAMENTA FASI

A ferramenta de FASI foi implementada na linguagem Java. Ao total, a ferramenta contém 32114 linhas de código, sem incluir a análise léxica e semântica. Os pacotes que integram a ferramenta são:

- *db*: contém um conjunto de classes para armazenar e recuperar as estruturas de dados criadas a partir de uma especificação executável;
- *FlowGraph*: contém as estruturas e métodos de construção de um CFG. Os subpacotes deste pacote são:
 - *EdgeInfo*: utilizado para anotar um vértice do grafo com informações adicionais;
 - *Nodes*: contém todos os diferentes tipos de nó de um CFG;
 - *SSA*: implementa os algoritmos para transformar um CFG em uma representação SSA equivalente;
 - *StatementInfo*: utilizado para anotar certos nós de um CFG com informações adicionais;
 - *Transformations*: algoritmos de transformação sobre um CFG;
- *gui*: possui as classes responsáveis por integrar a ferramenta ao ambiente Ptolemy II e apresentar os resultados de maneira visual;
- *JavaGrammar*: implementa o analisador léxico e sintático de uma classe em linguagem Java. Possui o seguinte subpacote:
 - *TreeScanners*: contém um conjunto de algoritmos que processam uma árvore sintática de maneira top-down.
- *MethodGraph*: implementa um GM, contendo estrutura de dados e transformações;
- *MoC*: algoritmos e métodos auxiliares para a implementação de algoritmos de análise de um modelo computacional;
- *preprocess*: implementa a etapa de preprocessamento;
- *sdf*: implementa os algoritmos de análise para o modelo computacional SDF;
- *simplification*: algoritmos para simplificar um CFG;
- *sr*: algoritmos de análise para o modelo computacional SR;

- *SymbolTable*: implementa as tabelas de símbolos para os membros e variáveis de uma classe em linguagem Java;
- *test*: possui um conjunto de testes para a ferramenta;
- *util*: contem classes que implementam funções auxiliares.

As Figuras 86 e 87 ilustram um exemplo de especificação com duas topologias. A topologia de nível 0 é específica ao modelo DE. O ator hierárquico *CTSubsystem* é específico ao modelo CT.

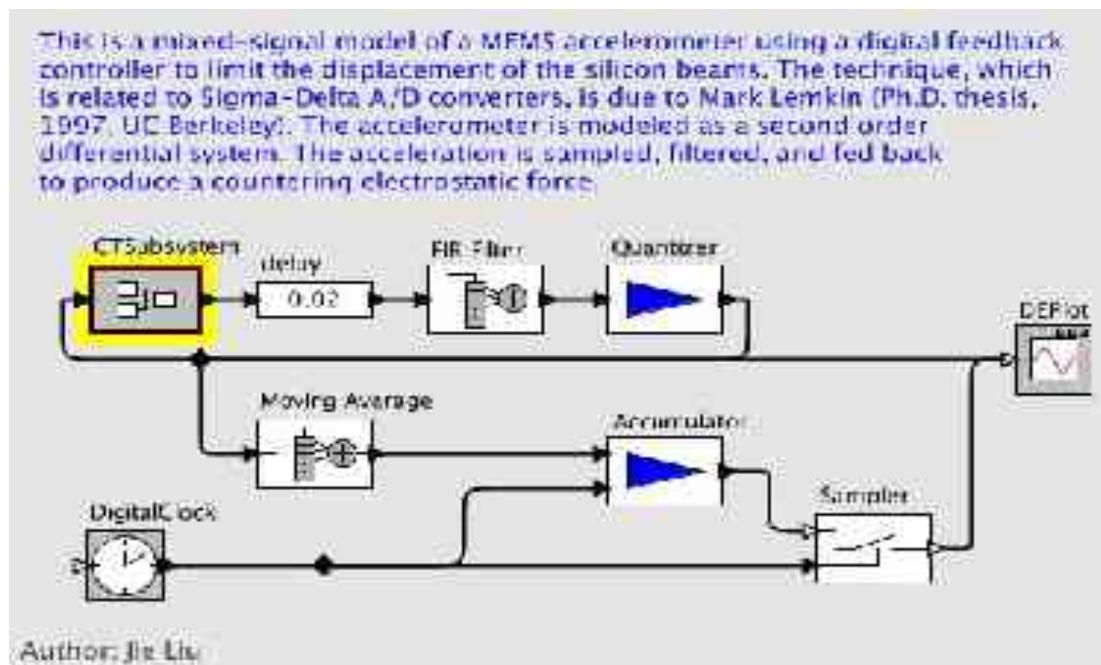


Figura 86 Topologia de nível 0.

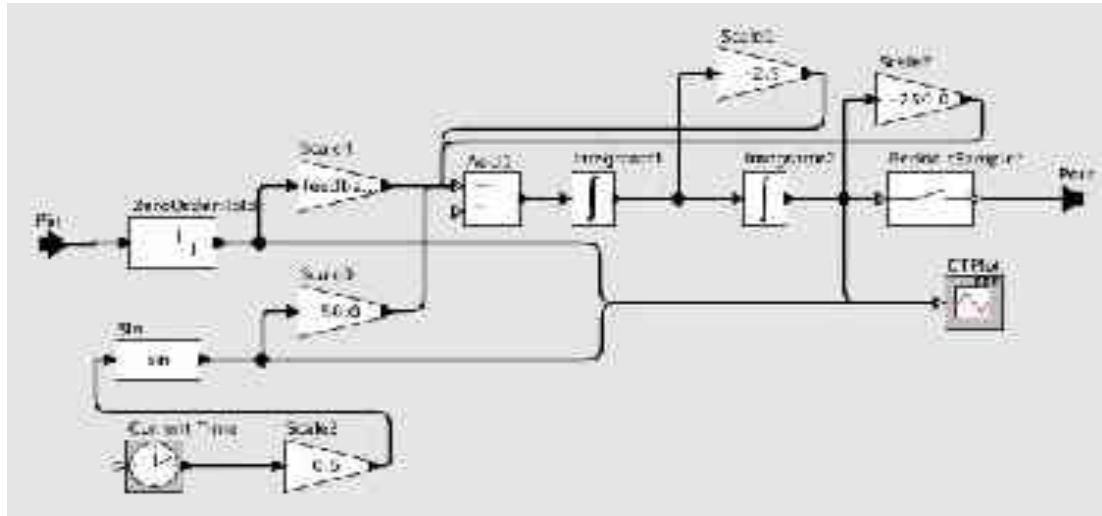


Figura 87 Topologia do ator CTSubsystem.

A Figura 88 apresenta o primeiro tipo de janela gráfica produzida pela ferramenta FASI. Trata-se da árvore de hierarquia de topologias de uma especificação. Também é listado para cada ator hierárquico os atores atômicos contidos. Associado a todo ator está a lista de modelos computacionais válidos encontrados pela ferramenta.

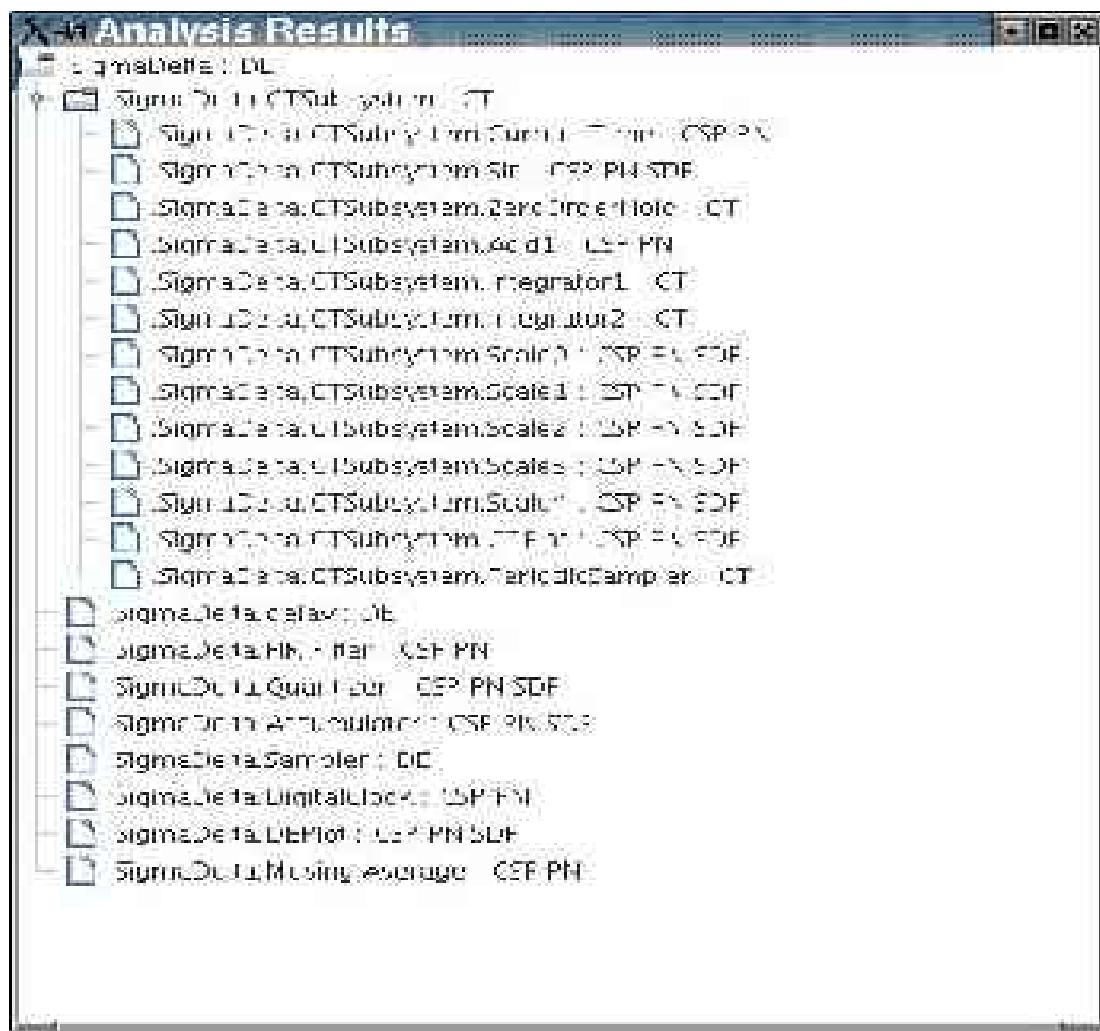


Figura 88 Resultado da análise: árvore de atores e seus modelos válidos.

O segundo tipo de janela gerada pela ferramenta contem uma lista das mensagens produzidas pela ferramenta durante o processo de análise. Três tipos de mensagens podem ser geradas: informação, aviso ou erro. É identificado a que ator a mensagem se refere, em que etapa da análise ela foi produzida e um pequeno texto explicativo. A Figura 89 apresenta a janela produzida para a especificação das Figura 86 e 87.

Figura 89: a janela de mensagens da ferramenta FASI.

Para cada mensagem, é possível localizar na interface gráfica do ambiente Ptolemy II o ator, e caso seja atômico, a construção do código fonte que produziu a mensagem. A Figura 90 apresenta o terceiro e último tipo de janela produzida pela ferramenta FASI. Trata-se do código fonte de um ator atômico, ilustrando uma construção que torna o ator inválido.

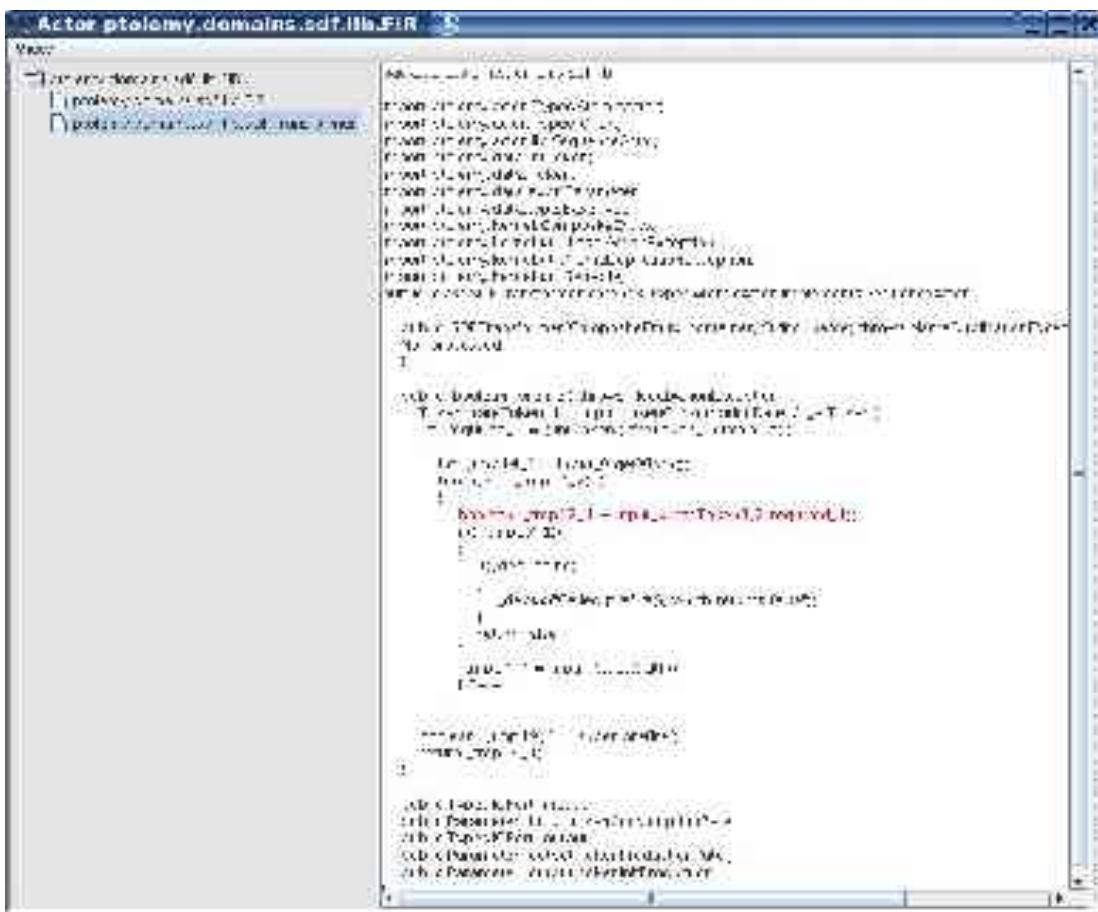


Figura 90: a janela de análise do código de um ator da ferramenta FASI.