Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Informática

Programa de Pós-Graduação em Ciência da Computação

Desenvolvimento de Aplicações Paralelas a partir de Modelos em Gramática de Grafos Baseada em Objetos

Fábio Pasini

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Fernando Luís Dotti

Porto Alegre

2007

Pontifícia Universidade Católica do Rio Grande do Sul
BIBLIOTECA CENTRAL IRMÃO JOSÉ OTÃO

Dados Internacionais de Catalogação na Publicação (CIP)

P282d Pasini, Fábio
Desenvolvimento de aplicações paralelas a partir de modelos em gramática de grafos baseada em objetos / Fábio Pasini. – Porto Alegre, 2007.
149 f.

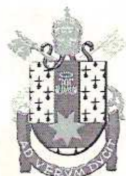
Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Fernando Luís Dotti.

1. Informática. 2. Programação Paralela. 3. Modelagem de Sistemas. 4. Teoria dos Grafos. 5. Sistemas Distribuídos. I. Título.

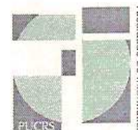
CDD 004.36

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca



PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.



TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Desenvolvimento de Aplicações Paralelas a partir de Modelos em Gramática de Grafos Baseada em Objetos***", apresentada por Fábio Pasini, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Confiabilidade de Sistemas, aprovada em 27/01/2006 pela Comissão Examinadora:

Prof. Dr. Fernando Luis Dotti – Orientador — PPGCC/PUCRS

Prof. Dr. César Augusto Fonticielha De Rose – PPGCC/PUCRS

Prof. Dr. Luiz Gustavo Leão Fernandes – PPGCC/PUCRS

Profa. Dra. Ana Paula Lüdtke Ferreira – UNISINOS

Homologada em 12/12/07, conforme Ata No. 025 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.pucrs.br/inf/pos e-mail: ppgcc@inf.pucrs.br

Aos meus pais, pelo grande empenho e incentivo
para que tudo isso acontecesse,
aos meus amigos, que sempre foram tanto apoio
quanto inspiração.

Agradecimentos

Agradeço ao professor Fernando Luís Dotti, pela paciência e disposição durante esses dois anos de trabalho, e ao colega Osmar Marchi, pela grande ajuda durante as etapas iniciais do mestrado. Agradeço também à HP e ao CAP pelo financiamento, sem o qual nada disso seria possível.

E por tudo que aconteceu nesses dois anos, tanto no ambiente do mestrado quanto fora, agradeço ao Odorico, por estar sempre disposto a discutir todo o tipo de idéias sobre modelagem e paralelismo. Certamente os resultados teriam sido bem menores se não fosse a sua ajuda. Agradeço também ao Leonel, pelo grande exemplo de dedicação e amizade; ao André, Guilherme, à Virgínia e à Joseane, pelo apoio e companheirismo em todas as horas; ao Ijuy pelo suporte em atividades aleatórias; ao Calheiros e ao Rafael, por responderem às minhas perguntas desesperadas, feitas sempre no último segundo. Agredeço também ao restante da turma do café: ao Diego, ao Czekster, à Mônica, ao Chanin, à Cristina e ao Patrick, pelos momentos legais passados juntos.

Todos esses amigos foram fundamentais para o andamento do trabalho. Ver os desafios sendo superados em cada passo, um a um, tanto por mim quanto por essas pessoas, muitos com os quais eu estudei durante toda a graduação, é uma das coisas mais gratificantes que eu levo de tudo isso.

Resumo

No desenvolvimento de aplicações paralelas, além da análise de aspectos ligados ao desempenho, torna-se também importante a análise das propriedades funcionais do sistema para garantir, por exemplo, que a estratégia de paralelização escolhida é adequada ao problema sendo abordado, ou que ela pode convergir para um resultado esperado, ou mesmo para identificar a possibilidade de um cenário de bloqueio na computação. A garantia de correção sobre o modelo de uma aplicação paralela, além de aumentar o grau de confiança nos resultados, pode também ser um fator de economia, já que possibilita a redução no tempo despendido no desenvolvimento e depuração da aplicação. Porém, uma vez identificados os problemas e correções no modelo analisado, ainda existe a necessidade de se mapear as mudanças necessárias à aplicação original. Nesse sentido, verificação formal e geração automática de código podem ser utilizadas como ferramentas complementares durante o desenvolvimento, possibilitando tanto a análise do comportamento do sistema quanto a rápida geração do código correspondente ao modelo proposto. Este trabalho apresenta o uso de Gramática de Grafos Baseada em Objetos (GGBO) para a construção de aplicações paralelas, a partir da definição de um método de tradução de modelos GGBO para código C, utilizando MPI como plataforma de comunicação.

Abstract

During parallel applications development, besides analysis regarding performance aspects, it is also important to analyze the system's functional properties to assure, for example, that the parallel strategy chosen is adequate for the problem being approached, or that it may converge to an expected result, or even to identify the possibility of a deadlock scenario. The correction guarantee over a parallel application model, besides improving the results reliability, also can be an economic factor, since it allows to reduce the time consumed for the application development and debugging. However, once identified the problems and corrections into the model analyzed, there is still the need to map the changes needed to the original application. In this sense, model-checking and automatic code generation can be used as complementary tools during development, allowing the system behavior analysis and a fast generation of the model's corresponding code. This work presents the use of Object-Based Graph Grammars (OBGG) for parallel applications development, through the definition of a method to translate OBGG models to C code, using MPI as communication platform.

Sumário

1 Introdução 23

2 Referencial 27

2.1 Processamento Paralelo . . . 27

2.2 Arquiteturas Paralelas . . . 29

2.3 Aplicações Paralelas . . . 30

2.3.1 Particionando o problema visando aumento de desempenho . . . 31

2.3.2 Detecção de Terminação em Sistemas Distribuídos . . . 33

2.4 Programação Paralela . . . 35

2.4.1 Modelagem de Aplicações Paralelas . . . 36

2.4.2 Linguagens de Programação Paralela . . . 39

2.4.3 Bibliotecas de Comunicação . . . 43

2.4.4 Linguagens Gráficas – Ferramentas para Geração de Código . . . 45

2.5 Verificação de Modelos . . . 49

2.5.1 Lógica Temporal . . . 51

2.5.2 Lógica Temporal Linear . . . 51

2.5.3 Padrões para Sentenças LTL . . . 56

2.5.4 Explosão do Espaço de Estados . . . 59

2.6 A Ferramenta SPIN para Verificação de Modelos . . . 59

3 Gramáticas de Grafos Baseadas em Objetos 65

3.1 A Linguagem GGBO . . . 65

3.1.1 Comparando GGBO com outras linguagens paralelas . . . 73

3.2 Análise de Sistemas Modelados em GGBO . . . 74

3.2.1 Simulação . . . 74

3.2.2 Verificação de modelos descritos em GGBO 75
3.3 Verificando o modelo Pi 76

4 Tradução 81

4.1 Tradução GGBO – MPI 81
4.1.1 Processo GGBO – Algoritmo Básico 81
4.1.2 Mapeamento Mensagens GGBO – Mensagens MPI 87
4.1.3 Mapeamento Regras GGBO – Código C 89
4.1.4 Inicialização do Sistema 91
4.1.5 Terminação 92
4.2 Experimentação e Medidas de Desempenho 94
4.2.1 Modelo Troca de Mensagens – ReSend, Rajada 94
4.2.2 O Modelo Pi 97

5 Semântica do Modelo Traduzido 101

5.1 O Modelo GGBO – Propriedades 101
5.2 Convertendo o *Template* utilizado 104
5.3 Verificando as propriedades definidas 109
5.3.1 Propriedade I – *Todas as mensagens com match são consumidas* 109
5.3.2 Propriedade II – *Qualquer mensagem com match pode ser consumida* . . . 110
5.3.3 Propriedade III – *Mensagens podem ser consumidas em qualquer ordem* . 112
5.3.4 Propriedade IV – *Regras podem ser aplicadas em qualquer ordem* 114

6 Estudos de Casos 117

6.1 Estudo de Caso – Perfil de Temperaturas em uma Peça Metálica 117
6.1.1 O Modelo *HeatCell* 120
6.1.2 Outra Alternativa para o Particionamento do Problema 127
6.2 Verificando os modelos *HeatCell* e *HeatCollumn* 130

7 Conclusão 137

Referências Bibliográficas 141

Lista de Tabelas

1 Sintaxe LTL utilizada pela ferramenta SPIN. 63

2 Características das linguagens GGBO e Java. 74

3 Características das linguagens GGBO e PROMELA. 76

4 Características da linguagem GGBO e padrão MPI. 82

5 Execução dos protótipos – Dois nodos alocados, dez fases. 131

Lista de Figuras

1 Metodologia de uso de um Verificador. 50

2 Autômato e interpretação segundo CTL e LTL para o comportamento. 52

3 Exemplos de interpretação para sentenças LTL. 55

4 Hierarquia de padrões para sentenças LTL. 57

5 Padrões de escopo para sentenças LTL. 58

6 Modelo de objeto em GGBO. 66

7 Grafos–tipo para o modelo proposto. 68

8 Uma exemplo de grafo inicial. 69

9 Regras para o objeto *Master*. 70

10 Regras para o objeto *Slave*. 71

11 Contra–exemplo associado à terceira sentença verificada. 78

12 Pseudo–código para *Receiver*. 86

13 Pseudo–código para *Sender*. 86

14 Tradução Mensagem GGBO – Código Fonte. 88

15 Preparando mensagem – *Packing*/*Unpacking*. 88

16 Função de *matching*. 89

17 Pseudo–código para *Evaluate*. 90

18 Grafo–tipo e regras para o objeto Teste. 95

19 Testes para Mensagens – Grafos Iniciais. 96

20 Testes para mensagens – Tempos de Execução. 96

21 Pseudocódigo de Base. 97

22 Medidas de desempenho. 98

23 Comparação entre resultados. 98

24 Nova regra para o objeto *Slave*. 99

25 Comparando programa base e modelo modificado. 99

26 Modelo para Verificação – Grafos–Tipo. 102
27 Modelo para Verificação – Regras. 103
28 Modelo para Verificação – Grafo Inicial 103
29 Modelo PROMELA para *Receiver*. 105
30 Modelo PROMELA para *Sender*. 106
31 Modelo PROMELA para *Evaluate*. 107
32 Modelos PROMELA – Concatenação de Listas e Deleção de Lista. 108
33 Modelos PROMELA – *Matching* e Retirada de par Mensagem–Regra. 108
34 Modelos PROMELA – Escolha de par Mensagem–Regra e Consumo de Mensagem. 108
35 Grafo Tipo para Objeto *Cell*. 120
36 Regras para o Objeto *Cell*. 121
37 Regras *HStart* para Objeto *Cell* com três objetos adjacentes. 122
38 Regras *HStart* para Objeto *Cell* com dois objetos adjacentes. 123
39 Regras *HBCast* para Objeto *Cell* com três objetos adjacentes. 124
40 Regras *HBCast* para Objeto *Cell* com dois objetos adjacentes. 125
41 Modelo *HeatCell* – Grafo Inicial. 128
42 Grafo Tipo para Objeto *Collumn*. 128
43 Regras para Objeto *Collumn*. 129
44 Regras *HSStart* para Objeto *Collumn* com um objeto adjacente. 129
45 Regras *HSBcast* para Objeto *Collumn* com um objeto adjacente. 130
46 Modelo *HeatCollumn* – Grafo Inicial. 130
47 Modelo *HeatCollumn* – Grafo Inicial mais detalhado. 131

Lista de Siglas

MPI	*Message Passing Interface*	23
GGBO	Gramática de Grafos Baseada em Objetos	24
UMA	*Uniform Memory Access*	29
NUMA	*Non–Uniform Memory Access*	29
NORMA	*Non–Remote Memory Access*	29
COW	*Clusters of Workstations*	29
NOW	*Network of Workstations*	29
SCI	*Scalable Coherent Interface*	30
CAD	*Computed Aided Design*	38
UML	*Unified Modeling Language*	40
P_3L	*Pisa Parallel Programming Language*	40
SPMD	*Single Program Multiple Data*	41
PVM	*Parallel Virtual Machine*	45
RPC	*Remote Procedure Calls*	46
NFS	*Network File System*	46
HeNCE	*Heterogeneous Network Computing Environment*	46
CODE	*Computation–Oriented Display Environment*	47
DPnDP	*Design Patterns and Distributed Processes*	48
CO_2P_3S	*Correct Object–Oriented Pattern–based Parallel Programming System*	48

CTL	*Computation Tree Logic*	51
LTL	Lógica Temporal Linear	51
BNF	*Backus–Naur Form*	52
PROMELA	*PROcess MEta LAnguage*	59
SPIN	*Simple Promela INterpreter*	59
ACM	*Association for Computing Machinery*	60
TAD	Tipo Abstrato de Dado	71

1 Introdução

Clusters ou *agregados* são máquinas de alto desempenho que possuem uma arquitetura baseada na reunião de um conjunto de estações de trabalho independentes, interconectadas por uma rede de comunicação de baixa latência, formando uma plataforma de execução para aplicações paralelas. Cada estação possui uma memória local, a qual somente os processadores locais tem acesso. Nesse contexto, o MPI (*Message Passing Interface*) [73] é um padrão bastante difundido para interface de passagem de mensagens para aplicações que utilizam computadores com memória distribuída. Ele não oferece nenhum suporte para tolerância a falhas e assume a existência de comunicações confiáveis.

Sistemas concorrentes[1], mesmo os mais simples, podem apresentar comportamentos bastante difíceis de predizer. Aspectos como condições de corrida e *deadlocks* podem surgir e comprometer seu funcionamento, mesmo quando a solução algorítmica apresentada inspirar um alto grau de confiança. Assim, no desenvolvimento de aplicações paralelas deve-se atentar ao correto funcionamento do sistema além do seu desempenho. Tal fato motiva a realização de análises como simulações, verificação e provas formais de maneira a aumentar a qualidade das soluções implementadas. Além de aumentar o grau de confiança nos resultados, a garantia de corretude sobre uma aplicação paralela pode ser um importante fator de economia já que reduz o tempo computacional de um *cluster* (que é um recurso caro) despendido com a realização de testes durante o desenvolvimento.

Numerosas tentativas no desenvolvimento de ferramentas para programação paralela usam abstração para reduzir o nível de complexidade no desenvolvimento, de forma que os usuários possam desenvolver rapidamente programas corretos. A maioria dessas ferramentas visa princi-

[1] O uso da palavra "concorrência" neste documento refere-se a sistemas onde várias ações podem estar habilitadas em um dado instante. A ocorrência de tais ações de forma simultânea ou intercalada depende da forma de implementação do sistema.

palmente a geração do código relativo à comunicação dentro do sistema [64], requerendo apenas que o usuário escreva trechos de código seqüenciais responsáveis pela manipulação dos dados envolvidos.

Gramática de Grafo Baseada em Objetos(*GGBO*) [26] é um formalismo para modelagem de sistemas concorrentes onde os elementos comunicam-se através de troca de mensagens, oferecendo um alto nível de abstração para modelagem. Por ser baseada em um conjunto simples de elementos e ter uma semântica formal, permite a descrição sem ambigüidades de sistemas concorrentes. Por ser formal, permite o mapeamento para verificadores e simuladores, possibilitando análises sobre o comportamento dos sistemas modelados, bem como prova de propriedades dos mesmos. Tanto GGBO quanto MPI apresentam unidades concorrentes distinguíveis, comunicação por troca de mensagens assíncronas, supõem ambientes comportados com relação a falhas e não impõem limites de atraso aos processos ou à comunicação (modelo assíncrono de computação).

Esse trabalho apresenta a definição de um método automatizável de tradução de modelos GGBO em código fonte C e utilizando MPI como plataforma de comunicação, possibilitando a realização de um mapeamento direto entre o modelo definido e a implementação correspondente. Os resultados obtidos trazem vantagens como permitir a especificação de aplicações paralelas utilizando linguagem com paralelismo implícito, aumentando o nível de abstração do projetista; e ainda a possibilidade de verificação, simulação e geração de código tendo como base um formalismo único, apoiando-se em outro resultados já obtidos dentro dos projetos PLATUS[2], ForMOS[3] e CASCO[4]. Ainda, a plataforma de comunicação MPI apresenta outras vantagens que justificam a sua adoção como alvo para a tradução de modelos GGBO: (i) é de ampla utilização; (ii) apresenta-se como uma solução robusta para comunicação de grupo e ponto-a-ponto; (iii) é de fácil utilização; (iv) portabilidade e (v) suporta redes de comunicação de alto desempenho como *Myrinet* e *SCI*.

O restante deste documento organiza-se da seguinte forma: No Capítulo 2 são discutidos aspectos relativos à plataforma de comunicação e de execução a que se aplica o trabalho e aspectos sobre modelagem, programação e análise de sistemas concorrentes O Capítulo 3 apresenta o formalismo utilizado como base para a geração dos modelos. No Capítulo 4 é introduzida a estrutura de tradução proposta, apresentando os elementos adotados em função das restrições relativas à linguagem de modelagem, linguagem de programação e ambiente de execução en-

[2]Desenvolvimento Formal e Simulação de Sistemas Reativos (FAPERGS/CNPQ)
[3]Métodos Formais para Código Móvel em Sistemas Abertos (FAPERGS/CNPQ)
[4]Centro de Análise de Sistemas Concorrentes (HP-PUCRS)

volvidos. No Capítulo 5, Verificação de Modelos é utilizada para relacionar o comportamento apresentado pelo código gerado e a semântica associada à linguagem GGBO. Estudos de caso visando exemplificar a utilização da metodologia proposta são apresentados no Capítulo 6. Finalmente, são apresentadas considerações sobre o estudo desenvolvido, assim como discutidos trabalhos futuros dentro da mesma proposta.

2 Referencial

Nesse capítulo será introduzido o referencial teórico relativo aos temas envolvidos neste trabalho. Inicialmente serão abordados conceitos gerais sobre paralelismo: a motivação em seu uso, classificação de arquiteturas paralelas e as abordagens utilizadas no tratamento paralelo de tarefas. São também discutidos aspectos sobre a detecção de parada em sistemas distribuídos, um tema que ganhará maior importância em seções futuras.

Após isso, serão discutidos aspectos sobre modelagem de sistemas paralelos, apresentando uma argumentação favorável à utilização de modelos para sua elaboração e análise. São analisados alguns exemplos de linguagens para modelagem de aplicações paralelas, linguagens de programação propriamente ditas e bibliotecas de comunicação para linguagens seqüenciais. Neste Capítulo serão também abordadas ferramentas gráficas para a geração de código de aplicações paralelas,

Encerrando esta parte do volume, são discutidos elementos sobre verificação formal de modelos, sua aplicação em sistemas paralelos e limitações no seu uso. São apresentadas a linguagem de especificação de propriedades e a ferramenta de verificação de modelos a ser utilizada no decorrer deste trabalho.

2.1 Processamento Paralelo

Historicamente, existe uma demanda crescente por poder computacional. Apesar do contínuo aumento da velocidade de processamento e capacidade de armazenamento obtidos pela indústria de equipamentos computacionais (Lei de Moore), existe uma ampla gama de aplicações onde mesmo o mais rápido computador tradicional não pode operar em um tempo razoável. Tais tarefas representam uma demanda imediata, sendo inviável esperar pelo aumento da velocidade dos componentes a ponto de viabilizar a sua realização. Uma das maneiras de aumentar o poder

computacional é a utilização de múltiplos processadores operando juntos em uma mesma tarefa. O problema maior é dividido em partes, cada uma das quais é realizada por um processador operando em paralelo com os demais.

A plataforma computacional – um *computador paralelo* – pode ser um computador especialmente projetado, contendo múltiplos processadores interconectados por um barramento, ou mesmo vários computadores independentes conectados através de uma rede de baixa latência. A presença de múltiplos processadores em um sistema distribuído abre caminho para a possibilidade de diminuição no tempo de resposta de uma tarefa intensiva em processamento. Ainda, a existência de múltiplos dispositivos de armazenamento (memória, disco rígido) permite o tratamento de um volume de informação maior que o possível em um sistema uniprocessado, tanto por questões de tamanho quanto de vazão de dados.

Além do aumento de desempenho, podem ser apontados ainda outros fatores favoráveis ao uso de paralelização em sistemas computacionais:

- **Simplificação do projeto através da especialização:** O desenvolvimento de um sistema computacional pode se tornar uma tarefa bastante complicada, especialmente se uma gama muito ampla de funcionalidades for requerida. Assim, o projeto pode ser simplificado separando o sistema em módulos, com cada um implementando uma parte das funcionalidades requeridas e comunicando–se com os demais. Cada módulo pode tomar vantagem de características específicas do *hardware* onde é executado (periféricos, memória), sendo possível ainda um ajuste fino do código para garantir o uso ainda mais eficiente desses recursos.

- **Compartilhamento de Recursos:** Mesmo com a progressiva diminuição do custo dos equipamentos, tornando praticável equipar cada funcionário de uma empresa com um computador, o mesmo não é verdade para todos os periféricos como por exemplo impressoras, scanners e unidades de disco de alta capacidade, ou mesmo *softwares* com restrições específicas. Em uma escala menor cada computador pode utilizar um servidor dedicado para prover serviços como compiladores ou gerenciar o uso de um dispositivo.

- **Aumento da confiabilidade através da replicação:** Sistemas distribuídos têm o potencial de se tornarem mais confiáveis do que sistemas uniprocessados devido à sua propriedade de *falha parcial* (*partial failure*) [79]. Falha parcial significa que alguns elementos do sistema podem falhar, enquanto outros ainda continuam operando corretamente e podem

assumir as tarefas do elemento falho. Uma falha em uma aplicação uniprocessada impossibilita o funcionamento do sistema como um todo. Um sistema de alta confiabilidade tipicamente consiste em duas ou mais unidades de processamento que rodam uma aplicação e utilizam um mecanismo de voto para filtrar os resultados obtidos. Além de protocolos de eleição, o correto funcionamento de um sistema distribuído em presença de falhas requer suporte algorítmico adequado, como por exemplo detecção de falhas, recuperação, etc. Tais mecanismos fogem do escopo definido nesse trabalho.

2.2 Arquiteturas Paralelas

Computadores paralelos consistem basicamente em três partes principais: processadores, módulos de memória e meios de comunicação. A rede de conexão interliga os processadores entre si, assim como a memória em alguns casos.

Uma classificação utilizada pra caracterizar máquinas paralelas diz respeito ao custo de acesso à memória [41]. Uma máquina paralela é dita UMA (*Uniform Memory Access*) quando todos os elementos de processamento acessarem a memória ou módulos de memória a um mesmo custo. De outra maneira, quando por questões de topologia ou distribuição dos módulos de memória, existirem diferentes possibilidades de atraso na comunicação com esses módulos, a arquitetura é chamada NUMA (*Non–Uniform Memory Access*). Quando cada um dos elementos de processamento de uma máquida paralela possui uma memória local, à qual só ele tem acesso, a arquitetura é chamada NORMA (*Non–Remote Memory Access*).

Clusters são máquinas paralelas NORMA de alto desempenho que possuem uma arquitetura baseada na reunião de um conjunto de estações de trabalho independentes, interconectadas por uma rede de comunicação de baixa latência. Estas máquinas paralelas também são referenciadas como *Agregados* ou ainda como *Clusters of Workstations* (COWs). Quando interconectados por uma rede comum, essas plataformas podem também ser chamadas de *Network of Workstations* (NOWs).

Clusters constituem uma plataforma de execução de aplicações paralelas que têm ganhado bastante destaque pelo custo acessível de implantação e pela escalabilidade oferecida. Por ser baseado em estações de trabalho comuns funcionando como nodos de processamento na estrutura, uma máquina como essa pode ser construída a partir de componentes *of–the–shelf*, encontrados a preços acessíveis, podendo ainda ser posteriormente ser ampliada através da compra de mais

nodos. A rede de intercomunicação utilizada pode ser baseada em padrões bem conhecidos como *Ethernet*, ou outras alternativas desenvolvidas especificamente visando maior desempenho como *SCI* [42] e *Myrinet* [10].

Diversos fatores facilitam a geração de aplicações paralelas para a execução nessa plataforma. A arquitetura utilizada nos nodos facilita a adaptação dos compiladores já existentes para a geração de código executável. Existe também uma ampla gama de bibliotecas de programação adequadas a essa arquitetura, disponíveis para linguagens de programação convencionais, assim como um conjunto considerável de linguagens de programação específicas para ambientes paralelos. Finalmente, existem ferramentas para o gerenciamento do recursos (*e.g.* [28, 47, 56]), possibilitando um fácil acesso aos nodos, inicialização e controle de execução.

2.3 Aplicações Paralelas

Programas concorrentes consistem em uma coleção de processos e estruturas de acesso comum a esses processos usados para a comunicação [61]. Cada processo pode ser considerado como um subprograma seqüencial sendo executado em paralelo com os demais dentro de um mesmo programa maior [1]. Os objetos compartilhados permitem aos programas cooperar para a execução de uma tarefa em outros momentos além de seu início e término, trocando dados e informações de controle. Basicamente, são utilizados dois mecanismos de comunicação:

- **Memória Compartilhada:** Em arquiteturas com memória compartilhada, os processos se comunicam escrevendo dados em posições específicas de memória, as quais podem ser acessadas pelos demais processos participantes. Quanto utilizando tal estratégia de comunicação torna–se necessário identificar quando é possível ler ou escrever os dados de maneira que a informação permaneça consistente. Estruturas de controle padrão em sistemas operacionais como semáforos ou monitores são usados para tal propósito.

- **Troca de Mensagens:** Quando não existir à disposição uma memória compartilhada entre os processos, estes podem compartilhar dados através de troca de mensagens. O processo empacota a informação em uma mensagem com um cabeçalho indicando a qual processador

[1] Em ambientes paralelos deve–se distinguir o *paralelismo virtual* de um programa, que corresponde ao número de processos lógicos independentes contidos nesse programa (dependente, pois, da lógica de execução utilizada), do *paralelismo físico*, relativo ao número de processos que podem estar ativos em um determinado momento (que é menor ou igual ao número de processadores em uma máquina paralela).

e processo o dado deve ser enviado. No envio *não–bloqueante*, uma vez que a mensagem tenha sido passada para a rede, o processo gerador pode continuar sua computação. Quando o envio é do tipo *bloqueante*, o processo permanece na operação de envio até que os dados estejam efetivamente disponíveis no processo destino. O processo destinatário deve ser capaz de receber os dados, através da execução de uma operação de recepção. Se não existir mensagem disponível, o processo pode bloquear até que exista mensagem (*recepção bloqueante*), ou continuar o processamento para posteriormente testar a existência de uma mensagem (*recepção não–bloqueante*).

Esses mecanismos de comunicação não precisam corresponder diretamente ao que a arquitetura oferece. É possível simular troca de mensagens usando memória compartilhada, assim como é possível simular memória compartilhada usando troca de mensagens (uma técnica conhecida como *virtual shared memory*). Uma das mais fortes razões para a utilização do paradigma de troca de mensagens está na sua aplicabilidade direta a computadores conectados através de uma rede [82].

A rede de intercomunicação é o mecanismo que permite aos processadores comunicarem–se entre si e com os módulos de memória. A *topologia* dessa rede é o arranjo completo das ligações individuais entre os elementos de processamento, sendo naturalmente representada como um grafo. A *latência* de uma rede de comunicação é o maior tempo necessário para qualquer par de elementos de processamento se comunicarem.

O *desempenho* (ou *performance*) de um programa é usualmente expresso em termos de seu tempo de execução. Isso depende tanto da velocidade individual de cada processador quanto do arranjo da comunicação, além da capacidade dos meios de comunicação em tratar o fluxo de dados.

2.3.1 Particionando o problema visando aumento de desempenho

Na busca por melhorias no desempenho de aplicações paralelas, podem ser encontrados na literatura um conjunto de padrões comuns para o fracionamento e distribuição das partes do problema sendo paralelizado. Dentre essas estratégias, destacam–se as seguintes abordagens:

- **Mestre–Escravo:** Essa abordagem pode ser utilizada quando uma tarefa maior puder ser dividida em partes menores sem dependência entre si. Dessa forma, cada subtarefa pode ser tratada independentemente por um processo escravo distinto, sendo os resultados enviados

de volta para o mestre. Na abordagem *Bag–of–tasks*, o mestre possui um conjunto grande de subtarefas, que são enviadas uma a uma para cada escravo. Mediante o particionamento apropriado da tarefa – o que muitas vezes não é uma decisão trivial – pode–se conseguir um bom balanceamento de carga entre os nodos escravos.

- ***Pipeline*:** Essa abordagem é bastante utilizada em *hardware* para aumentar a vazão total do sistema. Se existe um conjunto identificável de passos a ser realizado em uma seqüência específica, um algoritmo semelhante a uma linha de montagem pode ser utilizado, destinando–se um elemento de processamento para cada atividade a ser desempenhada. Cada passo é realizado por um processo diferente, sendo que a saída de um processo é tomada como entrada para o próximo, e assim sucessivamente.

  Após um determinado número de passos, o *pipeline* fica "cheio", e a vazão aumenta significativamente, uma vez que cada estágio da linha de montagem está efetivamente atacando uma tarefa em paralelo. Se o tamanho da tarefa a ser realizada não for significante, a abordagem em *pipeline* não é eficiente devido ao *overhead* necessário para encher o *pipeline.*

- **Dividir e Conquistar:** Dentro da abordagem "dividir e conquistar", o processamento é feito a partir de sucessivas divisões da tarefa a ser executada. O nodo inicial (raiz) avalia se o trabalho a ser desempenhado pode ser realizado localmente. Em caso negativo, ele delega partes da tarefa para dois ou mais processos escravos. Cada processo escravo decide se pode tratar a tarefa ou se é melhor realizar mais uma divisão do trabalho e delegar novamente as partes a outros processos. Dessa forma, a divisão da tarefa leva os processos a assumirem uma topologia em árvore para o tratamento do problema.

  Quando o processamento nos nodos folha é terminado, os resultados são repassados para o processo pai, que pode realizar algum processamento adicional sobre esses resultados antes de novamente repassar os valores obtidos. Esse processo continua sucessivamente, até que o nodo raiz processe o resultado final.

  Um problema potencial dessa abordagem recursiva está em certificar–se que existe uma condição de terminação para a condição de particionamento, evitando assim que processos continuem a ser disparados indefinidamente.

- **Fases Paralelas:** Nessa tipo de abordagem, todos os processos sincronizam suas computações regularmente durante a execução, garantindo que todos os participantes tenham

terminado uma fase do processamento antes de iniciar a próxima. Todos os processos iniciam uma etapa em um mesmo momento, executando uma operação (*e.g.*, um *barrier* [82]) em uníssono.

Dependendo do tipo de abordagem utilizada no tratamento do problema, pode–se compor um cenário no qual o grande desacoplamento entre os participantes torne difícil identificar quando a computação distribuída como um todo tenha atingido um determinado estado em que o processamento possa ser declarado terminado. Na próxima seção esse problema é formalizado, sendo discutidos brevemente algumas classes de soluções possíveis.

2.3.2 Detecção de Terminação em Sistemas Distribuídos

Desde que foi proposto em 1980 por Francez [36] e por Dijkstra e Scholten [22], o problema da terminação distribuída constitui um tema clássico em sistemas distribuídos, devido a sua importância prática e teórica. Considerando a execução de uma aplicação distribuída (chamada computação primária), o objetivo é construir um programa de controle que detecte a terminação da computação primária. Tal detecção não é trivial, uma vez que em um sistema distribuído onde os processos se comuniquem unicamente através de troca de mensagens, em geral nenhum processo tem uma visão atualizada do estado global do sistema [80].

Para a especificação de um algoritmo de controle, é assumido o seguinte modelo [36, 22, 79] para a computação primária genérica, ao qual qualquer aplicação paralela pode ser enquadrada:

i. Cada processo da computação primária pode estar em apenas dois estados: *ativo* ou *passivo*;

ii. Um processo ativo se torna passivo somente em função de seus eventos internos;

iii. Apenas processos ativos geram mensagens;

iv. Um processo passivo sempre se torna ativo com o recebimento de uma mensagem;

Uma computação distribuída é considerada globalmente terminada quanto

i. cada processo está localmente terminado;

ii. não existem mais mensagens em trânsito.

Para que a identificação seja possível, a computação primária deve ser modificada de forma que o algoritmo de controle possa obter informações sobre o estado atual de cada elemento. Isso é feito através da geração de *mensagens de controle*, destinadas ao algoritmo de controle,

associadas a determinadas transições do algoritmo primário. Essas mensagens de controle não alteram o estado da computação primária. A computação é considerada terminada quando o algoritmo combinado (primário mais algoritmo de controle) alcançar um estado que respeite ambas as condições impostas. Note que o algoritmo de controle pode tanto ser executado por apenas um dos processos participantes do sistema, quanto ser uma tarefa distribuída por todo o conjunto.

Uma solução possível é o uso de *pooling*, onde periodicamente um coordenador testa todos os processos, inspecionando seu estado (ativo ou bloqueado). Assim, uma vez que tenha confirmado o estado bloqueado para todos os processos, ele pode concluir que a computação efetivamente terminou. Essa solução não garante uma identificação rápida do estado do sistema – um ciclo inteiro, no pior caso – além de possibilitar falsos positivos – quando ainda existirem mensagens em trânsito pelos mecanismos de comunicação.

Existem muitas classes de soluções para o problema da detecção. As mais importantes são algoritmos baseados em *Probe*, algoritmos baseados em *Acknowledgements* e algoritmos baseados em *Contagem de Mensagens*.

- **Algoritmos baseados em *Probe*:** Um *Probe* é uma tarefa distribuída que visita todos os processos na rede. Ela pode ser implementada como um *token* circulando um anel, por exemplo. Para a detecção utilizando *probes*, tenta–se identificar um estado em que todos os processos visitados estejam bloqueados, e nenhuma mensagem esteja em trânsito destinada a um processo visitado. Uma violação ocorre quando um processo não visitado envia uma mensagem a um processo visitado. Quando isso ocorre, o *probe* corrente é marcado como inválido, e depois de sua finalização outro é iniciado. A terminação é detectada quando o *probe* passa por todos os nodos sem encontrar violações. Um exemplo desse tipo de algoritmo é dado em [21], e um tratamento geral em [78].

- **Algoritmos baseados em *Acknowledgements*:** Nessa classe de algoritmos, toda a mensagem primária é confirmada (*acknowledged*), mas somente após toda a atividade computacional resultante dela ter cessado. Isso é, se um processo ativo recebe uma mensagem, ele a confirma imediatamente. Se um processo bloqueado recebe uma mensagem, ele defere a confirmação até se tornar bloqueado novamente e ter recebido a confirmação de todas as mensagens que tenha originado após o desbloqueio, durante seu período de atividade. Quando todos os processos estejam bloqueados e tenham recebido confirmações de todas as

mensagens primárias, a terminação é identificada. Generalizações sobre essa classe podem ser encontradas em [66].

- **Algoritmos baseados em *Contagem de mensagens*:** Mantendo–se um controle do número de mensagens primárias geradas no sistema, é possível determinar se existe informação em trânsito pelos mecanismos de comunicação, evitando assim a ocorrência de falsos positivos. Na Seção 4.1.5 é apresentada em maiores detalhes uma solução utilizando essa estratégia.

A escolha de um algoritmo de detecção em um sistema que busque desempenho deve considerar a extensão em que o algoritmo de detecção interfere (e atrasa) o fluxo principal de processamento. O número absoluto de mensagens de controle não é um caracterizador único desse fator, uma vez que muitas dessas mensagens de controle são geradas/processadas por participantes que não estão engajados na computação principal. O volume de mensagens geradas gera impacto, porém, no tráfego da rede sendo utilizada. Esse efeito é amortizado por exemplo, quando a rede sendo utilizada apresentar baixa latência e alta confiabilidade.

Em [69] e [52] é argumentado que o principal fator a ser considerado na decisão por um algoritmo de detecção é o tempo necessário para identificar a parada. Existem muitos fatores que tornam o atraso de detecção o fator mais importante a ser considerado. Primeiro, esse atraso significa desperdício de recursos computacionais. Além disso, em muitas aplicações (*e.g.* algoritmos de busca) o resultado da computação principal não pode ser utilizado até que a terminação de todos os nodos tenha sido confirmada. Ainda, muitas estratégias de processamento utilizadas por aplicações consistem em fases distintas, onde uma nova fase somente pode começar depois que a fase anterior tenha terminado, o que requer a utilização de múltiplos algoritmos de detecção, acumulando o tempo necessário para cada uma delas.

2.4 Programação Paralela

A programação paralela difere em vários aspectos da programação de um sistema seqüencial, modelo com o qual a maioria dos desenvolvedores tem seu primeiro contato. A intercalação de estados possíveis entre os processos dentro de um sistema paralelo é difícil de predizer, dificultando a identificação da causa do problema com técnicas tradicionais de depuração de programas usados para algoritmos seqüenciais. Ainda, considerando–se as diferenças entre as várias plataformas de

execução e comunicação possíveis, têm–se um quadro bastante propenso a erros de programação durante o desenvolvimento.

São listadas como as três maiores fontes de erros em programas paralelos [43]:

- **(i) Erros semânticos:** esses erros são causados por equívocos sobre o modelo de programação paralela sendo utilizado e sua aplicação para o problema em questão. Por exemplo, erros podem surgir devido ao não entendimento apropriado dos mecanismos de compartilhamento de memória entre os processos e da semântica da linguagem utilizada.

- **(ii) Erros de implementação:** esses erros ocorrem devido à complexidade extra inerente à utilização de programação paralela, como o lançamento de processos, comunicação e sincronização. Erros incluem o empacotamento/desempacotamento (*packing/unpacking*) de uma mensagem ou a criação de um cenário de *deadlock*.

- **(iii) Erros de *performance*:** são erros causados pela falta de intuição ou experiência acerca dos custos de paralelismo e concorrência. Por exemplo, essa classe de erros inclui a execução de segmentos de programas em grão–fino utilizando mecanismos de sincronização que restringem a concorrência.

Os dois primeiros erros normalmente resultam em um programa que executa de maneira incorreta, ou mesmo impossibilitam a geração de um executável. O terceiro tipo de erro geralmente leva a programas que rodam corretamente, porém com um ganho de desempenho baixo quando comparados com a solução seqüencial correspondente.

A utilização de ferramentas para a geração automática de código paralelo possibilita evitar os dois primeiros problemas naturalmente, uma vez que libera o usuário da necessidade de trabalhar aspectos específicos à linguagem e bibliotecas de comunicação utilizadas. O terceiro erro, por sua vez, pode ser contornado através da especialização do gerador utilizado, de modo a guiar o usuário em direção a um modelo mais adequado durante o desenvolvimento. Tais ferramentas tomam como entrada uma descrição – um modelo – do sistema a ser gerado, expressos seguindo algum padrão, formalismo ou metodologia de especificação.

2.4.1 Modelagem de Aplicações Paralelas

Um modelo de computação paralela é uma interface separando propriedades de alto nível de propriedades de baixo nível [72]. Ele provê uma descrição do sistema considerado, focando

os elementos mais diretamente envolvidos. Retome como exemplo a descrição apresentada na Seção 2.3.2, onde a definição do problema e das soluções correspondentes é feita sobre um modelo bastante abstrato e simples, apresentando somente as características necessárias. Tal abstração permite a definição de uma solução genérica e facilita seu mapeamento para aplicações reais.

Mais concretamente, um modelo para implementação é uma máquina abstrata que provê certas operações para o nível superior e requer uma implementação para cada uma dessas operações provenientes de um nível inferior. É utilizado para separar conceitos de projeto de *software* de conceitos de execução paralela e provê tanto abstração quanto estabilidade [72].

Abstração é oferecida devido às operações que o modelo disponibiliza são de um nível mais alto do que aquelas oferecidas pela nível/arquitetura subjacente, simplificando a estrutura do *software* e diminuindo a dificuldade em sua construção. Estabilidade é obtida porque o desenvolvedor pode assumir uma interface padrão, válida mesmo quando mudanças ocorrerem nos níveis inferiores da arquitetura paralela[2]. Ainda, o modelo funciona como um ponto de partida para os esforços de implementação (transformação do sistema, tradução, compilação) direcionados a cada máquina paralela.

Modelos existem nos mais diferentes níveis de abstração. Por exemplo, cada linguagem de programação pode ser considerada um modelo, dado que todas provêm uma visão simplificada do *hardware* subjacente. Dentro das várias possibilidades, um bom modelo de computação paralela deve oferecer as seguintes propriedades [72]: (i) facilidade de utilização; (ii) metodologia de desenvolvimento; (iii) independência de arquitetura; (iv) fácil compreensão; (v) desempenho e (vi) custo previsível. A seguir, são discutidas cada uma dessas propriedades.

i) Facilidade de utilização: Um modelo deve ocultar ao programador detalhes que ele não seja capaz de controlar durante o desenvolvimento do sistema. A exata estrutura do programa a ser executado pode ser inserida pelo mecanismo de tradução (ou compilação) em vez do programador. Por exemplo, uma linguagem de modelagem pode abstrair ao programador elementos como decomposição do programa em *threads*, mapeamento das *threads* entre os processadores, comunicação e sincronização.

Modelos devem ser o mais abstratos e simples possível, dentro da finalidade à qual se destinam; devem ser o mais próximo possível da forma natural de se descrever o programa e das necessidades da linguagem de programação. Em alguns casos, isso pode significar que o par-

[2]Desde que os elementos afetados pela mudança sejam adequados à nova situação, o modelo pode permanecer inalterado.

alelismo não precisa necessariamente estar explícito ao programador, como por exemplo *Haskell* [40], *Unity* [19] e GGBO[26].

ii) Metodologia de desenvolvimento: O item anterior envolve um enorme espaço entre as informações fornecidas pelo programador sobre a estrutura semântica do programa e a estrutura detalhada requerida para as sua execução. Para ligar ambas faz–se necessária uma fundamentação semântica concisa sobre as quais técnicas de transformação possam ser elaboradas.

iii) Independência de arquitetura: O modelo (e conseqüentemente o programa gerado) deve poder ser migrado para outro ambiente paralelo sem necessitar ser redesenvolvido ou modificado de qualquer maneira não trivial. Essa característica é importante para disseminar a utilização da linguagem de modelagem, assim como possibilitar que esse uso seja possível para novos equipamentos, uma vez adaptado o mecanismo de tradução.

iv) Fácil compreensão: Um modelo deve ser fácil de entender, facilitando assim a adoção da linguagem de modelagem e a sua efetiva utilização. A existência de ferramentas adicionais como simuladores associados à linguagem de modelagem pode facilitar ao usuário o entendimento da semântica proposta, auxiliando no entendimento dos mecanismos providos pela linguagem e sua correta utilização no desenvolvimento.

v) Desempenho: Um modelo deve oferecer um bom *desempenho*. Note que isso não significa que a implementação deve extrair cada possibilidade em *performance* de uma determinada arquitetura. Para a maioria dos problemas, um nível tão apurado de *performance* só pode ser obtido através de gastos consideráveis em depuração e especialização do código.

vi) Custo previsível: Implementações devem respeitar a ordem de complexidade aparente de um modelo e procurar manter o custo de suas operações baixo.

Essas características desejáveis para linguagens de modelagem podem ser bastante dispendiosas e estarem em conflito entre si. Por exemplo, modelos abstratos facilitam a elaboração de programas mas tornam difícil a geração do código correspondente, enquanto modelos de baixo nível, apesar de dificultar a modelagem, facilitam a tradução e implementação do sistema.

A seguir, são discutidos vários exemplos de linguagens para modelagem de aplicações paralelas. Algumas apresentam–se como linguagens de programação, enquanto outras são utilizadas como bibliotecas, ou mesmo dentro de ambientes específicos para o projeto semelhantes à ferramentas CAD (*Computed Aided Design*). Cada uma dessas linguagens quantifica de forma diferente as características apresentadas, definindo assim uma melhor ou pior adequação de seu

uso durante o projeto o sistema.

2.4.2 Linguagens de Programação Paralela

2.4.2.1 Haskell – *Higher Order Functional Programming*

A técnica utilizada por linguagens funcionais ditas *higher order* para avaliar as funções é chamada *redução de grafos* (*graph reduction*) [6]. Segundo essa representação, as funções necessárias para a realização do processamento são expressas como árvores, utilizando–se sub–árvores comuns para armazenar funções compartilhadas. Regras selecionam as subestruturas do grafo, que representam seqüências de operações passíveis de execução em um dado momento. O resultado correspondente toma o lugar da subestrutura original, reduzindo progressivamente o grafo para formas mais simples. Quando nenhuma outra regra computacional puder ser aplicada, o grafo final é tomado como resultado da computação.

O método *graph–reduction* pode ser paralelizado aplicando–se regras em seções distintas do grafo, e utilizando–se múltiplos processadores para buscar independentemente por partes redutíveis dentro da estrutura. Por exemplo, numa expressão como $if\ f(x)\ then\ a(x)\ else\ b(x)$, onde $f(x), a(x)$ e $b(x)$ são representados como sub–árvores, *threads* podem avaliar independentemente cada uma delas. Apesar de elegante, essa estratégia de paralelização pode não levar a um aproveitamento eficiente dos recursos da máquina. No exemplo, apenas um dentre $a(x)$ e $b(x)$ necessitaria efetivamente ser calculado, porém a escolha somente é possível se existir uma serialização na avaliação (realizá–las após o cálculo de $f(x)$). Esse tipo de dependência é um limitador para o paralelismo possível na linguagem.

Haskell [40] é um exemplo de linguagem que utiliza *graph–reduction*, incluindo também muitas características típicas de linguagens funcionais como *lazy evaluation* [44], *pattern matching* e compreensão de listas.

2.4.2.2 P_3L – *Skeletons*

Padrões para programação paralela existem há cerca de duas décadas em formas como *skeletons*[34, 60], *templates*[68, 70] e *design patterns*, por exemplo.

Skeletons e *Templates* correspondem a algum algoritmo padrão ou fragmento de algoritmo de uso comum dentro da qual o usuário introduz o código dependente da aplicação. A implementação de cada *skeleton* é realizada de maneira que eles possam ser compostos seqüencialmente ou paralelamente. É necessária a disponibilização de uma implementação de cada bloco para cada

arquitetura alvo desejada, sendo que cada bloco pode ocultar uma quantidade arbitrária de computação paralela. Durante o desenvolvimento da aplicação, o programador seleciona os *skeletons* que deseja utilizar e compõe o programa. O compilador ou gerador de código determina como o paralelismo intra e entre blocos será explorado para cada arquitetura alvo possível.

Design patterns são estruturas descritivas para trechos de código de uso comum. Eles incluem um diagrama esquemático (muitas vezes em UML) e uma descrição que consiste em sete partes [37]: objetivo, motivação, aplicabilidade, estrutura, participantes, colaborações, conseqüências, implementação, código fonte, usos conhecidos e padrões relacionados. Na maioria dos casos também existem pontos onde devem ser inseridos trechos adicionais de código. *Design patterns* implementam padrões comuns para estruturas de interações entre processos encontradas em sistemas paralelos e seqüenciais, porém deixando não–especificados os procedimentos dependentes da aplicação [70].

Design patterns, *Skeletons* e *Templates* abstraem estruturas de ocorrência comum na comunicação em aplicações paralelas, permitindo que os usuários desenvolvam aplicações de uma maneira mais rápida e fácil. Esta aproximação fortalece a corretude da aplicação paralela por fornecer estruturas de código de comunicação bem testadas que de outra maneira teriam que ser escritos manualmente pelo usuário.

Pisa Parallel Programming Language (P_3L) é um exemplo de linguagem que utiliza um conjunto de *skeletons* que capturam paradigmas paralelos comumente utilizados como *pipelines*, *bag–of–tasks* e *worker farms*. Por exemplo, um *worker farm* em P_3L é modelado pelo seguinte construtor:

```
farm P in (int data) out (int result)
  W in (data) out (result)
  result = func(data)
  end
end farm
```

Quando o *skeleton* é executado, um determinado número de *workers* W é acionado em paralelo como os dois processos P (um gerador de dados, outro coletor). Cada *worker* executa a função $func()$ em sua partição de dados.

Skeletons são simples e abstratos. Porém a expressividade de uma linguagem de programação baseada em *skeletons* fica limitada aos blocos oferecidos [72]. Ainda, a semântica heterogênea permitida pelas diferentes implementações de um mesmo padrão para arquiteturas diferentes

pode tornar difícil a formalização do uso de cada bloco.

2.4.2.3 *Cellular Processing Languages*

Cellular processing languages expressam sistemas paralelos baseando–se em um modelo de execução de autômatos celulares. Um autômato celular consiste em um reticulado n–dimensional de células, onde cada célula é conectada a um conjunto limitado de células adjacentes. O estado de um autômato celular como um todo é definido pelos valores das variáveis em cada célula.

A atualização do estado das células é feita em tempo discreto, em passos atômicos e simultaneamente em vários pontos do reticulado. As células atualizam seus valores utilizando funções de transição que tomam como entrada o estado corrente da célula local e um subconjunto dos valores das células vizinhas.

Linguagens Celulares de Processamento como Cellang [32], CARPET [74], CDL [38], e CEPROL [65] permitem descrever algoritmos celulares através da definição do estado das células como variáveis ou estruturas de variáveis, e as funções de transição contêm as regras de evolução de um autômato. São disponibilizados construtores para a definição de padrões de vizinhança de um célula que são considerados na aplicação das regras.

Essas linguagens implementam autômatos celulares como programas SIMD ou SPMD, dependendo da arquitetura alvo. Segundo o paradigma SPMD (Single Program, Multiple Data) algoritmos celulares são implementados como uma coleção de processos mapeados em diferentes elementos de processamento. Cada processo executa o mesmo programa (no caso, definido pelas funções de transição). Assim, todos os processos executam, em paralelo, as mesmas regras locais, alterando a configuração do reticulado localmente. Isso permite implementações escaláveis tanto para plataformas MIMD quanto SIMD [14].

2.4.2.4 Linda – *Coordination Languages*

Linguagens de Coordenação (*Coordination Languages*) buscam simplificar o projeto de sistemas paralelos separando aspectos relativos ao processamento da entrada de aspectos inerentes à comunicação dentro do sistema, provendo uma linguagem separada para especificar essa comunicação. Tal separação torna a computação e a comunicação ortogonais entre si, de modo que um esquema de comunicação em particular possa ser aplicado a muitas aplicações seqüenciais [72].

Um exemplo dessa classe de linguagens é Linda [2, 15], onde a comunicação ponto–a–ponto é

substituída por um *pool* compartilhado (*shared pool*) no qual dados são colocados e retirados associativamente. Esse *pool* compartilhado é chamado *espaço de tuplas*. O modelo de comunicação oferecido pela linguagem Linda contêm três operações: (i) *in*, que remove a tupla do espaço de tuplas, baseada em sua aridade e nos valores de alguns de seus campos, preenchendo os demais campos com os valores a partir da tupla obtida; (ii) *read (rd)*, que apenas copia a tupla do espaço de tuplas, sem removê–la do *pool*; e (iii) *out*, que inclui uma tupla no espaço de tuplas. Linda oferece uma operação ($eval(t)$) para a criação de novos processos para avaliar o espaço de tuplas.

Por exemplo, uma operação $rd(\text{``}Planeta'', \ ?X, \ \text{``}Brasil'')$ procura dentro do espaço de tuplas por uma tripla contendo "Planeta", uma variável do mesmo tipo que X, e ainda "Brasil" como terceiro elemento, copiando para X o valor presente na tupla encontrada.

O modelo Linda requer que o programador gerencie as *threads* de um programa, mas reduz os custos impostos no controle da comunicação. Outra característica importante é a existência de uma metodologia de desenvolvimento de *software* associada à linguagem, oferecida por ambientes de programação como, por exemplo, o *Linda Program Builder* [1]. Tais ambientes tornam possível projetar, codificar, monitorar e executar programas utilizando essa linguagem.

2.4.2.5 Orca – *Broadcast*

Orca [8] é uma linguagem baseada em objetos que utiliza objetos de dados compartilhados para a comunicação entre processos. Orca baseia–se em um conjunto hierarquicamente estruturado de abstrações. No nível mais baixo, *broadcast* confiável é a primitiva básica de comunicação. Dessa forma toda escrita na estrutura compartilhada reflete–se nas réplicas locais mantidas pelos processos, o que oferece a abstração de um elemento único compartilhado.

O paralelismo em Orca é obtido a partir da criação explícita de processos. Na chamada, é possível especificar em qual processador/nodo executar o processo filho. Ainda, os parâmetros definem qual o objeto compartilhado a ser usado para comunicação entre os processos criados.

2.4.2.6 Ada – *Rendezvouz*

No modelo de comunicação baseado em *rendezvouz*, uma interação entre dois processos A e B ocorre quando A chama uma *entrada (entry)* de B e B executa um *accept* para aquela entrada [72]. Uma chamada *entry* é similar a uma chamada de procedimento. Uma chamada *accept*, por sua vez, envolve uma lista de ações a serem executadas quando a entrada é executada. O mecanismo como um todo é bastante semelhante ao de um *send*/*receive* síncrono entre dois

processos.

A linguagem mais conhecida baseada em *rendezvouz* é Ada [55]. O paralelismo na linguagem Ada é baseado em processos, chamados nesse contexto de *tasks*. Uma *task* pode ser criada explicitamente ou ser estaticamente declarada. *Tasks* são compostas em duas partes: a primeira (chamada *specification*) é responsável pela especificação, definindo quais as ações realizadas pela *Tasks*. A segunda parte, chamada *body*, define como as ações a são executadas. Declarações do tipo *entry* são permitidas apenas na parte de especificação de uma *task*. As estruturas *accept* referentes às entradas declaradas aparecem no corpo de uma *task*.

Ada comporta de maneira nativa um mecanismo de manipulação de exceções para tratar falhas de *software*. Outra característica importante oferecida pela linguagem Ada para a programação paralela é o uso da estrutura *select* para expressar não–determinismo de forma similar à estrutura encontrada em PROMELA, conforme será apresentado na Seção 2.6.

2.4.2.7 Java – *Threads*

Java [50] oferece multiprogramação e multiprocessamento baseados em *threads*, e permite a comunicação utilizando memória compartilhada através de variáveis de condição. Essas variáveis são acessadas através de métodos *synchronized*. Uma seção crítica protegendo o código do método é gerada automaticamente.

Existem ainda muitos outras bibliotecas que estendem linguagens convencionais possibilitando a comunicação entre processos, tanto através de memória compartilhada quanto de troca de mensagens. Esse assunto será explorado na Seção a seguir.

2.4.3 Bibliotecas de Comunicação

As operações necessárias na implementação de sistemas baseados em troca de mensagens são muito semelhantes para arquiteturas de memória distribuída, o que facilita a elaboração de padrões de interfaces de comunicação nesses sistemas. A implementação de tais padrões na forma de bibliotecas de comunicação, por sua vez, permite a sua utilização dentro de linguagens originalmente seqüenciais como Fortran e C. Tal possibilidade tem grande aceitação por parte dos desenvolvedores, uma vez que a utilização dessas linguagens lhes é familiar, representando um desafio muito menor do que o oferecido por uma linguagem nova ou mesmo paradigma completamente diferente. Nesse contexto, padrões como o MPI e PVM destacam–se pela grande

aceitação e larga utilização.

2.4.3.1 MPI

O MPI (*Message Passing Interface*) é um padrão de interface de passagem de mensagens para aplicações que utilizam computadores com memória distribuída. Ele não oferece nenhum suporte para tolerância a falhas e assume a existência de comunicações confiáveis.

O comitê MPI, criado em 1992, define este padrão para os ambientes de passagem de mensagens, reunindo membros de aproximadamente 40 instituições e inclui quase todos os fabricantes de máquinas paralelas, universidades e laboratórios governamentais pertencentes à comunidade envolvida na computação paralela mundial [54] [81]. Os principais objetivos são a padronização e eficiência para fabricantes de *hardware* e plataformas portáteis. Sua semântica foi escrita para ser independente da linguagem utilizada e visando definir uma interface de troca de mensagens que fosse possível de ser implementada eficientemente.

MPI define um conjunto de rotinas para facilitar a comunicação (troca de dados e sincronização) entre processos paralelos. A biblioteca MPI é portável para qualquer arquitetura, tendo aproximadamente 125 funções para programação e ferramentas para analisar o desempenho. Atualmente, a biblioteca MPI permite que os programas possam ser escritos nas linguagens Fortran, C e C++. O MPI não é um ambiente completo para programação concorrente, visto que ele não implementa I/O paralelos, depuração de programas concorrentes, canais virtuais para comunicação e outras características próprias de tais ambientes [81].

Um conjunto de rotinas responsáveis pela comunicação ponto–a–ponto entre os pares de processos forma o núcleo do MPI. Todo paralelismo é explícito, ou seja, o programador é responsável por identificar o paralelismo e implementar o algoritmo utilizando chamadas aos comandos da biblioteca MPI. São implementadas rotinas bloqueantes e não bloqueantes para enviar e receber mensagens. A rotina que envia a mensagem no modo bloqueante não retorna enquanto a mensagem contida no buffer não estiver segura. No modo não bloqueante a rotina que envia a mensagem pode retornar enquanto a mensagem ainda está volátil. A biblioteca disponibiliza tipos primitivos para o envio de dados em mensagens quanto rotinas para o encapsulamento de estruturas complexas.

MPI possui grupos de processos e rotinas para o gerenciamento dos grupos. Os grupos podem ser usados para duas funções distintas. Na primeira os grupos especificam os processos envolvidos em uma operação de comunicação coletiva, como um *broadcasting*. Toda a comunicação ocorre

dentro e através dos grupos. Na segunda eles podem ser usados para introduzir o paralelismo dentro da aplicação, onde diferentes grupos realizam diferentes tarefas. Cada grupo pode possuir códigos executáveis diferentes ou o mesmo código.

Recentemente, o padrão MPI foi atualizado pelo comitê MPI. A nova versão, MPI–2, contém tanto modificações no núcleo do *daemon* MPI quanto novas características, incluindo I/O paralela, operações de acesso remoto a memória e criação dinâmica de processos [35].

2.4.3.2 PVM

Outro modelo de troca de mensagens independente de arquitetura desenvolvido visando permitir o uso transparente de NOWs é chamado *Parallel Virtual Machine* (PVM) [75]. PVM permite que um conjunto heterogêneo de computadores conectados por uma rede sejam utilizados pelo programador como um único computador distribuído [7].

Esse padrão provê um conjunto de primitivas para a criação de processos e comunicação que são incorporada às linguagens de programação C e Fortran. Utilizando PVM, o programador fica encarregado de toda a decomposição, alocação e comunicação explicitamente, sendo que a plataforma realiza de forma transparente as conversões necessárias entre os diferentes formatos de dados que podem coexistir no ambiente utilizado [7].

2.4.4 Linguagens Gráficas – Ferramentas para Geração de Código

As linguagens de modelagens apresentadas anteriormente possibilitam geração de código executável a partir da compilação direta da linguagem de entrada, que se apresenta como código fonte textual. As ferramentas apresentadas nessa seção utilizam uma linguagem de modelagem com forte apelo gráfico, buscando explicitar elementos como topologia da aplicação e mapeamento entre processos e processadores. Por modelarem a aplicação em um nível ainda mais alto do que o das linguagens de programação paralela, normalmente os modelos gerados por essas ferramentas são primeiramente traduzidos para alguma linguagem intermediária, a qual é efetivamente compilada para a geração de um programa executável. Assim, para cada linguagem dessas linguagens de modelagem, existe um tradutor e uma ferramenta de edição de modelos associada.

Linguagens gráficas simplificam a definição da comunicação por utilizarem descritores em um nível mas alto, intuitivo e estruturado que o possibilitado por linguagens textuais. A decomposição da tarefa entre os elementos de computação e mapeamento é explícita, sendo o sistema

descrito como um grafo, onde os elementos de processamento são representados por vértices e o fluxo de dados por arestas. A seguir, são discutidos alguns exemplos de maior destaque encontrados na literatura.

2.4.4.1 Frameworks

A ferramenta *Frameworks* [68] é um dos primeiros trabalhos a utilizar recursos gráficos para a modelagem da aplicação. O sistema gerado consiste em módulos que se comunicam através de *Remote Procedure Calls* (RPC), sendo C a linguagem utilizada em todo o sistema gerado. Os construtores oferecidos baseiam–se na idéia de fluxograma de dados, com símbolos que representam estruturas de divisão da massa de dados, escalonamento e sincronização inseridos na modelagem. Nodos usados para a modelagem têm uma semântica variável e arestas representam canais de comunicação. Assim, a ferramenta facilita a modelagem da topologia do sistema.

2.4.4.2 Enterprise

A ferramenta *Enterprise* [64, 43] é uma evolução do sistema *Frameworks.* Ela consiste em uma interface gráfica, uma biblioteca de códigos, um pré–compilador e um gerente de execução. A metáfora utilizada para modelagem é a de um escritório, substituindo termos como *pipelines, master, slave* por *individuals, departments, receptionists, etc.*.

O sistema suporta balanceamento de carga e distribuição dinâmica de trabalho. *Enterprise* assume a existência de um sistema de arquivos global (*e.g.* NFS), não oferecendo de maneira nativa mecanismos para controle de I/O concorrente em arquivos.

As bibliotecas implementadas usam uma interface genérica para um *kernel* de troca de mensagens. Inicialmente, a ferramenta comportava as plataformas de comunicação ISIS [9] e NMP [53]. Posteriormente, foi incluído suporte a PVM e Concern/C [5]. As facilidades oferecidas pelo plataforma ISIS permitem aos sistemas gerados através da ferramenta *Enterprise* um limitado grau de tolerância a falhas. Ainda, PVM e Concern/C oferecem suporte à utilização de ambientes heterogêneos.

2.4.4.3 HeNCE

HeNCE (Heterogeneous Network Computing Environment) [7, 3] é um ambiente gráfico de programação criado para auxiliar o desenvolvimento de programas paralelos utilizando PVM

como plataforma de comunicação. A ferramenta permite a programação, compilação, execução e depuração, assim como a configuração do ambiente paralelo virtual oferecido pela plataforma.

A metodologia de utilização da ferramenta envolve as seguintes etapas: (i) criar um grafo descrevendo a semântica da computação paralela; (ii) configurar a máquina virtual para a execução do programa; (iii) construir os executáveis para os vários nodos, considerando a arquitetura; (iv) executar o programa na máquina virtual; (v) opcionalmente, o programador pode visualizar uma animação da troca de mensagens entre os nodos durante ou após a execução.

Nodos representam rotinas seqüenciais providas pelo usuário, ou estruturas de controle de fluxo de dados entre os nodos (*e.g. Loops*, *Condicionais*, *Fans*) providas pela linguagem de modelagem. Um nodo *Condicional* define para qual dos subgrafos os dados devem ser direcionados. O construto *Loop* define a execução de múltiplas iteração usando como corpo do laço o subgrafo associado. O construto *Fan* permite a criação dinâmica de subgrafos.

Arestas representam os canais de comunicação utilizados entre os nodos, sendo que HeNCE não utiliza nenhum rótulo associado às arestas. Segundo a semântica associada à linguagem de modelagem utilizada, um nodo não executa até que seus pais tenham executado, dado que a entrada em um nodo é obtida a partir de seus ancestrais no grafo.

2.4.4.4 CODE

A ferramenta *CODE (Computation–Oriented Display Environment)* [12] [13] provê a geração de código tanto utilizando MPI quanto PVM, permitindo a geração de programas paralelos para arquiteturas com memória compartilhada e memória distribuída [11].

Ela utiliza uma linguagem de modelagem baseada em grafos de fluxo de dados, na qual computações representadas por nodos são conectadas por arestas, que representam canais FIFO para a comunicação entre um nodo e outro [57]. Nodos podem possuir mais de um canal de entrada, sendo que a definição de qual canal é utilizado é feita rotulando–se a aresta que faz a ligação. Opcionalmente, pode existir uma regra de acionamento a cada computação, indicando uma condição que permite a execução do nodo e as variáveis receber os valores de entrada. Assim como na ferramenta HeNCE, além dos nodos representando processamento sobre os dados existem nodos para o controle de repetições e criação dinâmica de novos subgrafos.

2.4.4.5 DPnDP

DPnDP (Design Patterns and Distributed Processes) [70, 71] é um dos primeiros sistemas a apresentar a implementação e uso de *design patterns* parametrizáveis independentes da aplicação como uma biblioteca extensível de esqueletos de código para a geração de aplicações paralelas. Uma aplicação é composta usando–se um ou mais *design patterns*, que podem ser combinados com outras partes de código inclusive fazendo uso de primitiva de sincronização e comunicação em baixo nível.

Cada *design pattern* fornecido pela ferramenta provê uma interface padrão. Utilizando essa interface padrão, novos *design patterns* podem ser adicionados no sistema incrementalmente, tornando o sistema extensível. Esse é um dos principais focos da ferramenta. Uma característica dos *design patterns* utilizados pela ferramenta DPnDP é o fato de serem livres do contexto (*context insensitive*). Isso significa que uma implementação ou uso de um *design pattern* não faz suposição ou restrição nenhuma sobre a implementação dos demais.

Em DPnDP, uma aplicação paralela é representada graficamente por um grafo dirigido, onde os nós interagem entre si através de troca de mensagens. Cada módulo possui um ou mais manipuladores de mensagens que recebem mensagens de outros nós e invocam o código seqüencial dentro do módulo para processar estas mensagens apropriadamente. Cada vértice do grafo é associado a um modelo de código da computação seqüencial escrito em C ou C++. Alternativamente, uma aresta do grafo da aplicação pode representar um *design pattern*. A interface de um *design pattern* é indistingüível da de um módulo, *i.e.*, em ambos os casos os outros nós interagem usando troca da mensagem através de uma interface comum. Entretanto, a estrutura interna de um *design pattern* pode ser um subgrafo multi–nó. Alguns destes nós podem inclusive ser outros *design patterns*.

2.4.4.6 CO_2P_3S

A ferramenta CO_2P_3S (*Correct Object–Oriented Pattern–based Parallel Programming System*) [76, 4] gera código Java também utilizando *design patterns*, visando ambientes de execução distribuídos. A linguagem de modelagem utilizada expressa principalmente a topologia da aplicação. A ferramenta possui recursos para monitoração do processamento e comunicação entre os processos, possibilitando ao usuário modificar o modelo em busca de um melhor balanceamento. Ainda, está associada a esse ambiente uma ferramenta para a edição dos *patterns* utilizados chamada MetaCO_2P_3S. A metodologia de utilização da ferramenta baseia–se em cinco etapas:

(i)identificação de um ou mais *design patterns* para a parte paralela da aplicação; (ii) configuração do *design pattern*; (iii) execução do *framework* que indica onde o usuário deve inserir o código para o processamento dos dados; (iv) análise dos resultados (e possível modificação do modelo) e (v) retomar passo anterior ou mesmo inicial, caso o *design pattern* utilizado não se mostre adequado.

As linguagens, bibliotecas e ferramentas apresentadas até então utilizam modelos como uma descrição do sistema visando gerar um programa executável. Mesmo quando elaborada com cuidado, determinadas características na solução proposta podem carregar implicitamente algum aspecto negativo, que leve o comportamento do sistema a um estado indesejado.

Para evitar tais situações, é natural a realização de um conjunto de testes e simulações sobre o sistema gerado. Porém, é comum afirmar que a realização não–exaustiva de testes pode provar apenas a presença de erros, não as sua ausência (conforme enunciado pela primeira vez por Edsger Dijkstra em [20]). Isso motiva a utilização de outras metodologias complementares na análise de aplicações paralelas, dentre as quais a verificação de modelos (ou *model checking*) tem sido um grande foco de pesquisas e desenvolvimento.

2.5 Verificação de Modelos

Basicamente, verificação de modelos consiste na aplicação de algoritmos executados através de ferramentas computacionais objetivando verificar a corretude de um sistema. O usuário provê uma descrição do sistema através de um modelo (o qual é utilizado para gerar seus comportamentos possíveis) e uma descrição de sua especificação (uma descrição de seu comportamento esperado). A ferramenta então realiza uma busca exaustiva no espaço de estados possibilitado pelo modelo, procurando por um comportamento (uma seqüência de estados) que viole a especificação fornecida. A metodologia de uso de uma ferramenta de verificação pode ser ilustrada como na Figura 1.

Caso seja encontrado um comportamento que viole especificação (um erro), a ferramenta provê um contra–exemplo. O contra–exemplo consiste num cenário no qual o modelo apresenta um comportamento indesejado, indicando que o modelo é falho e necessita de correções. Ainda, o contra–exemplo pode indicar que a especificação (ou sua descrição) está errada quando, por exemplo, a ferramenta retorna como contra–exemplo um cenário que não fazia parte do significado

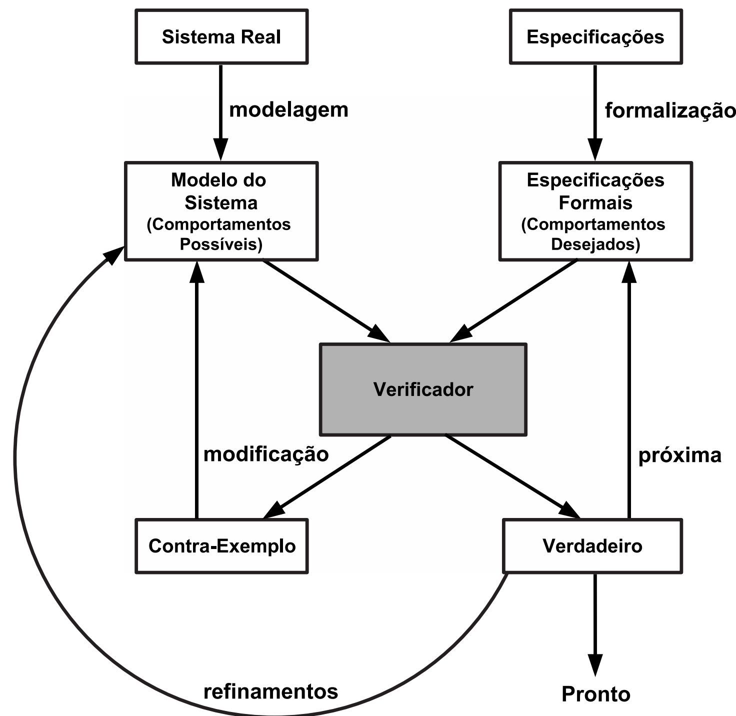


Figura 1: Metodologia de uso de um Verificador.

atribuído à especificação. O contra–exemplo permite ao usuário localizar o erro e reparar sua estrutura, modificações as quais devem ser mapeadas à implementação. Caso nenhum erro seja encontrado, o usuário pode refinar a descrição do modelo, isto é, tomar um conjunto menor de abstrações de forma que o modelo torne–se mais concreto, e reiniciar o processo de verificação.

Existem limitações para a aplicação de verificadores de modelos, impedindo sua utilização para qualquer tipo de sistema. Este método de análise é apropriado para aplicações intensivas em controle, como comunicação entre componentes. Não é adequada para aplicações intensivas em dados, uma vez que o tratamento de dados usualmente introduz um espaço de estados infinito. Outro aspecto a ser considerado é que determinar o conjunto apropriado de abstrações sobre o sistema para construção do modelo requer experiência. Abstrações erradas podem levar o comportamento do modelo a perder a correspondência ao sistema real. Ainda, como a verificação é realizada sobre o modelo construído, qualquer resultado obtido é tão bom quanto o modelo apresentado.

2.5.1 Lógica Temporal

Lógicas especiais foram definidas objetivando fornecer a expressividade necessária para a especificação de propriedades e verificação de modelos. A mais amplamente estudada é a lógica temporal, que foi introduzida na ciência de computação com propósitos de especificação e verificação por Amir Pnueli (1977). As origens de lógica temporal remontam ao campo da filosofia, onde A. Prior é apontado [46] como o criador na década de 60. Lógica temporal estende a lógica clássica através da inclusão de novos operadores temporais, os quais permitem a formulação de propriedades como "A e B são válidos *depois* de C ocorrer" e "*no futuro* A é válido", por exemplo. Existem dois tipos principais de lógicas temporais com tempo discreto: ramificada e linear.

Computation Tree Logic (CTL) é uma lógica de tempo ramificado sobre árvores de computações. Lógica de tempo ramificado permite ao usuário escrever fórmulas que incluem sensibilidade a escolhas possíveis durante a execução. Ela permite considerações acerca de possíveis seqüências de estados a partir de um estado inicial. Seus operadores consistem de operadores de caminhos seguidos imediatamente por um operador temporal. Os quantificadores de caminho são A ("para todos os caminhos") e E ("para algum caminho").

Lógica Temporal Linear (LTL) é uma linguagem de asserções sobre computações, permitindo o estabelecimento de propriedades sobre seqüências de execuções de um sistema. Um programa satisfaz uma fórmula LTL se todas as suas possíveis computações satisfazem essa fórmula.

CTL e LTL diferem pela maneira como interpretam as seqüências de estados possíveis. Em CTL, uma seqüência de estados é interpretada conforme uma árvore, onde os filhos de um estado são os demais próximos estados alcançáveis a partir dele. Assim, durante uma execução o modelo "ramifica" os estados conforme as possibilidades do modelo. Em LTL, cada nova possibilidade representa uma nova seqüência linear de estados. A Figura 2 apresenta o comportamento do autômato (Figura 2.a) segundo a interpretação em CTL (Figura 2.b) e LTL (Figura 2.c):

2.5.2 Lógica Temporal Linear

O ponto inicial para a utilização de LTL na verificação formal de um modelo é a definição de um conjunto de proposições atômicas sobre variáveis desse modelo. Exemplos de proposições atômicas são "x é maior que 0", "x é igual a 1", dada uma variável x. Em princípio, proposições atômicas podem envolver tanto variáveis quanto constantes (0,1,2,...), funções (*e.g.* max, min, ...) e predicados (*e.g.* $x == 2$, $x \ mod \ 2 = 0$). O conjunto de proposições atômicas será denotado por AP. Determinar o conjunto inicial de proposições atômicas pode ser considerado como o

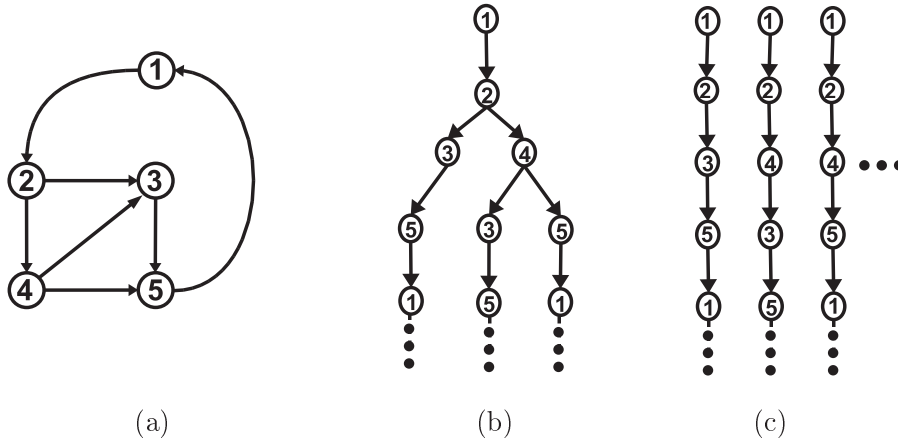


Figura 2: Autômato e interpretação segundo CTL e LTL para o comportamento.

primeiro passo de abstração. Por exemplo, se é fixado que um certo subconjunto das variáveis do sistema não podem ser referenciadas em AP, então nenhuma propriedade pode ser elaborada sobre essas variáveis e, conseqüentemente, propriedades sobre elas não podem ser verificadas.

A definição a seguir apresenta o conjunto básico de fórmulas que podem ser expressas em LTL:

Definição 1 - Sintaxe da LTL:

Seja AP um conjunto de proposições atômicas:

1. Para todo $p \in AP$, p é uma fórmula.
2. Se ϕ é uma fórmula, então $\neg\ \phi$ é uma fórmula.
3. Se ϕ e ψ são fórmulas, então $(\ \phi \vee \psi\ )$, $(\ \phi \wedge \psi\ )$ e $(\ \phi \rightarrow \psi\ )$ são fórmulas.
4. Se ϕ é uma fórmula, então $X\phi$, $F\phi$ e $G\phi$ são fórmulas.
5. Se ϕ e ψ são fórmulas, então $[\ \phi\ U\ \psi\ ]$ é uma fórmula.

O conjunto de fórmulas elaborado de acordo com essas regras denotam sentenças em LTL. Note que os três primeiros itens são relativos à lógica proposicional. Os operadores temporais introduzidos são o X (pronunciado *próximo*), G (pronunciado "sempre" ou "globalmente"), F (pronunciado "eventualmente" ou "no futuro") e U (pronunciado *até*)

Alternativamente, a sintaxe de LTL pode ser dada na forma de Backus–Naur (BNF) como:

Definição 1.b - Sintaxe da LTL na forma BNF:

Para $p\ \in AP$, o conjunto de fórmulas LTL é definido por.

$$\phi ::= p \mid \neg\phi \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid X\phi \mid F\phi \mid G\phi \mid (\phi U \phi)$$

Em LTL, assim como na lógica clássica, os operadores booleanos de conjunção, implicação e equivalência podem ser também definidos como:

$$\begin{aligned}
&\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)\\
&\phi \rightarrow \psi \equiv \neg\phi \vee \psi\\
&\phi \leftrightarrow \psi \equiv \ (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)\\
&true \equiv \phi \vee \neg\phi\\
&false \equiv \neg true
\end{aligned}$$

A hierarquia de precedência dos operadores de LTL segue o mesmo esquema da lógica clássica. Operadores unários têm precedência maior do que operadores binários. Os operadores $\neg$ e X têm a mesma precedência. O operador temporal U tem precedência maior do que $\vee$, $\wedge$ e $\rightarrow$. O operador $\rightarrow$ tem precedência menor do que $\vee$ ou $\wedge$, os quais tem mesma precedência. A inclusão balanceada de parênteses pode ser utilizada para forçar outras ordens de avaliação. Assim, a sentença $((\neg\phi) \rightarrow \psi)U((X\phi) \wedge (F\psi))$ pode também ser escrita $(\neg\phi \rightarrow \psi)U(X\phi \wedge F\psi)$.

Formalmente, uma sentença LTL especifica um padrão para uma seqüência de estados. Intuitivamente, $X\phi$ representa que ϕ é válido no próximo estado, $F\phi$ representa que ϕ é válido no estado atual ou para algum estado no futuro. O significado formal de LTL é definido em termos de um *modelo*.

Definição 2 – Modelo LTL [46] :

Seja um modelo LTL $\mathcal{M}$ uma tripla $\mathcal{M} = (S, R, Label)$ onde:

1. S é um conjunto não vazio e enumerável de estados.
2. $R : S \rightarrow S$ associando a $s \in S$ seu estado sucessor $R(s)$;
3. $Label : S \rightarrow 2^{AP}$, associando a cada estado $s \in S$ as proposições atômicas $Label(s)$ que são válidas em s.

Para todo estado $s \in S$, $R(s)$ é o estado subseqüente a s alcançado através da aplicação de R. A função R age como um gerador infinito de seqüências de estados[3]. A função $Label(s)$ indica quais proposições atômicas são válidas para cada estado em $\mathcal{M}$. Se para o estado s tem–se

[3]Note que seqüências de estado infinitas podem ser geradas sobre um conjunto finito de estados, desde que a aplicação de R produza um laço como o observado na Figura 2a.

$Label(s) = \phi$ isso significa que ϕ é válida em s. Um estado s para o qual a proposição ϕ é válida, ou seja, $\phi \in Label(s)$, é muitas vezes referenciado como um ϕ-*state*.

Para o estado $s \in S$, o estado $R(s)$ é o único próximo estado de s alcançado a partir da aplicação de R. Uma característica importante da função R é o fato dela agir como um gerador para infinitas seqüências de estados tal como $s, R(s), R(R(s)), R(R(R(s))), \ldots$, sendo essas seqüências os elementos fundamentais a que se referem as sentenças LTL.

O significado de fórmulas em LTL são definidos através de uma relação de satisfação (denotada por $\models$) entre um modelo $\mathcal{M}$, um de seus estados s, e uma fórmula ϕ. O sentido da relação $(\mathcal{M}, s, \phi) \in \models$ é representado em notação infixada por $\mathcal{M}, s \models \phi$

Definição 3 – A Semântica de LTL:

Seja $p \in AP$ uma proposição atômica, $\mathcal{M} = (S, R, Label)$ um modelo LTL, $s \in S$, e ϕ e ψ fórmulas LTL. A relação de satisfação $\models$ é definida por:

$s \models p$	iff $p \in Label(s)$
$s \models \neg\, \phi$	iff $\neg(s \models \phi)$
$s \models \phi \vee \psi$	iff $(s \models \phi) \vee (s \models \psi)$
$s \models \phi \wedge \psi$	iff $(s \models \phi) \wedge (s \models \psi)$
$s \models \phi \rightarrow \psi$	iff $(s \models \phi) \rightarrow (s \models \psi)$
$s \models X\phi$	iff $R(s) \models \phi$
$s \models \phi\ U\psi$	iff $\exists j \geq 0 \mid R^j(s) \models \psi \wedge (\forall\, 0 \leq k < j.R^k(s) \models \phi)$
$s \models F\phi$	iff $\exists j \geq 0 \mid R^j(s) \models \phi$
$s \models G\phi$	iff $\forall j \geq 0 \mid R^j(s) \models \phi$

Aqui, $R^0(s) = s$, e $R^{n+1}(s) = R^n(R(s))$ para todo $n \geq 0$. Se $R(s) = s'$, o estado s' é chamado de *sucessor direto* de s. Se $R^n(s) = s'$ para $n \geq 1$, o estado s' é um *sucessor* de s. Se $\mathcal{M}, s \models \phi$ é dito que o modelo $\mathcal{M}$ satisfaz ϕ no estado s. Expresso de outra maneira, a fórmula ϕ é válida para o estado s do modelo $\mathcal{M}$.

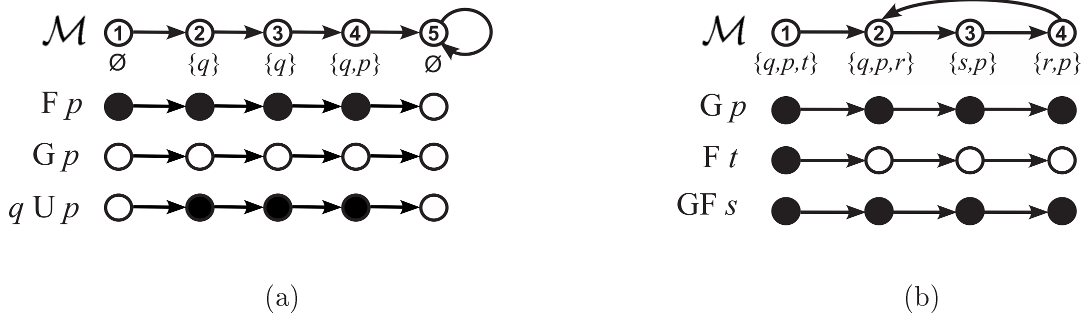
Ainda, pela definição, $F\phi$ é válido para s se e somente se existir algum estado sucessor de s (não necessariamente direto) onde ϕ é válido, ou se ϕ for válido em s. A fórmula $G\phi$ é válida em s se e somente se para s e todos os seus sucessores a fórmula ϕ é válida. Baseado ainda na semântica apresentada, pode–se também definir os operadores temporais G e F como:

$F\phi \equiv true\ U\ \phi$

$G\phi \equiv \neg F \neg \phi$

Para a primeira sentença, uma vez que *true* é válido em todos os estados, $F\phi$ define que ϕ é verdade em algum ponto do futuro. Para a segunda sentença, uma vez que não exista nenhum estado no futuro onde $\neg\phi$ seja válido; então ϕ é válido em todos os pontos.

A Figura 3 [46] exemplifica a utilização de LTL sobre seqüências de estados. Círculos representam estados, e a aplicação da função R a um estado é representada por uma aresta (*i.e.* existe uma aresta de s para s' se e somente se $s' = R(s)$). A seqüência superior de estados representa o modelo LTL. Os rótulos associados a cada estado representam quais proposições atômicas são válidas naquele ponto do processamento. Abaixo de cada modelo encontram–se marcados os estados onde a sentença proposta à esquerda de cada seqüência é verdadeira. Para primeira seqüência (Figura 3.a), uma vez que o modelo tenha alcançado o estado 5, nenhuma proposição é válida, e o modelo permanece nesse estado indefinidamente. Para o segundo modelo (Figura 3.b), cada seqüência pode retornar ao estado 2 infinitas vezes. Observe que os modelos definem computações infinitas sobre seqüências finitas de estados



(a) (b)

Figura 3: Exemplos de interpretação para sentenças LTL.

Pode–se comparar e combinar sentenças LTL a partir da definição de uma lista de axiomas que respeitem a semântica definida. Formalmente, o axioma $\phi \equiv \psi$ é chamado válido se e somente se, para todo modelo LTL $\mathcal{M}$ e estado s em $\mathcal{M}$:

$$\mathcal{M}, s \models \phi \text{ se e somente se } \mathcal{M}, s \models \psi$$

A aplicação de axiomas a uma certa fórmula resulta em outra fórmula de mesmos significado e validade, *i.e.*, os axiomas não mudam a semântica da fórmula. A seguir, é apresentada uma lista não exaustiva de axiomas válidos. Uma listagem completa de axiomas para LTL existe, mas foge do escopo desse trabalho.

Dualidades	$\neg G\phi \equiv F\neg\phi$	Absorção	$FGF\psi \equiv GF\psi$
	$\neg F\phi \equiv G\neg\phi$		$GFG\psi \equiv FG\psi$
	$\neg X\phi \equiv X\neg\phi$	Distributividade	$X(\phi U\psi) \equiv (X\phi)U(X\psi)$
Idempotência	$GG\phi \equiv G\phi$	Expansão	$\phi U\psi \equiv \psi \vee [\psi \vee X(\phi U\psi)]$
	$FF\phi \equiv F\phi$		$F\phi \equiv \phi \vee XF\phi$
	$\phi U(\phi U\psi) \equiv \phi U\psi$		$G\phi \equiv \phi \vee XG\phi$
	$(\phi U\phi)U\psi \equiv \phi U\psi$		

Axiomas são utilizados principalmente para simplificar sentenças obtidas a partir da combinação de partes mais simples, visando a elaboração de propriedades correspondentes à especificação do sistema. A utilização de sentenças LTL dessa forma para a definição de propriedades de um modelo requer certo grau de habilidade por parte do usuário, tanto na construção da propriedade quanto na interpretação dos resultados obtidos. Para facilitar tal atividade, torna–se comum a utilização de padrões pré–definidos para sentenças de uso freqüente.

2.5.3 Padrões para Sentenças LTL

A literatura apresenta vários trabalhos onde são introduzidos padrões reutilizáveis para propriedades em lógica temporal, facilitando a utilização e evitando erros de especificação.

A classificação *savety–liveness* proposta por Lamport em 1977 é uma das classificações mais importantes encontrada na literatura, por adequar–se a uma ampla gama de propriedades. Ela diferencia dois grandes grupos principais: (i) *safety*: algo de "ruim" nunca acontece durante a execução de um sistema; e (ii) *liveness:* algo de "bom" deve ocorrer em cada uma das execuções do sistema. Na categoria *safety* enquadram–se propriedades como "dois processos não podem estar na seção crítica ao mesmo tempo" ou "o sistema nunca entra em *deadlock*". Propriedades de *liveness* comuns são terminação ("um sistema deve chegar ao término em algum momento") e *starvation freedom* ("sempre deve ocorrer progresso em um processo").

Em [29, 30], é introduzido um amplo conjunto de padrões para sentenças em lógica temporal. Os padrões são organizados hierarquicamente baseado em sua semântica, como ilustrado na Figura 4 [29], e organizam–se em dois grupos principais: ocorrência (*Occurrence*) e ordenação (*Order*).

Dentro de cada grupo existem ainda quatro subdivisões. Dentro da categoria *Existence* encontram–se as seguintes possibilidades:

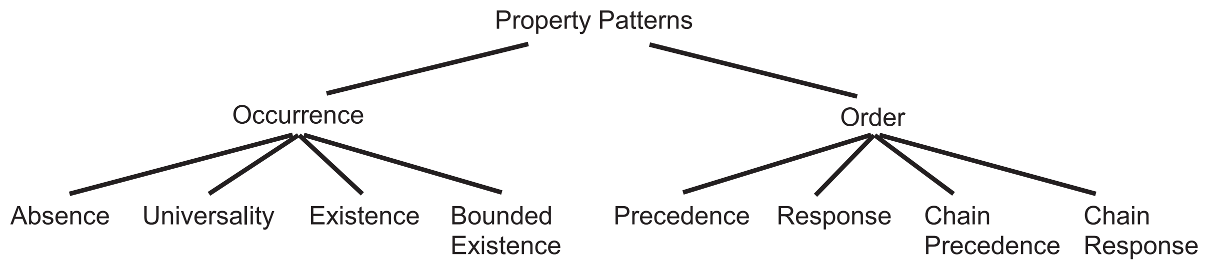


Figura 4: Hierarquia de padrões para sentenças LTL.

- ***Absence*:** a proposição não ocorre[4] dentro das execuções;
- ***Universality*:** a proposição ocorre em todos os estados durante as execuções;
- ***Existence*:** a proposição deve ocorrer em pelo menos um estado dentro de cada uma das execuções;
- ***Bounded Existence*:** a proposição deve ocorrer k vezes em cada uma das execuções.

Para a categoria *Order*:

- ***Response*:** a proposição P deve sempre ser seguida da proposição Q em cada uma das execuções;
- ***Precedence*:** a proposição P deve sempre ser precedida pela proposição Q em cada uma das execuções;
- ***Chain Response*:** a seqüência de proposições $P_1, P_2, \ldots, P_n$ deve sempre ser seguida pela seqüência de proposições $Q_1, Q_2, \ldots, Q_m$ em cada uma das execuções;
- ***Chain Precedence*:** a seqüência de proposições $P_1, P_2, \ldots, P_n$ deve sempre ser precedida pela seqüência de proposições $Q_1, Q_2, \ldots, Q_m$ em cada uma das execuções.

Dentro de cada padrão apresentado, pode–se ainda definir os escopos de aplicação para as proposições. Escopos definem as regiões de interesse dentro das quais as proposições apresentadas devem ser válidas. A Figura 5 [29] apresenta os diferentes escopos oferecidos pelos padrões apresentados. Existem cinco tipos de escopos:

- ***Global*:** durante toda a seqüência de estados;
- ***Before Q*:** a seqüência de estados até a proposição Q ser verdadeira;

[4]Utiliza–se "a proposição ocorre dentro das execuções" significando "ela é verdadeira para os estados dentro das execuções".

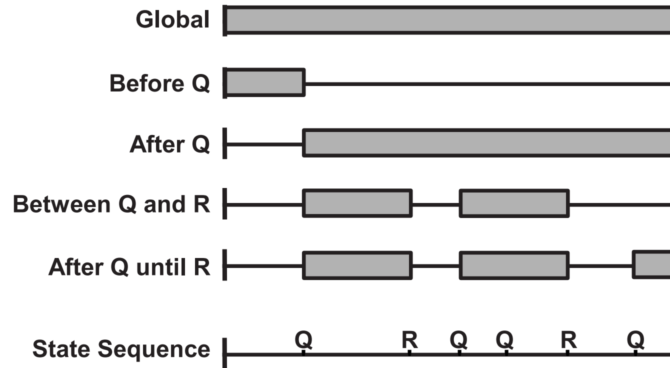


Figura 5: Padrões de escopo para sentenças LTL.

- ***After*** Q**:** a seqüência de estados após a proposição Q ser verdadeira;
- ***Between*** Q ***and*** R**:** dentro da seqüência de estados, o trecho entre as proposições Q e R;
- ***After*** Q ***until*** R**:** dentro da seqüência de estados, o trecho após a proposição Q, até a proposição R ocorrer;

Os padrões propostos em [29, 30] aplicam–se para propriedades expressas sobre as seqüências de estados. Na verificação de determinados modelos, pode ser necessário especificar propriedades sobre as transições entre os estados, ou seja, propriedades sobre os *eventos* que levam às mudanças de estado.

Em [16, 58, 59] eventos em LTL são chamados *edges*, sendo definidos a partir de uma proposição s como:
Up–Edge s : $\uparrow s = (\neg s \ \wedge \ X\ s)$
Down–Edge s : $\downarrow s = (s \ \wedge \ X\ \neg s)$

Um evento $\uparrow s$ denota a mudança de estado de uma variável ou proposição s de *não–ativa* para *ativa*. Um evento $\downarrow s$, por sua vez denota a mudança de estado de *ativo* para *não–ativo*. Pela modo como é expresso em LTL, um evento é detectado no estado anterior em que a proposição torna–se verdadeira, *i.e.*, imediatamente antes da variável ou variáveis usadas na proposição mudarem de valor. Em [59] também são introduzidos padrões de propriedades sobre eventos para uso na verificação de modelos. Eventos são utilizados para representar a aplicação de uma regra em um modelo GGBO, como será visto no Capítulo 3.

2.5.4 Explosão do Espaço de Estados

A maior limitação no uso de verificação de modelos diz respeito ao problema de explosão de estados. Quando da verificação de um modelo, muitas vezes o espaço de estados gerado pela composição do modelo com a sentença LTL pode se tornar muito grande, especialmente se forem envolvidos vários processos concorrentes nesse modelo ou se a sentença LTL for complexa, de tal modo que fica impraticável o tratamento computacional do sistema.

Pode–se tentar contornar esse problema a partir de técnicas para a diminuição do espaço de estados dentro das ferramenta de verificação, ou através da adoção de técnicas de modelagem diferentes por parte do usuário.

Uma das técnicas utilizada pelas ferramentas de verificação é chamada Redução de Ordem Parcial (*Partial Order Reduction*), a qual consiste em reduzir o espaço de estados gerados durante a verificação, selecionando apenas um subconjunto das possíveis intercalações para transições que são executadas independentemente [17]. Dois eventos são ditos independentes se a execução alternada deles resulta no mesmo estado global. Os padrões de sentenças apresentados em [29, 30, 16, 58] foram construídos de forma a possibilitar o uso adequado de Redução de Ordem Parcial na verificação dos modelos.

O usuário, por sua vez pode tratar o problema da explosão de estados utilizando abordagens do tipo dividir –e–conquistar, onde a especificação de sistemas é feita em termos das propriedades de seus componentes, sendo estas últimas verificadas separadamente. A utilização de uma abordagem composicional para a verificação pode ser utilizada de maneira a restringir o tamanho total do modelo sendo verificado em cada etapa, possibilitando tratar de maneira iterativa o problema e permitindo a verificação de sistemas maiores, contornando o problema de explosão de estados. Maiores detalhes sobre tal abordagem podem ser encontrados em [27].

2.6 A Ferramenta SPIN para Verificação de Modelos

PROcess MEta LAnguage (PROMELA) é uma linguagem de especificação formal baseada em processos, sendo utilizada como linguagem de entrada pelo verificador de modelos *Simple Promela INterpreter* (SPIN) [39]. No verificador de modelos SPIN, a partir de um modelo expresso em linguagem PROMELA e propriedades usando LTL é possível verificar especificações para todo o espaço de estados desse modelo. SPIN é considerado um dos verificadores de modelos mais utilizados e eficientes da atualidade, tendo ganho o prêmio de sistema de *software* em 2001

conferido pela ACM (*Association for Computing Machinery*), e apresentando mais de uma década de desenvolvimento.

Uma especificação PROMELA consiste em processos seqüenciais, variáveis locais e globais, e canais de comunicação usados para conectar os processos seqüenciais. Variáveis globais também podem ser usadas na comunicação entre processos, de forma que ambas as primitivas de comunicação citadas na Seção 2.3 podem ser modeladas.

A sintaxe de PROMELA é fortemente influenciada pela linguagem C, apresentando ainda estruturas de controle inspiradas nos comandos guardados de Djikstra, usada para notação abstrata de programas.

Um processo P é definido como:

```
proctype P (parâmetros formais) { definições locais ; comandos; }
```

Esta estrutura define o processo P, algumas variáveis e constantes locais a P, e seu comportamento. Comandos são separados por ponto–e–vírgula (;). Processos são iniciados através da palavra reservada *run*, o qual aparece em uma seção de inicialização indicada por *init*. No exemplo a seguir, instâncias do processo P e Q são criadas e iniciadas:

```
init{
run P (parâmetros locais);
run Q (parâmetros locais);
}
```

Processo podem ser criados tanto de forma estática quanto dinâmica, ou seja, um processo pode ser criado a qualquer momento dentro de uma especificação.

Comandos em PROMELA podem estar habilitados ou bloqueados. Caso um comando esteja bloqueado, a execução do comando é suspensa naquele ponto até que alguma mudança no ambiente o torne ativo. Por exemplo, um comando $(a == b)$; é equivalente a $while(a! = b)$ *do skip*;,em uma situação na qual a e b sejam diferentes. As seguintes estruturas de controle existem em PROMELA:

- Comando vazio, denotado *skip*. Indica que nada deve ser feito, sendo usado para satisfazer estruturas sintáticas ou melhorar a legibilidade/entendimento do modelo.
- Comandos de atribuição (por exemplo, $x = 7$;). Tais comandos são não–bloqueantes.
- Comandos de comparação (*e.g.* $a == b$;).

- Estruturas condicionais. Possuem a forma

  if
  :: guarda$_1$ -> comando(s)$_1$;
  :: guarda$_2$ -> comando(s)$_2$;
  ...
  :: guarda$_n$ -> comando(s)$_n$;
  :: else -> comando(s);
  fi;

  Dentro de uma estrutura condicional, uma das alternativas cuja guarda esteja ativa é selecionada e a lista (não vazia) de comandos correspondentes são executados. Caso mais de uma guarda esteja habilitada em um determinado momento da execução, a escolha é feita de maneira não–determinística. Uma guarda especial *else*, se presente, é considerada habilitada quando todas as demais estiverem bloqueadas. Caso todas as guardas estejam bloqueadas, a execução pára até que pelo menos uma delas torne–se habilitada.

- Estrutura de repetição. Uma estrutura de repetição possui a mesma forma de uma estrutura condicional

  do
  :: guarda$_1$ -> comando(s)$_1$;
  :: guarda$_2$ -> comando(s)$_2$;
  ...
  :: guarda$_n$ -> comando(s)$_n$;
  :: else -> comando(s);
  od;

  Basicamente, essa estrutura representa um *loop* infinito, o qual pode ser terminado com um comando *goto* ou *break* acionado por uma guarda. Novamente, caso nenhuma guarda esteja habilitada, a execução é suspensa.

- Estruturas *Send* e *Receive*. Processos podem ser interconectados via canais unidirecionais de capacidade arbitrária, porém não infinita. Canais são especificados da mesma maneira que variáveis ordinárias; por exemplo, $chan\ c\ =\ [5]\ of\{int, char\};$ define um canal c de capacidade 5, que armazena tuplas do tipo $(int, char)$. Um canal pode ser considerado como um buffer fifo. Operações de escrita em um canal (denotadas por $c!a, b$; com a e b recebendo os elementos da tupla) são não bloqueantes. As operações de leitura

(representadas por $c?a, b;$) são não–bloqueantes enquanto existirem elementos no canal. Adicionalmente, existem operações para inspecionar um canal sem afetar seu estado atual. Canais de capacidade 0 também são permitidos, e seu uso indica que a comunicação entre os processo é realizada de forma síncrona (um processo que realize uma operação sobre um canal síncrono fica bloqueado até que outro processo realize a operação complementar).

A ferramenta de verificação oferece atomicidade em nível de linha de código para cada processo previsto dentro do modelo (*i.e.*, a intercalação pode ser feita executando–se uma linha de cada processo, por exemplo). Também é possível definir seqüências atômicas em uma especificação. Tais seqüências caracterizam–se pela execução em um conjunto de comandos e declarações em um único passo observável, não intercalando a execução dessas operações com a de outros processos. As seqüências atômicas em PROMELA são definidas com o uso da estrutura *atomic*, sendo que as instruções dentro do escopo *atomic* são executadas sem intercalação com os demais processos. Se existirem comandos guardados dentro de uma estrutura *atomic* e em alguma situação nenhum destes estiver habilitado, irá ocorrer a perda de atomicidade da estrutura, acarretando na execução intercalada de comandos do escopo *atomic* com comandos de outros processos.

A verificação de modelo PROMELA é feita a partir de uma interface especial, que permite a elaboração de proposições atômicas e sentenças em LTL. Proposições atômicas somente podem ser elaboradas a partir de variáveis globais. Por exemplo, considere duas proposições atômicas ϕ e ψ, definidas sobre duas variáveis, a e b, onde ϕ é verdadeiro se a e b forem iguais a 1 e ψ é verdadeiro se a e b forem maiores que 2. Dentro da ferramenta, essas proposições são definidas como:

$\#define\ phi\ (a == 1\ \&\&\ b == 1)$

$\#define\ psi\ (a > 2\ \&\&\ b > 2)$

A sintaxe utilizada pela ferramenta para a especificação de propriedades é mostrada na Tabela 1. Assim, a sentença $(\neg\ \phi\ \rightarrow\ \psi)\ U\ (X\ \phi\ \wedge\ F\ \psi)$ anteriormente formulada é expressa como $(!\ phi\ ->\ psi)\ U\ (X\ phi\ \&\&\ <> psi)$.

Uma vez elaborada a sentença, a ferramenta realiza a verificação expandindo o espaço de estados do modelo à procura de um contra–exemplo. Pode–se configurar a ferramenta para parar a verificação com o primeiro contra–exemplo ou para encontrar todos contra–exemplos os

Tabela 1: Sintaxe LTL utilizada pela ferramenta SPIN.

Operador	LTL Genérica	LTL Ferramenta
Sempre	G	[ ]
Eventualmente	F	<>
Next	X	X
Até (*Until*)	U	U
E	$\wedge$	&&
Ou	$\vee$	||
Implicação	$\rightarrow$	$->$
Negação	$\neg$	!

possíveis. Ao final da verificação, são apresentados dados sobre o espaço de estados, número de transições, total de memória utilizada e tempo necessário consumido.

A ferramenta comporta também a simulação de um modelo PROMELA. Simulações não envolvem sentenças LTL. Durante uma simulação, sempre que o modelo encontrar mais de uma possibilidade de prosseguir, um caminho é aleatoriamente escolhido. Alterando–se a semente numérica do algoritmo pseudo–aleatório utilizado, pode–se realizar simulações para diferentes cenários.

Tanto para os contra–exemplos obtidos quanto para os cenários de simulação é possível acompanhar o comportamento do modelo. Para tanto, a ferramenta apresenta uma janela contendo o diagrama de troca de mensagens entre os processos, assim como uma janela indicando a intercalação de instruções entre os de processos e uma janela exibindo os valores assumidos pelas variáveis e canais durante a animação. Ainda, pode–se exibir um diagrama de barras contendo a proporção entre o número de operações executadas de cada processo presente.

3 Gramáticas de Grafos Baseadas em Objetos

Gramática de Grafos baseada em objetos (GGBO) é um formalismo gráfico e declarativo para modelagem de sistemas reativos, oferecendo conceitos básicos de sistemas baseados em objetos, abstrações de não–determinismo e concorrência inerentes à utilização de tais conceitos. Por ser baseada em objeto, carrega vantagens como modularidade, facilidade de reuso de definições e construção de sistemas grandes. Por ser relativamente restrita em suas construções e ter uma semântica formal, permite seu mapeamento para ambientes de análise tais como simuladores [18, 49] e verificadores [25]. GGBO também se presta para o tratamento analítico necessário para avaliação de desempenho [51]. A linguagem de especificação GGBO foi proposta em [26] para modelar sistemas concorrentes reativos.

3.1 A Linguagem GGBO

As gramáticas de grafos [33] fornecem uma forma bastante natural de expressar situações complexas, onde os estados do sistema sob análise são descritos por grafos e os aspectos dinâmicos podem ser capturados pelas regras da gramática. Uma gramática de grafos é composta por

(i) um grafo de tipos, que representa os tipos de vértices e arestas permitidas no sistema;
(ii) um grafo inicial, que representa o estado inicial do sistema e
(iii) um conjunto de regras que descrevem as possíveis mudanças e estado que podem ocorrer em um sistema.

Em [26] é proposta uma restrição de gramática de grafos chamada GGBO para descrever sistemas baseados em objetos.

Em GGBO, um sistema consiste em entidades autônomas, chamadas *objetos*. Os objetos possuem um estado interno definido pelo conjunto de valores assumidos por seus atributos e se comunicam unicamente através de troca assíncrona de mensagens.

A Figura 6 apresenta o modelo de um objeto segundo GGBO. Graficamente, um objeto tem a notação de um retângulo, onde consta seu nome (Obj_N, na Figura 6), um identificador de tipo de objeto (i) e seu conjunto de atributos (*e.g.* atr_A e atr_B). Instâncias de um mesmo tipo de objeto possuem identificadores de tipo idênticos. Atributos de tipos de dados pré–definidos são listados dentro do retângulo, sendo que os atributos que referenciam outros objetos têm a notação de arestas que se ligam a outros objetos ou a identificadores de tipo de objetos. O objeto alvo de uma referência é indicado por um círculo localizado no final da aresta.

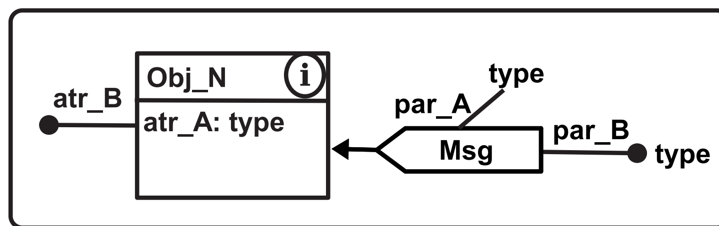


Figura 6: Modelo de objeto em GGBO.

O comportamento de um objeto corresponde às reações executadas por ele ao receber mensagens, reações estas que podem mudar o estado interno do objeto e/ou causar o envio de mensagens tanto para outros objetos quanto para si mesmo.

Graficamente, uma mensagem tem a forma de um polígono como os da mensagem Msg na Figura 6. O destino dessa mensagem é dado por uma seta e os vários parâmetros das mensagem são dados por linhas que ligam estes parâmetros ao polígono da mensagem. Mensagens podem ter como destino apenas um objeto e podem ter como parâmetros valores de tipos de dados pré–definidos (*e.g.* par_A) ou outros objetos (*e.g.* par_B).

Exemplo: Para ilustrar a utilização desse formalismo será adotado um modelo GGBO para um sistema mestre–escravo que implementa um método de Monte Carlo para o cálculo do valor aproximado de π [82]. O método consiste em:

i. Inscrever um círculo de raio $s/2$ em um quadrado de lado s;

ii. Gerar aleatoriamente n pontos dentro do quadrado;

iii. Seja k o número de pontos dentro do quadrado que também estão dentro do círculo;

iv. Calcular $\pi \approx 4 \times k \div n$.

Observe que quanto maior o número de pontos, melhor a aproximação. No modelo, serão utilizados três escravos e um mestre.

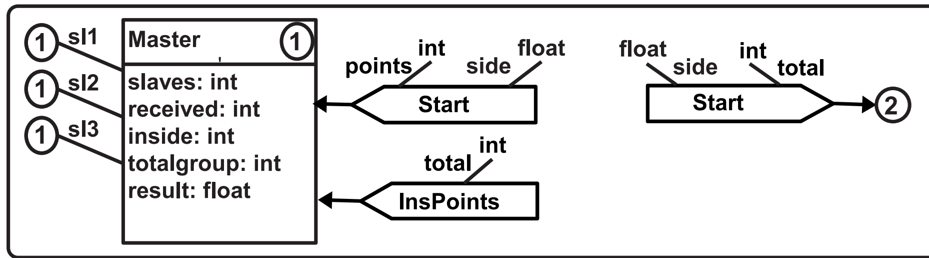
Um bom gerador pseudo–aleatório fornece uma distribuição homogênea de pontos dentro de um domínio. Se um gerador ruim for utilizado nesse modelo, o valor obtido não será próximo o bastante de π. Assim, dependendo do conjunto de pontos aleatorizado (*i.e,* dependendo do gerador utilizado), existem para o modelo computações possíveis na qual o valor final não é uma boa aproximação de π.•

Em um sistema baseado em objetos podem existir vários tipos de objetos. Para descrever um sistema baseado em objetos usando GGBO é necessária a definição do *grafo de tipos* (ou *grafo–tipo*) do sistema, que descreve todos os tipos de objetos que o compõe. O grafo–tipo de um objeto (Figura 7) determina tanto os atributos que um objeto possui quanto as mensagens que qualquer instância de objeto daquele tipo pode receber e enviar, assim como seus parâmetros.

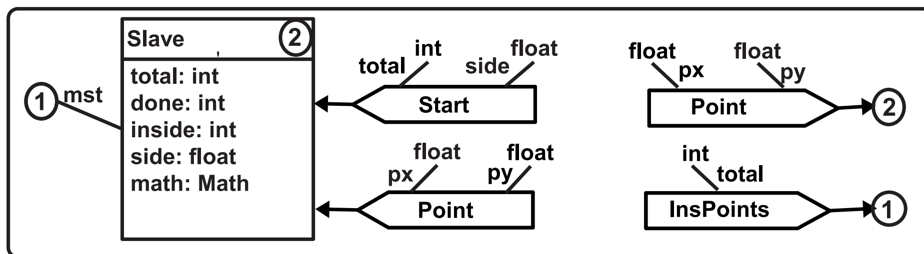
O *grafo inicial* de um sistema descreve todos os objetos, mensagens e valores de atributos que devem existir na situação inicial desejada para o modelo. Todos estes objetos com seus atributos e mensagens são instâncias dos tipos de objetos definidos em grafos–tipo. Os objetos podem ser instanciados estaticamente, a partir do grafo inicial, ou dinamicamente, através da execução de regras que criem novos processos. Como exemplo do grafo inicial, vide Figura 8.

Exemplo: A seguir, os grafos–tipo para os objetos *Master* e *Slave* propostos para o modelo. Observe que junto ao retângulo que representa o objeto são apresentadas as mensagens que este pode receber. As mensagens que o objeto gera estão representadas à direita em cada grafo–tipo, ligadas ao identificador do tipo de objeto ao qual são enviadas.

No grafo–tipo, referências a outros objetos também se ligam a identificadores de tipo, como é o caso das referências $sl1$, $sl2$ e $sl3$ para o objeto *Master* e *mst* para o objeto *Slave*.•



a) Grafo–tipo para o objeto *Master*.



b) Grafo–tipo para o objeto *Slave*.

Figura 7: Grafos–tipo para o modelo proposto.

Em GGBO, o estado de um sistema computacional – que envolve os valores dos atributos, as mensagens existentes e os processos em execução – é representado através de um grafo de estado, e o funcionamento do sistema fica atrelado a um conjunto de regras de transformação que são aplicadas sobre o estado atual do sistema. Dentro de um modelo expresso em GGBO, as modificações no grafo de estado ocorrem onde for encontrado um subgrafo que represente um padrão previsto no lado esquerdo de uma regra, considerando ainda a condição associada a esta regra, caso exista. A identificação de existência de um subgrafo e uma condição associada a uma regra é chamado *ocorrência* ou *match*.

Em uma gramática de grafos, cada regra $r : L \xrightarrow{C} R$ especifica uma mudança de estado no sistema que ocorre da seguinte forma:

- Todos os itens que estão do lado esquerdo da regra L devem estar presentes no estado atual de um sistema para possibilitar a aplicação da regra;

- C, quando presente, especifica uma condição sobre os valores dos atributos presentes em L também necessários para a aplicação da regra;

- Todos os itens que são mapeados de L para R através do mapeamento r são preservados;

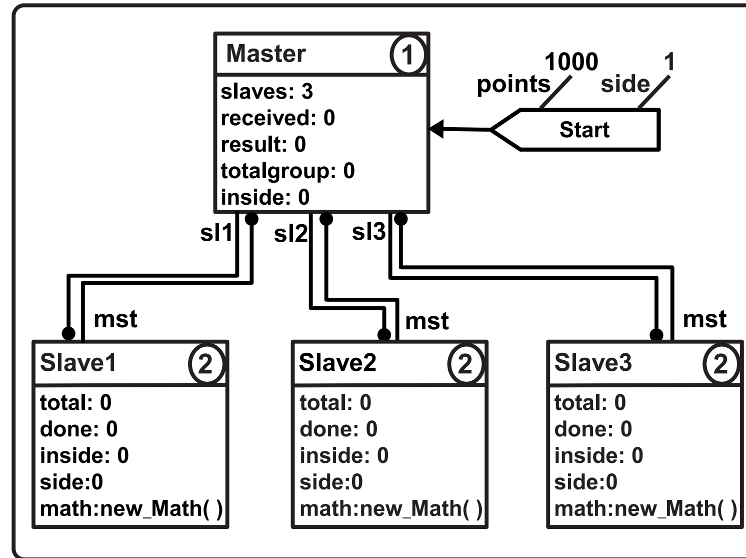


Figura 8: Uma exemplo de grafo inicial.

- Todos os tens que não são mapeados de L para R são apagados do estado atual;
- Todos os tens que estão em R e não estão em L são adicionados para a obtenção do novo estado.

As regras de um tipo de objeto são as que apresentam, no lado esquerdo da regra, uma mensagem sendo recebida por um objeto do tipo em questão. Esta regra especifica a reação de um objeto daquele tipo à recepção daquela mensagem. No lado direito da regra, esta mensagem terá sido consumida, atributos do objeto podem mudar de valor e novas mensagens podem ser geradas, indicando a continuidade do processo. Cada regra descreve o tratamento de apenas uma mensagem. Todas as ações descritas em uma mesma regra ocorrem de forma atômica e são chamadas *eventos*. Um objeto somente pode enviar uma mensagem a outro objeto se existir pelo menos uma referência ao destino no lado direito da regra. Esta referência pode ser tanto um atributo do objeto quanto um parâmetro da mensagem que disparou a regra.

As regras de uma GGBO permitem modelar a concorrência e o não–determinismo. A concorrência é modelada pela possibilidade de aplicação de mais de uma regra em uma mesma situação, quando essas regras não forem conflitantes. Duas regras são conflitantes se estas consomem ou sobrescrevem os mesmos itens. A concorrência pode ocorrer envolvendo objetos diferentes (concorrência externa) ou um mesmo objeto (concorrência interna – quando um objeto pode tratar diversas mensagens ao mesmo tempo). O não determinismo é modelado na escolha

da regra para aplicação. Caso mais de uma regra puder ser aplicada em uma situação, uma destas é escolhida não–deterministicamente para executar.

Exemplo: As Figuras 9 e 10 apresentam respectivamente regras para os objetos *Master* e *Slave* definidos anteriormente. Será utilizada a notação *NomeObjeto_NomeRegra* para referenciar as regras apresentadas.

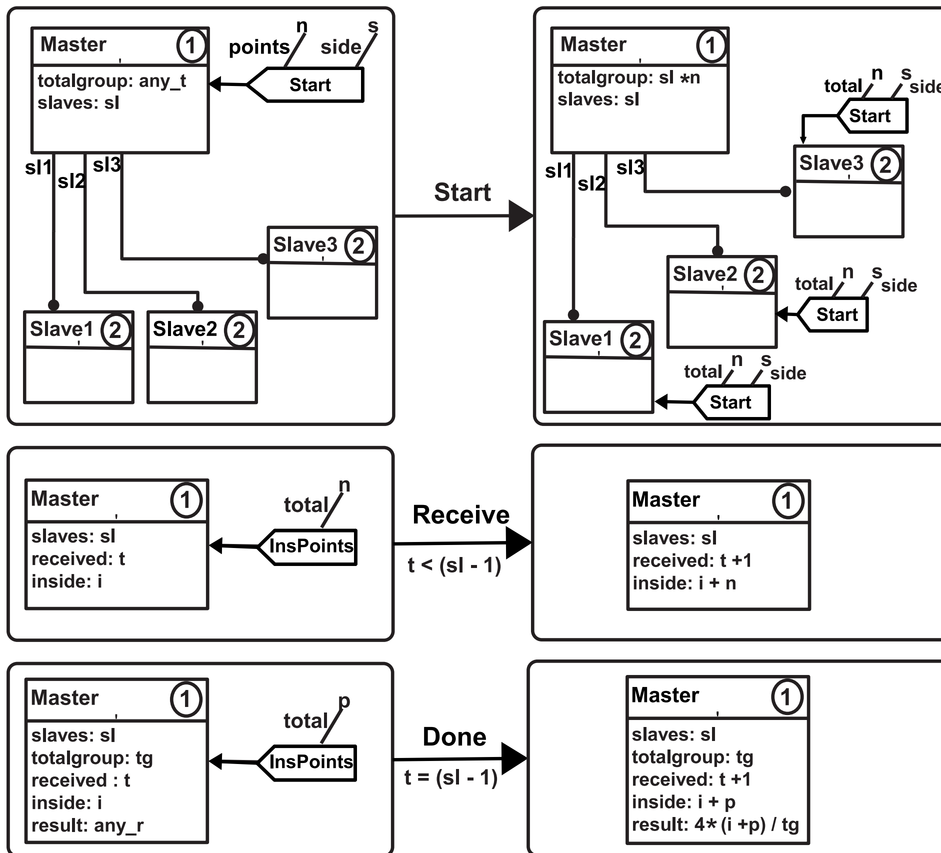


Figura 9: Regras para o objeto *Master*.

Master_ Start é a primeira regra aplicada ao sistema, por ser a única que possibilita um *match* no grafo inicial. Nela, consumindo a mensagem *Start*, o objeto *Master* envia para todos os escravos referenciados uma mensagem *Points*, que indica em n quantos pontos devem ser gerados aleatoriamente. A regra *Master_ Receive* apresenta o comportamento de *Master* recebendo respostas *InsPoints*, acumulando no atributo *inside* o total de pontos inscritos randomizados pelo *Slave* originador. Recebendo a última mensagem de resposta de um escravo (regra *Master_ Done*), *Master* finalmente calcula o valor aproximado de *pi*. A diferenciação entre a escolha

de $Master_Receive$ e $Master_Done$ é feita através da guarda associada a cada regra, que envolve uma comparação entre o número total de participantes (valor *sl* para o atributo *slaves*) e o número de mensagens recebidas (valor *t* para o atributo *received*).

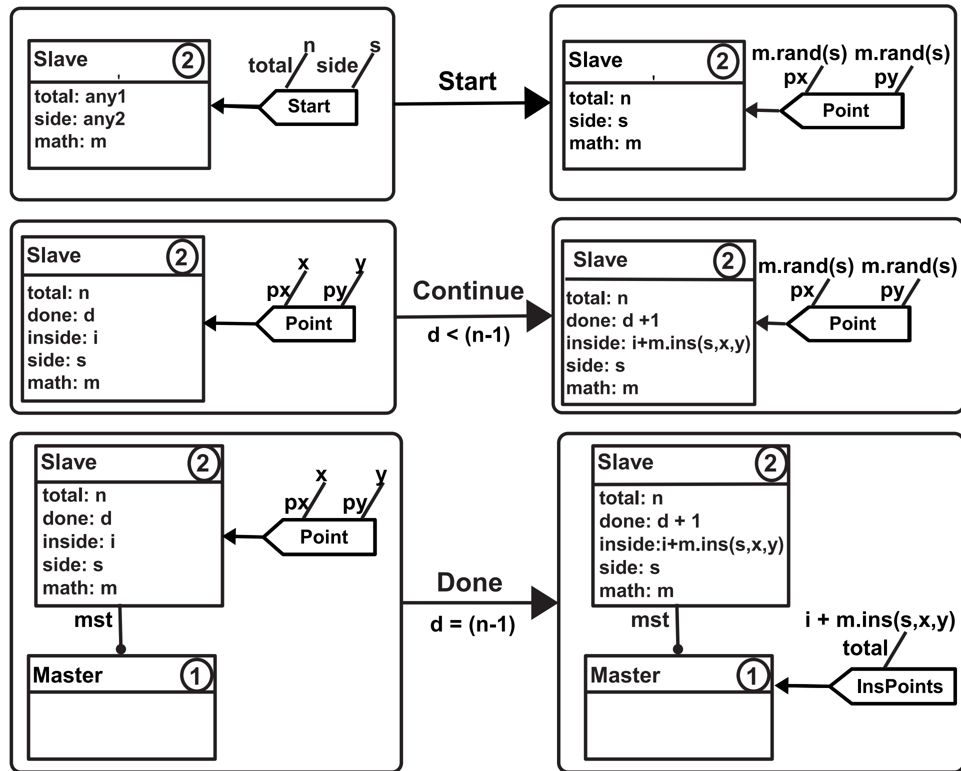


Figura 10: Regras para o objeto *Slave*.

Pela regra $Slave_Start$, na presença de uma mensagem *Start*, *Slave* fixa o número de pares a serem aleatorizados no atributo *total* e envia para si mesmo o primeiro par como parâmetro de uma mensagem *Point*. A partir desse evento, *Slave* repetidamente consome *Points*, computa se o ponto pertence ou não à circunferência (atributo *inside*) e atualiza o total de mensagens geradas (atributo *done*). Quando processada a última mensagem necessária para compor o total informado por *Master*, executa $Slave_Done$ enviando o total de pontos inscritos encontrado.

As funções $new_Math()$, $ins(s, x, y)$ e $rand(s)$ são definida externamente ao modelo GGBO, utilizando–se o seguinte Tipo Abstrato de Dado (TAD):.

Tipo Abstrato de Dado *Math*:

Operations:

Math new_Math ();

Inicializa o TAD.

int ins (**float** s, **float** x, **float** y);

A função $ins(s, x, y)$ determina se o ponto (x, y) pertence ou não a uma circunferência de raio $s/2$ inscrita em um quadrado de lado s, sendo que o quadrado e a circunferência têm o mesmo ponto como centro. Retorna 1 (um) caso o ponto esteja contido dentro do círculo, ou 0 (zero) caso contrário.

float rand (**float** limit);

Retorna um número menor ou igual a *limit*.

int rand_eval (**float** s, **int** n);

Aleatoriza um conjunto de n pontos dentro de um quadrado de lado s, retornando o total de pontos inscritos em um círculo de raio $s/2$ inscrito dentro do quadrado.

End ***Math.***

A função $ins()$ é responsável por determinar se o ponto pertence ou não à circunferência adotada. A implementação correspondente ao TAD apresentado deve respeitar as restrições previamente mencionadas. A função $rand_eval()$ será usada na Seção 4.2.•

GGBO fornece uma linguagem baseada em objetos para a especificação de sistemas concorrentes. Essas especificações apresentam algumas características que facilitam o desenvolvimento do sistema que descrevem. Um sistema baseado em objetos é modular, uma vez que é composto por entidades autônomas (objetos) conectados via interfaces bem definidas (mensagens). Os atributos e as operações que manipulam os objetos são descritos junto ao objeto e não podem se acessados por outros objetos (encapsulamento).

Um modelo baseado em objetos facilita o reuso das especificações devido ao encapsulamento e às interfaces de importação (operações que utiliza) e de exportação (operações que disponibiliza) de cada objeto. Assim, objetos especificados em um modelo podem se utilizados em outros modelos, sem a necessidade de conhecer sua estrutura interna. Ainda, um sistema descrito

por um modelo baseado em objetos apresenta uma arquitetura simples (devido à abstração) e descentralizada (devido à sua estrutura de objetos), que são dois princípios necessários para a extensibilidade, onde novas funcionalidades podem ser incluídas no sistema com pouco esforço, sem que o resto do sistema tenha de ser alterado. Dentro de um modelo GGBO, a extensão de funcionalidades de um objeto ou do sistema é alcançada através da inclusão de novos tipos de mensagens e objetos e da inclusão de novas regras.

3.1.1 Comparando GGBO com outras linguagens paralelas

Diferente das ferramentas gráficas apresentadas na Seção 2.4.4 que utilizam grafos para modelar a topologia da aplicação, GGBO utiliza grafos para representar tanto o estado do sistema quanto as transformações permitidas. A comunicação entre os nodos das ferramentas gráficas citadas anteriormente é feita através de canais de comunicação, simbolizados explicitamente através de arestas ligando os nodos. Existem operadores gráficos que permitem a criação de subgrafos, possibilitando certa dinamicidade na topologia. Porém, não existem operadores permitindo a criação e deleção de canais de comunicação entre nodos já existentes durante a execução do programa. GGBO, por sua vez, representa a comunicação através de mensagens, permitindo um arranjo dinâmico dos mecanismos de comunicação através da manipulação das referências existentes tanto como atributos dos objetos quanto como parâmetros nas mensagens. Tal característica oferece bastante flexibilidade para alterações estruturais na topologia do sistema modelado.

Por apresentar mecanismos centrados na troca de mensagens entre os participantes, GGBO apresenta características que a deixam mais próxima de uma linguagem de controle, como é o caso de Linda. Existem semelhanças no não–determinismo possível na escolha da tupla da ser consumida do *pool* de mensagens (para Linda) com o não–determinismo na escolha da mensagem a ser consumida do grafo de estado (para GGBO). Outra importante semelhança entre essas duas linguagens fica na utilização de mecanismos de *matching* para acionar o recebimento.

GGBO apresenta semelhanças com linguagens celulares, principalmente na utilização de regras para alterar o grafo de estado do sistema (que para linguagens celulares é chamado reticulado). Porém, GGBO não apresenta a possibilidade de considerar atributos da vizinhança de um nodo na aplicação de uma regra. Por outro lado, linguagens celulares têm restrições quanto à comunicação entre os nodos (somente entre nodos vizinhos), o que não ocorre em GGBO, onde é possível enviar uma mensagem para qualquer outro objeto ao qual se tenha referência.

3.2 Análise de Sistemas Modelados em GGBO

3.2.1 Simulação

Um modelo de simulação é uma representação simplificada de um processo ou sistema que permite analisá–lo. A principal vantagem é a possibilidade de validar uma estratégia de projeto, bem como os algoritmos a serem utilizados antes de sua implementação. Além disso, a existência de um modelo formal descrevendo o sistema torna as interdependências entre seus componentes explícitas e claras [18]. Além do uso da simulação como uma ferramenta para teste de especificações, esta também ser usada para avaliar o desempenho da aplicação modelada.

O simulador desenvolvido no ambiente do projeto PLATUS [18] permite a realização de simulações sobre modelos descritos utilizando GGBO, a partir do mapeamento desse modelos para código fonte JAVA.

Tabela 2: Características das linguagens GGBO e Java.

Linguagem	GGBO	Java
Unidade básica de computação	Objeto	Processo *Thread*
Criação de unidades básicas	Dinâmica e Estática	Dinâmica e Estática
Inicialização de modelos	Grafo inicial	Programa Principal
Forma de comunicação	Troca de mensagens	Troca de mensagens
Não–determinismo	Escolha de regras Consumo de mensagens	Estruturas de condição Randomização
Equivalências atômicas	Regras	Objetos
Concorrência	Dentro de objetos Entre objetos	Entre processos Entre *threads*

Dentro dos projetos citados, a linguagem GGBO foi estendida, de maneira a possibilitar definir o intervalo de tempo necessário para que uma mensagem seja tratada. No contexto de mobilidade, atribuir tempos para a entrega local e remota de mensagens assim como para a migração componentes de *software* permite que o desenvolvedor represente a latência das conexões da rede, o intervalo de tempo necessário para transferir um componente móvel ou

outras operações tempo–dependentes, tornando possível comparar o desempenho de diferentes abordagens e estratégias de distribuição e mobilidade em cenários complexos.

Essa tradução utiliza um gerenciador de comunicações para repassar todas as mensagens entre os objetos, e dar uma coerência temporal a estes eventos. Os objetos são descritos como classes em Java. Cada objeto tem e controla um *buffer* de mensagens recebidas e cria *threads* para executar as regras que manipulam mensagens recebidas. As regras são descritas também por classes Java, onde é definido qual mensagem tratar, assim como as condições que têm que ser satisfeitas para ativar a execução das regras.

3.2.2 Verificação de modelos descritos em GGBO

Para que possa ser realizada a verificação de um modelo descrito por uma GGBO, todos os elementos desse modelo (objetos, mensagens, regras e parâmetros) são traduzidos para estruturas na linguagem PROMELA, permitindo a utilização do verificador SPIN [39].

O processo de tradução consiste em mapear objetos GGBO para processos PROMELA (Tabela 3). Além da transformação de objetos para processos, outros elementos do modelo de gramática de grafos devem ser representados por um elemento equivalente na sintaxe da linguagem PROMELA. Mais detalhes sobre o processo de tradução de GGBO para PROMELA são encontrados em [24, 25], incluindo uma discussão sobre a compatibilidade semântica entre o modelo GGBO original e o modelo traduzido. Ainda, em [25, 63] é apresentado como especificar propriedades sobre modelos GGBO usando LTL, levando em consideração a definição de eventos conforme descrita em [59].

O processo de tradução – tanto para simulação quanto para a verificação – é feito atualmente de forma automática por uma ferramenta de edição de modelos desenvolvida dentro do projeto CASCO [23]. A ferramenta é capaz de interpretar os contra–exemplos gerados pelo verificador, compondo um diagrama de troca de mensagens entre os objetos GGBO presentes no sistema modelado. Essa funcionalidade mantém o mesmo nível de abstração para o desenvolvedor, eliminando a necessidade de analisar o contra–exemplo gerado pelo verificador, que é apresentado originalmente segundo os elementos presentes no modelo PROMELA gerado pela tradução.

Tabela 3: Características das linguagens GGBO e PROMELA.

Linguagem	GGBO	PROMELA
Unidade básica de computação	Objeto	Processo
Criação de unidades básicas	Dinâmica Estática	Dinâmica Estática
Inicialização de modelos	Grafo inicial	Processo inicial
Forma de comunicação	Troca de mensagens	Troca de mensagens
Não–determinismo	Escolha de regras Consumo de mensagens	Estruturas de condição Estruturas de repetição
Equivalências atômicas	Regras	Estruturas tipo *atomic*
Concorrência	Dentro de objetos Entre objetos	Entre processos

3.3 Verificando o modelo Pi

Nessa seção serão discutidas as propriedades a serem provadas para o modelo apresentado na Seção 3.1. Para manter tratável o espaço de estados gerado, o modelo será verificado usando–se três *Slaves* e um *Master*, e cada *Slave* aleatorizará somente quatro pontos. A seguintes propriedades serão formalizadas em LTL e verificadas para o modelo:

(i) Terminação: O modelo gera um resultado, e após isso nenhuma outra regra é aplicada;

(ii) Resposta: Um *Slave* sempre responde a uma requisição;

(iii) Convergência: Existe (pelo menos um) cenário levando a um resultado aceitável.

A seguir, será utilizada a sintaxe do verificador SPIN para expressar as propriedades. Serão utilizados padrões de eventos conforme definidos em [59], sendo indicado após a propriedade qual foi o padrão de sentença utilizado para a fórmula LTL.

- **(i) Terminação:** *Master_Done* é a regra a qual calcula o valor final para π. Assim, deve–se provar que essa regra é aplicada, e depois de sua aplicação nenhuma outra regra

é aplicada. Para provar a aplicação da regra o modelo pode ser verificado para a seguinte sentença:

$<> (\uparrow Master_Done)$

Que resulta em verdadeiro. Como segundo passo, para provar que depois dessa regra nenhuma outra é aplicada, a sentença a seguir foi aplicada ao modelo:

$[](\uparrow Master_Done \rightarrow []! \uparrow Any_Rule)$

Onde: $\uparrow Any_Rule = (\uparrow Master_Start || \uparrow Master_Receive || \uparrow Master_Done || \uparrow Slave_Start || \uparrow Slave_Continue || \uparrow Slave_Done)$.

Essa fórmula é o padrão de *Ausência Depois de um Evento*, como apresentado em [59]. No caso, ausência de Any_Event depois de $Master_Done$. Essa sentença também representa uma propriedade verdadeira para o modelo.

- **(ii) Resposta:** A cada regra $Slave_Start$ aplicada existe sempre uma regra $Slave_Done$ associada.

  $<>\uparrow Slave_Start \rightarrow <> (\uparrow Slave_Start \&\& <>\uparrow Slave_Done)$

  Essa fórmula corresponde ao padrão *Existência (de Slave_Done) Após um Evento (Slave_Start)* [59], e resulta em verdadeiro.

  Novamente, é importante lembrar que esse passo assume como corretas as funções $ins()$ e $rand()$. Essas funções foram manualmente traduzidas para o modelo PROMELA gerado. No modelo, $rand()$ sempre retorna o valor 0 (zero), e $ins()$ retorna 0 (zero) ou 1 (um) não–deterministicamente. Essa abstração foi utilizada para diminuir o espaço de estados, e não modifica o significado do sistema modelado, uma vez que é o resultado de $ins()$ que define se o ponto pertence ou não à circunferência.

- **(iii) Convergência:** A convergência do modelo pode ser mostrada para o resultado dentro de um intervalo. Porém, uma vez que a verificação do modelo considera todas as combinações possíveis de pontos gerados (neste caso, todas as possíveis combinações de resultados para $ins()$), serão também incluídas na verificação computações onde todos os pontos estarão fora do círculo.

  É possível provar através de *model-checking* que existem computações que convergem. Usando LTL, para provar que a convergência é possível, será elaborada uma sentença afirmando que o modelo não irá convergir. O contra–exemplo para essa sentença será

um *trace* mostrando uma seqüência de eventos para a qual o modelo converge. Porém, é importante lembrar que o fato do resultado convergir ou não para esse modelo está associado à qualidade do gerador aleatório utilizado na implementação real. Assim, a sentença utilizada foi:

$[]!\ (\uparrow Master_Done_Pi_Appox)$

Onde: $\uparrow Master_Done_Pi_Appox = \uparrow Master_Done$ && $(3,0 \leq result \leq 3,4)$

Esta fórmula corresponda ao padrão de *Ausência Global de um Evento* [59] e, como esperado, resultou falso. A Figura 11 mostra um *trace* com a troca de mensagens entre processos relativa ao contra–exemplo obtido a partir dessa sentença.

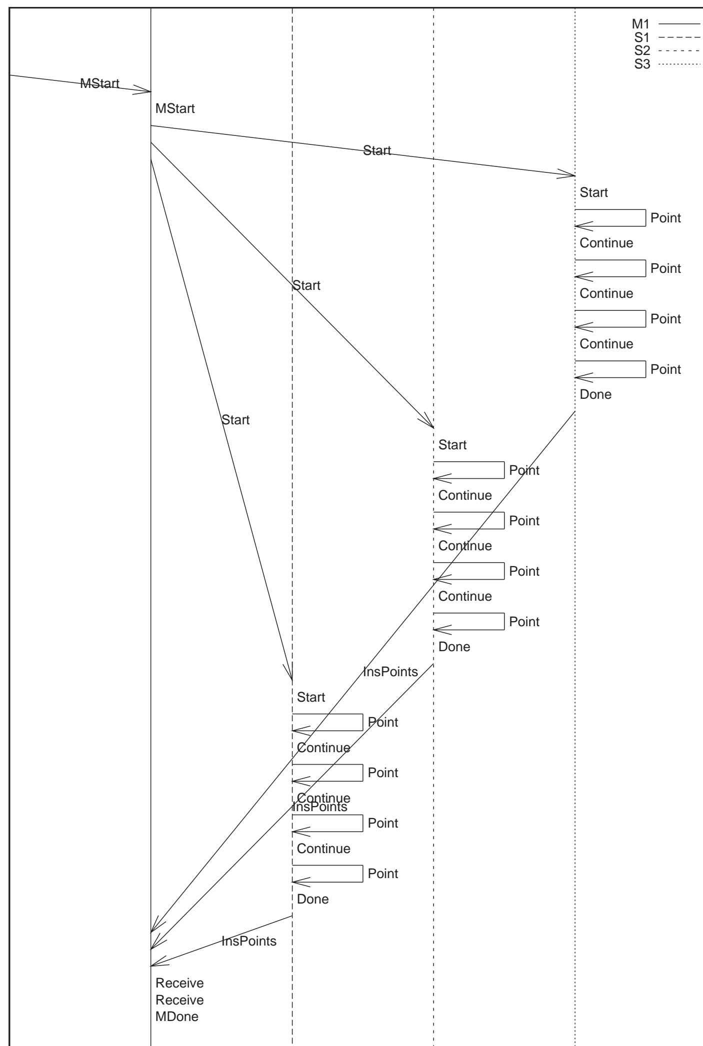


Figura 11: Contra–exemplo associado à terceira sentença verificada.

Este *trace* foi gerado automaticamente pela ferramenta utilizada na modelagem. As linhas verticais representam o fluxo de execução de cada processo envolvido. O nome da regra aplicada aparece junto ao fluxo de execução, e pode estar associado a qualquer uma das mensagem que estejam disponíveis para aquele objeto quando da aplicação. As setas representam o envio de mensagens, e junto a cada uma delas está o nome da mensagem sendo enviada. Observe que no contra–exemplo existe uma seqüencialização no consumo das mensagens.

Uma limitação na ferramenta de verificação é a falta de suporte a variáveis de ponto flutuante. Para a verificação da sentença, o modelo foi modificado de maneira que $\pi \approx 1000 \times (4 \times k \div n)$ fosse a operação usada na regra$Master_Done$, e testando a convergência do resultado dentro do intervalo $(3000, 3400)$. Utilizando–se um número maior de pontos aleatorizados em cada *Slave* poderia–se diminuir o intervalo.

4 Tradução

Nas seções anteriores foram apresentados a fundamentação teórica e o formalismo de modelagem usado para o projeto de aplicações paralelas. Neste Capítulo serão propostas e discutidas estruturas de programação genéricas que possibilitem a conversão de um modelo GGBO em código fonte utilizando MPI como biblioteca de comunicação. A partir dos mecanismos propostos, será gerado o programa correspondente ao modelo introduzido na Seção 3.1, para o qual será analisado o desempenho. A partir dessa análise, o modelo é modificado e reavaliado. Ainda, é apresentado um novo modelo visando ilustrar o comportamento do algoritmo proposto para os objetos GGBO em testes de envio de mensagens.

4.1 Tradução GGBO – MPI

A tabela 4 apresenta um comparativo entre as características oferecidas pelo formalismo GBBO e seus correspondentes encontrados em um ambiente C/MPI.

Além do fato de existirem elementos análogos entre GGBO e MPI que por si só facilitam a tradução, a utilização de uma linguagem formal como entrada para um gerador de código facilita ainda mais a tarefa, por possibilitar a correta e inambígua identificação das estruturas de programação adequadas e geração do código correspondente ao modelo.

4.1.1 Processo GGBO – Algoritmo Básico

Basicamente, modificações são feitas no grafo de um sistema completo identificando–se *matches* entre mensagens presentes no grafo e regras do objeto, executando regras que consomem mensagens e alteram seus atributos internos e gerando novas mensagens que são inseridas dentro do grafo do sistema. O grafo de estado pode ser visualizado como uma estrutura dinâmica, onde

Tabela 4: Características da linguagem GGBO e padrão MPI.

Linguagem	GGBO	C/MPI
Unidade básica de computação	Objeto	Processo
Criação de unidades básicas	Dinâmica Estática	Estática Dinâmica[a]
Inicialização de objetos	Grafo inicial	Chamada MPI
Forma de comunicação	Troca de mensagens	Troca de mensagens
Não– determinismo	Escolha de regras, Consumo de mensagens	Funções Pseudo–Aleatórias
Concorrência	Dentro de objetos Entre objetos	Entre Processos

[a] o padrão MPI–2 inclui a possibilidade de criação dinâmica de processos MPI.

ocorrem paralelamente várias modificações.

Um primeiro passo natural no processo de tradução é mapear cada objeto em um processo separado (processo GGBO), possibilitando a execução paralela de regras por processos diferentes do sistema (paralelismo entre objetos – concorrência externa). Poderia ser oferecido também paralelismo na execução de regras dentro de um mesmo objeto GGBO (concorrência interna), porém, isso envolveria o desenvolvimento de um mecanismo de identificação de conflitos entre regras, o que não foi feito por inserir um grau de complexidade muito alto para uma implementação inicial.

Outro aspecto a ser considerado diz respeito ao número de mensagens suportadas pelo modelo. O grafo de um sistema GGBO comporta, em tese, uma quantidade ilimitada de mensagens que podem ser consumidas a qualquer momento pelo objeto ao qual essa mensagem se destine. Essa característica pode ser aproximada utilizando–se uma lista encadeada (L_{in}) para armazenar as mensagens já recebidas e ainda não processadas em cada processo GGBO. Assim, pode–se suportar um volume relativamente grande de mensagens, dentro dos limites que a alocação dinâmica de memória permitir.

Definido o tratamento a ser dado para as mensagens do sistema, pode–se determinar o processo para seleção e execução das regras existentes. Uma vez que L_{in} abriga todas as mensagens endereçadas ao objeto, sobre essa lista podem ser identificados todos os *matches* mensagem–regra possíveis. Isso é feito varrendo–se toda a lista, cruzando cada mensagem (M_i) com as regras (R_e) existentes para aquele objeto GGBO e verificando se a condição de ativação da regra (caso exista) é satisfeita. Como uma mesma mensagem pode ativar mais de uma regra em um determinado momento, a varredura deve cobrir todas as regras suscetíveis àquela mensagem. Cada par (M_i, R_e) gerado pela varredura é armazenado em uma lista ($L_{matches}$), da qual é selecionado aleatoriamente um elemento. A mensagem indicada é retirada da lista de entrada e consumida disparando a regra associada.

A execução de uma regra pode modificar os valores dos atributos presentes dentro do processo GGBO e/ou gerar uma ou mais mensagens. De maneira análoga com o que é feito com as mensagens recebidas, pode–se inserir as mensagens geradas em uma lista L_{out} durante a aplicação da regra, deixando o envio dessas mensagens para um passo posterior. Essa escolha foi feita por livrar a aplicação de regras do ônus relativo ao envio de dados, permitindo assim passar imediatamente ao processamento da próxima mensagem em $L_{in.}$ A adoção de L_{out} possibilita também outras vantagens que serão discutidas nas próximas seções.

A partir das considerações tomadas, o algoritmo básico executando dentro de um processo GGBO pode ser organizado em oito passos principais:

1. Recebe uma mensagem M;
2. Inclui M dentro da lista de mensagens L_{in};
3. Avalia L_{in} construindo lista $L_{matches}$;
4. Seleciona aleatoriamente um par (M_i, R_e) de $L_{matches}$;
5. Retira M_i de L_{in};
6. Aplica a regra R_e (consumindo M_i e gerando n mensagens);
7. Insere cada mensagem gerada em L_{out};
8. Envia cada mensagem em L_{out} segundo ordenação FIFO.

Esse foi o conjunto de considerações utilizados como base para a geração do código dos processos GGBO. A seguir, são discutidos com maiores detalhes elementos adotados na definição

de um esquema de tradução que possibilitam a transformação desse algoritmo básico em código fonte utilizável.

4.1.1.1 Código do Processo GGBO

Dentro do formalismo GGBO, toda operação de processamento de mensagem – *send* ou *receive* – é não–bloqueante. Isso significa que nenhum processo enviando/recebendo mensagens deve ficar parado esperando a finalização da operação de comunicação.

MPI oferece tanto primitivas de comunicação bloqueantes ($MPI_Send()$, $MPI_Recv()$) quanto não bloqueantes ($MPI_Isend()$, $MPI_IRecv()$). A semântica associada às primitivas não–bloqueantes permite que elas sejam usadas apenas para sinalizar o início da operação de envio/recebimento em um *buffer*, possibilitando que o processo continue e execute as próximas operações enquanto os dados são transferidos. A transferência de dados começa apenas quando um segundo processo iniciar uma operação de comunicação complementar com o primeiro. Primitivas não–sincronizantes não sinalizam explicitamente o término da operação de transferência, tornando necessárias chamadas a funções que verificam o término antes de efetivamente indicar a finalização do *send/receive*.

Considerando a complexidade em se implementar um sistema com tal característica, optou–se por utilizar primitivas bloqueantes para envio e recebimento, porém adotando *threads* distintas para recebimento, processamento e envio de mensagens[1]. Essas *threads* serão referenciadas respectivamente como *Sender*, *Receiver* e *Evaluate*. Do algoritmo básico apresentado anteriormente, *Receiver* fica associado à execução os passos 1 e 2, *Evaluate* comporta os passos 3 a 7 e *Sender* executa o passo 8. Dessa forma, quebra–se a sincronicidade e o processo GGBO (agora composto por três *threads*) estará apto a tratar as mensagens recebidas mesmo quando ainda existirem mensagens a serem (ou sendo) enviadas. Ainda, tal estratégia possibilita a implementação de um programa mais modular, facilitando o entendimento, legibilidade e manutenibilidade

[1]Devido ao número de ações de comunicação que podem ocorrer em um programa paralelo típico, processos são interrompidos mais freqüentemente do que em um ambiente seqüencial. Assim, a manipulação de processos em um ambiente multiprogramado fica penalizada [72]. Para amortizar esse efeito, é comum o uso de *threads* de processamento dentro de um mesmo programa paralelo. Diferente de processos completos, que possuem regiões de memória bem delimitadas e protegidas pelo sistema operacional, todas as *threads* de um mesmo processo podem compartilhar uma mesma regiões de memória. Como resultado existe muito menos contexto para ser salvo quando uma *thread* deixa o processador em favor de outra, com um custo mais baixo do que o envolvido no chaveamento entre processos. Ainda, como o código relativo a todas as *threads* vêm de um mesmo código fonte, o compilador pode gerar de forma eficiente o programa que controla a interação entre os elementos.

do código gerado.

Todavia, a utilização de *threads* junto com MPI pode não ser trivial, pois determinadas implementações do padrão de comunicação não são *thread–safe* [77]. Uma trecho de código qualquer é dito *thread-safe* quando funciona corretamente durante a execução simultâneas por múltiplas *threads*. Um código pode ser tornado *thread–safe*, por exemplo, protegendo os dados sensíveis através de exclusão mútua e manipulando–os através de operações atômicas. Para contornar esse problema, foi feito o uso de um semáforo (MPI_mutex) para garantir a exclusão mútua em determinadas chamadas à biblioteca MPI – aquelas envolvendo o *deamon* de comunicação.

Essa escolha implica em cuidados adicionais enviando e recebendo mensagens utilizando primitivas bloqueantes. Como a chamada Recv é bloqueante, se a *thread Receiver* bloquear MPI_mutex e após isso ficar bloqueada em um Recv, a *thread Sender* não poderá fazer nenhum envio, já que não poderá entrar na sua seção crítica antes que *Receiver* efetivamente finalize o Recv e libere o mutex. A ocorrência simultânea desse evento em múltiplos processos GGBO pode colocar o sistema em uma situação de *deadlock*. Assim, antes que *Receiver* bloqueie o semáforo, essa *thread* verifica se realmente existe uma mensagem a ser recebida através da chamada MPI_Iprobe, a qual retorna *false* – caso não exista nenhuma mensagem sendo enviada por outro processo (ou *thread* de processo) – ou *true* (e outras informações de controle) caso contrário. Sendo necessário executar um Recv, o semáforo é fechado e o processo realiza a chamada, ao final da qual o semáforo é liberado. Caso não seja necessário receber uma mensagem, a *thread* executa uma chamada $sched_yeld()$ e libera[2] o processador.

As Figuras 12 e 13 apresentam o pseudo–código do laço principal executado pelas *threads* *Receiver* e *Sender*, respectivamente. *Evaluate* será apresentada posteriormente, na Seção 4.1.3. Observe a utilização de semáforos nas linhas 16 da Figura 12 e 2 da Figura 13, visando sincronizar o funcionamento entre essas duas *threads* e *Evaluate*. Entre *Receiver* e *Evaluate* é utilizado um semáforo binário, uma vez que *Evaluate* bloqueia somente quando não existirem mais *matches*. Assim, sempre que *Receiver* obtiver uma mensagem, ele seta o semáforo binário em 1, liberando *Evaluate* caso esse esteja bloqueado esperando uma mensagem que talvez possibilite um *match*. Caso a mensagem não possibilite nenhum, *Evaluate* executa um *down* no semáforo e pára novamente[3].

[2]Disponível a partir da inclusão da biblioteca "sched.h". Essa chamada força o processo ou *thread* que a invocou a abandonar o processador em favor de outro processo ou *thread* que esteja pronto para execução. Dessa forma, é amortizado o efeito de espera ocupada enquanto não existam mensagens a serem recebidas.

[3]Na realidade, *Evaluate* deve executar um segundo *down* para efetivamente bloquear. Assim, existe um

```
1   do{
2         pthread_mutex_lock (&mutex_MPI);
3         MPI_Iprobe( MPI_ANY_SOURCE, ... ,  &flag, ...);
4
5         if (flag == true)
6         {
7                    MPI_Recv( buff, TAM, MPI_PACKED, ...);
8                    pthread_mutex_unlock (&mutex_MPI);
9
10                   position = 0;
11                   MPI_Unpack(buff, TAM, &position, &buffer_mesg.type, ...);
12                   /*demais chamadas a MPI_Unpack */
13
14                   pthread_mutex_lock  (&mutex_Incoming);
15                   inseriu = Insere_Dir (&Incoming, buffer_mesg.type, &buffer_mesg, 0);
16                   if (inseriu==true) pthread_mutex_unlock(&mutex_evaluate);
17                   pthread_mutex_unlock(&mutex_Incoming);
18        }
19        else
20        {
21                   pthread_mutex_unlock (&mutex_MPI);
22                   sched_yield();
23        }
24  } while (receiver_continue == true);
```

Figura 12: Pseudo–código para *Receiver*.

```
1   do{
2         sem_wait(&mutex_send);
3
4         pthread_mutex_lock (&mutex_Outcoming);
5         retirou = Retira_Esq (&Outcoming, &buffer_mesg, &dest);
6         if ( retirou == true)
7         {
8                   pthread_mutex_unlock (&mutex_Outcoming);
9
10                  if(buffer_mesg.type == control_mesg)
11                  {
12                            /*Ações relativas ao controle de execução*/
13                  }
14                  else
15                  if(dest!=my_id) /*mensagem para outros elementos*/
16                  {
17                            position=0;
18                            MPI_Pack(&buffer_mesg.type,   1, MPI_INT, buff, ...);
19                            /*demais chamadas a MPI_Unpack */
20
21                            pthread_mutex_lock (&mutex_MPI);
22                            MPI_Send( buff, position, MPI_PACKED, dest, ...);
23                            pthread_mutex_unlock (&mutex_MPI);
24                  }
25                  else
26                  if(dest==my_id) /*mensagem para si mesmo*/
27                  {
28                            pthread_mutex_lock  (&mutex_Incoming);
29                            Concatena_Lista(&Selected,&Incoming);
30                             pthread_mutex_unlock(&mutex_evaluate);
31                            pthread_mutex_unlock  (&mutex_Incoming);
32                  }
33        }
34        else
35        {
36                  pthread_mutex_unlock (&mutex_Outcoming);
37        }
38  } while (sender_continue == true);
```

Figura 13: Pseudo–código para *Sender*.

Entre *Evaluate* e *Sender* é utilizado um semáforo contador, incrementado/decrementado con-

deslizamento de um ciclo de execução do laço principal de *Evaluate* antes que esse efetivamente bloqueie.

forme o número de mensagens geradas e enviadas pelo processo GGBO. O semáforo $mutex_MPI$ usado é usado para controlar o acesso exclusivo ao MPI entre as linhas 2 a 8 da Figura 12 (ou 2 e 21, caso a condição da linha 5 seja falsa), e entre as linhas 21 a 23 da Figura 13. Também são utilizados semáforos para controlar o acesso às listas de armazenamento de mensagens, de modo a garantir a consistência dos dados nas linhas 14 a 17 de *Receiver* e 4 a 8 (ou 4 e 36) e 29 a 31 de *Sender*.

Apesar de problemas relativos à *thread–safety* estarem bastante relacionados à implementação da biblioteca MPI sendo utilizada, a adoção de estruturas de controle bem conhecidas como semáforos e variáveis condicionais possibilita a geração de um código capaz de coexistir com tais problemas, com um pequeno custo em performance.

4.1.2 Mapeamento Mensagens GGBO – Mensagens MPI

Mensagens GGBO são traduzidas em mensagens MPI utilizando–se estruturas de dados *C*. Essas estruturas, compostas por tipos de dados primitivos como *char*, *int* e *float*, podem ser enviadas desde que antes a mensagem MPI seja construída campo–a–campo, utilizando–se primitivas de empacotamento ($MPI_Pack()$) disponibilizada pela biblioteca MPI. Em contrapartida, tais mensagens devem ser desempacotadas em mesma ordem pelo código gerado para o receptor, fazendo uso de chamadas correspondentes($MPI_Unpack()$).

Dentro da estrutura de dados gerada pela tradução, o primeiro campo contém identificador do tipo da mensagem correspondente. Os demais campos (quando existentes) relacionam–se a cada parâmetro presente na mensagem. A Figura 14 apresenta a tradução correspondente em C para as mensagens GGBO $Start$, $Points$, $Point$ e $InsPoints$ previstas no modelo[4]. A Figura 15 apresenta as operações de empacotamento e desempacotamento relativas aos parâmetros sendo enviados nas mensagens[5].

Apesar de mensagens MPI possuírem um campo $FLAG$ que poderia ser utilizado para in-

[4]O tipo *mtype* utilizado na estrutura é uma enumeração contendo todas os nomes de mensagens e regras previstos no modelo, além de outras mensagens de controle (Vide Seção 4.1.5).

[5]O parâmetro *buffer* indica a área de memória (limitada a TAM bytes) onde a mensagem é preparada para envio. Em *buffer_mesg* está a estrutura sendo enviada. Vetores de tipos de dados primitivos podem ser enviados passando–se como parâmetro a posição do primeiro elemento seguida pelo número de elementos a serem empacotados. Observe que para a estrutura em questão, é informado apenas 1(um) elemento. O inteiro *position* é utilizado como um para indicar a posição em *buffer* sendo modificada. Elementos do tipo *mtype* são empacotados como inteiros.

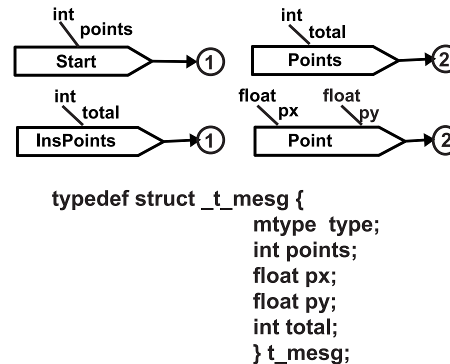


Figura 14: Tradução Mensagem GGBO – Código Fonte.

```
position=0;
 MPI_Pack(&buffer_mesg.type,   1, MPI_INT     , buffer, TAM, &position, MPI_COMM_WORLD);
 MPI_Pack(&buffer_mesg.points,1, MPI_INT      , buffer, TAM, &position, MPI_COMM_WORLD);
 MPI_Pack(&buffer_mesg.px,      1, MPI_FLOAT, buffer, TAM, &position, MPI_COMM_WORLD);
 MPI_Pack(&buffer_mesg.py,      1, MPI_FLOAT, buffer, TAM, &position, MPI_COMM_WORLD);
 MPI_Pack(&buffer_mesg.total,   1, MPI_INT     , buffer, TAM, &position, MPI_COMM_WORLD);
  /*Envio*/

  /*Recebimento*/
position = 0;
  MPI_Unpack(buffer, TAM, &position, &buffer_mesg.type,     1, MPI_INT     , MPI_COMM_WORLD);
  MPI_Unpack(buffer, TAM, &position, &buffer_mesg.points,  1, MPI_INT     , MPI_COMM_WORLD);
  MPI_Unpack(buffer, TAM, &position, &buffer_mesg.px,        1, MPI_FLOAT, MPI_COMM_WORLD);
  MPI_Unpack(buffer, TAM, &position, &buffer_mesg.py,        1, MPI_FLOAT, MPI_COMM_WORLD);
  MPI_Unpack(buffer, TAM, &position, &buffer_mesg.total,    1, MPI_INT     , MPI_COMM_WORLD);
```

Figura 15: Preparando mensagem – *Packing/Unpacking*.

dicar o tipo de mensagem sendo transmitida, optou–se por não fazer uso de tal característica, possibilitando assim uma dependência menor entre o código gerado e a biblioteca de comunicação utilizada.

Dentro desse primeiro esquema de tradução apresentado, o sistema gerado comporta apenas um tipo de estrutura de dados para mensagem, o qual é um superconjunto contendo todos os parâmetros de mensagem permitidos dentro do sistema. Devido a isso, existem mensagens as quais possuem mais parâmetros que o estritamente necessário, assim como operações de empacotamento e desempacotamento relativos a esses campos extras, o que degrada a performance do sistema gerado. Entre os melhoramentos a serem incluídos em versões futuras está a definição de operações de empacotamento/desempacotamento mais complexas, associadas ao tipo de mensagem a ser enviada/recebida em cada situação.

Outra restrição importante adotada é a limitação dos parâmetros das mensagens apenas às estruturas baseadas em tipos de dados suportados pelas primitivas de empacotamento MPI. Assim, durante a modelagem dos sistemas GGBO, não é permitido o envio de estruturas de dados

complexas como listas ou filas, a menos que o usuário disponibilize operações para transformar essas estruturas em tipos primitivos de dados.

4.1.3 Mapeamento Regras GGBO – Código C

Antes de apresentar a tradução das regras propriamente ditas, será retomado em maiores detalhes o algoritmo de *matching*, de modo a deixar clara a implementação correspondente aos passos 3, 4 e 5 apresentados na Seção 4.1.1.

A lista $L_{matches}$ é composta por pares (M_i, R_e), onde M_i associa–se a uma mensagem em L_{in} e R_e a uma regra do objeto. $L_{matches}$ é preenchida tomando–se cada mensagem presente em L_{in} e avaliando– se a condição de ativação de todas as regras, de modo a identificar todas as possibilidades de *match* para a mensagem naquele momento. Para regras que não possuem uma condição associada, a simples existência da mensagem já corresponde a um *match*.

```
1  int Do_Matching (Elo Lista_mesgs, Elo_par* Lista_pares)
2  {
3    Elo P;
4    int contador = 0;
5    int rule = (-1);
6
7    P = Lista_mesgs;
8    while (P != NULL)
9    {
10       /*rule_Master_Start */
11       if (P->type == msg_Master_Start)
12       {
13               rule = rule_Master_Start;
14               Insere_Esq_par (Lista_pares, P, rule);
15               contador++;
16       }
17       /*rule_Master_Receive */
18       if ((P->type == msg_Master_InsPoints)  && (atr_received < (atr_slaves - 1))
19       {
20              rule = rule_Master_Receive;
21              Insere_Esq_par (Lista_pares, P, rule);
22              contador++;
23       }
24       ...
25       /*avançar */
26       P = P->Dir;
27   }
28   return contador; /*retorna qtos matchings */
29 }
```

Figura 16: Função de *matching*.

A Figura 16 apresenta um trecho correspondente ao algoritmo de *matching* proposto. O tipo *Elo* declarado na função referencia elos da lista encadeada usada para o armazenamento das mensagens. Por exemplo, se a regra *Master_Receive* envolvesse o parâmetro *points* presente na mensagem e um atributo *atr* qualquer, a comparação $(P- > points == atr)$ seria incluída na linha 18.

A Figura 17, por sua vez, apresenta o pseudo–código do laço principal sendo executado por *Evaluate*. A função de *matching* é invocada na linha 6. Na figura encontra–se explicitado em detalhes a tradução da regra $Master_Start$ prevista no modelo Pi.

```
1   do{
2           pthread_mutex_lock  (&mutex_Incoming);
3           Concatena_Lista(&Selected,&Incoming);
4           pthread_mutex_unlock  (&mutex_Incoming);
5
6           total_matches = Do_Matching(Selected,&Matches);
7
8           if (total_matches > 0)
9           {
10                  selected_match= rand()%total_matches;
11                  Elo_temp = Consulta_Pos_par (Matches, selected_match, &selected_rule);
12                  retirou = Retira_No (&Selected, Elo_temp, &buffer, &dummy);
13                  Libera_Memoria_par(&Matches); /*destroi a listade matches*/
14
15                  if(retirou == true )
16                  {
17                  /*rule_Master_Start*/
18                  if(selected_rule == rule_Master_Start)
19                  {
20                  /*copia valores das regras*/
21                  n=buffer.points;
22                  /*Aplica regra*/
23                  atr_totalgroup = 4*n;
24                  /*prepara mensagem e insere na lista de saida o resultado*/
25                  buffer.type    = msg_Slave_Start;
26                  buffer.points  = 0;
27                  buffer.px      = 0;
28                  buffer.py      = 0;
29                  buffer.total   = n;
30
31                  Insere_Dir (&Generated, buffer.type , &buffer, sl1);
32                  n_mesgs_created++;
33                  Insere_Dir (&Generated, buffer.type , &buffer, sl2);
34                  n_mesgs_created++;
35                  Insere_Dir (&Generated, buffer.type , &buffer, sl3);
36                  n_mesgs_created++;
37
38                  pthread_mutex_lock  (&mutex_Outcoming);
39                  Concatena_Lista(&Outcoming,&Generated);
40                  for(loop_count=0;loop_count<n_mesgs_created;loop_count++)sem_post(&mutex_send);
41                   event_RuleName = rule_Master_Start;
42                  pthread_mutex_unlock  (&mutex_Outcoming);
43                  }
44                  else
45                  /*rule_Master_Receive*/
46                  if(selected_rule == rule_Master_Receive)
47                  {
48                  /*copia valores das regras*/
49                  /*Aplica regra*/
50                  /*prepara mensagem e insere na lista de saida o resultado*/
51                  }
52                  else
53                  /*demais regras*/
54          }
55          else
56          {
57               /*signal_blocked_state(); */
58               pthread_mutex_lock(&mutex_evaluate);
59          }
60  }while(evaluate_continue==true);
```

Figura 17: Pseudo–código para *Evaluate*.

Sabendo–se o total de mensagens presentes em $L_{matches}$ (linhas 15 e 21 da Figura 16), pode–se aleatoriamente gerar o índice do par mensagem–regra a ser utilizado (linha 10 da Figura 17).

A mensagem selecionada pode estar em qualquer posição de L_{in}. Fazendo–se que o primeiro elemento do par (M_i, R_e) seja um apontador para o endereço em que se encontra M_i em L_{in}, pode–se alcançar diretamente a mensagem, sem a necessidade de percorrer L_{in} novamente após a construção de $L_{matches}$. Além disso, decidiu–se pela utilização de uma lista duplamente encadeada para implementar L_{in}, por facilitar a remoção de elementos em qualquer ponto da lista uma vez conhecida sua posição.

Uma vez identificada a regra a ser executada, o processo GGBO realiza os seguintes passos:

(i) consome a mensagem (retira de L_{in}, passando–a para um *buffer* dentro do código da regra (linha 12 da Figura 17);

(ii) atualiza os atributos internos ao processo GGBO, se necessário fazendo uso de funções definidas externamente (linhas 20 a 22 da Figura 17);

(iii) gera as mensagens, inserindo–as em uma lista temporário L_{temp} interna à *thread* de processamento (linhas 23 a 35);

(iv) anexa L_{temp} ao final de L_{out} (linha 38).

Como a guarda relativa à regra selecionada já é avaliada para a geração de $L_{matches}$ e nenhum atributo foi modificado desde então (uma vez que apenas a *thread Evaluate* é pode alterar os atributos do processo), não é necessário reavaliar essa condição antes de efetivamente disparar a regra.

Caso a inserção de cada mensagem gerada fosse efetuada diretamente na lista de saída, seriam necessários vários *locks* consecutivos sobre L_{out}, impossibilitando o acesso à lista por parte *Sender* em múltiplas situações. Fazendo uso de uma lista temporária, só é necessário bloquear o acesso L_{out} para realizar uma operação, tornando a lista mais disponível à *thread* de envio. Mensagens endereçadas ao próprio processo (como ocorre nas regras *Slave_Continue*) são inseridas diretamente em L_{in}. De maneira análoga, *Receiver* também inclui as mensagens recebidas em uma lista temporária (chamada *Incoming* no pseudo–código mostrado). Após a aplicação de uma regra, *Evaluate* anexa *Incoming* à lista de mensagens *Selected*, sobre a qual é realizada a operação de *matching*.

4.1.4 Inicialização do Sistema

O grafo inicial é responsável pela definição dos valores de todos aos atributos dos objetos dentro do sistema, assim como a amarração das referências necessárias. Outro função realizada

pelo grafo inicial é a de definir no grafo de estado as mensagens que irão iniciar o funcionamento dos objetos.

Essas duas funções são realizadas em separado no código gerado. A definição dos valores relativos aos atributos dos objetos é traduzida como um conjunto de atribuições, que ficam inseridas no código de cada objeto GGBO imediatamente antes do ponto em que as *threads Evaluate*, *Receiver* e *Sender* são disparadas. Dessa forma esses valores são fixados antes do recebimento de qualquer mensagem, impossibilitando que o mecanismo de *matching* encontre o objeto em um estado ainda inconsistente.

A geração e o envio de todas as mensagens é realizada por um processo especial chamado *INIT*. A ordem de envio a ser seguida é definida aleatoriamente. Após a última mensagem necessária ter sido enviada, *INIT* passa a operar como um coordenador entre os demais processos, conforme será exposto na Seção seguinte. Tal configuração para *INIT* implica na necessidade de um processo a mais quando da chamada de inicialização do sistema. O esquema de tradução atualmente utilizado não oferece suporte à criação dinâmica de objetos GGBO.

4.1.5 Terminação

Para este trabalho, torna–se interessante adotar um protocolo de detecção de terminação que apresente uma baixa taxa de atraso para a identificação da parada. Nesse sentido, destaca–se o recente trabalho apresentado em [52]. Porém, o algoritmo proposto, apesar de apresentar resultados interessantes tanto em termos de número de mensagens quanto em atraso de detecção, baseia–se no uso de *acknowledgements*, o que torna sua utilização pouco adequada para este trabalho devido às modificações necessárias no mecanismo de comunicação.

Assim, optou–se por utilizar a solução proposta em [69] chamada *Counting*, por apresentar maior afinidade com os objetivos do trabalho, bem como por adequar–se a características de topologia do sistema gerado, conforme será discutido a seguir. Esse algoritmo baseia–se em contagem de mensagens, sinalizando a terminação quando todos os processos estiverem bloqueados e o total geral de mensagens enviadas pelos processos for igual ao total de mensagens recebidas. Ele adapta–se à carga do sistema, gerando menos mensagens de controle quando o sistema está ocupado e mais mensagens quando ele estiver com pouco carga.

O algoritmo *Counting* é uma algoritmo distribuído em duas fases alternantes, utilizado em sistemas gerados a partir da linguagem de programação paralela *Charm* [45, 67]. Toda a comunicação de controle entre os componentes ocorrem seguindo a topologia formada inicialização

dos processos, formando uma *spanninng tree*, suportando a geração dinâmica de novos processos, desde que eles sejam corretamente associados a um pai dentro da árvore de ativação.

O nodo raiz da árvore assume a função de coordenador, ficando responsável por identificar a parada e determinar a finalização do programa primário. Para o modelo gerado, dentro do algoritmo de detecção os processos GGBO organizam–se hierarquicamente segundo uma topologia em estrela, com *INIT* como coordenador. Apesar da geração dinâmica de processos não ser suportada dentro do esquema proposto, o protocolo escolhido comporta tal característica, não sendo necessária sua substituição caso a geração dinâmica venha a ser incluída posteriormente na tradução.

O algoritmo usa três tipos de mensagens de controle:

1. ***Initialization:*** Essa mensagem é enviada a todos os componentes, e indica a inicialização da Fase 1 ou Fase 2, alternadamente;

2. ***Idle:*** Essa mensagem significa que cada processo na sub–árvore abaixo esteve passivo pelo menos uma vez desde a última mensagem *idle*.

3. ***Activity:*** Essa mensagem é enviada para o pai durante a Fase 2 e contém contador de mensagens (criação e processamento) nas sub–árvores abaixo do processo que originou a mensagem.

Dentro do algoritmo, cada elemento mantém os seguintes contadores:

n_c: o somatório de mensagens primárias geradas pelo processo;

n_p: o somatório de mensagens primárias recebidas pelo processo;

N_c: o somatório de mensagens primárias criadas pelo processos na sub–árvore abaixo;

N_c: o somatório de mensagens primárias recebidas pelo processos na sub–árvore abaixo;

Todos são setados em zero no início da Fase 1 e Fase 2, e são enviados como parâmetros em mensagens *idle* e *activity*.

Na Fase 1, cada folha que alcance o estado passivo envia uma mensagem *idle* a seu pai, com parâmetros $N_c = n_c$ e $N_d = n_p$. Todos os demais componentes esperam até ter recebido uma mensagem *idle* de cada filho, adicionando os valores recebidos em N_c e N_d adequadamente. Uma vez que isso tenha ocorrido, quando o processo torna–se passivo, ele envia uma mensagem *idle* as seu pai, sucessivamente. Quando o nodo raiz recebe todas as mensagens, ele avalia (comparando

os valores locais de N_p e N_d) se o sistema alcançou uma possível configuração final. Em caso negativo, o nodo raiz considera a Fase 1 ainda ativa.

Em caso positivo, o nodo raiz envia a todos os filhos da mensagem *initiation*, sinalizando o início da segunda fase. O *broadcast* continua até alcançar os nodos folha. De maneira semelhante à Fase 1, mensagens *idle* são enviadas. Quando o processo raiz estiver de posse de todas as mensagens necessárias, compara se os valores obtidos são os mesmos da rodada anterior. Em caso negativo, o nodo raiz inicia a Fase 1 novamente. Em caso positivo, o nodo raiz sinaliza a terminação para os demais processos. São utilizadas duas fases para evitar casos em que numa configuração intermediária qualquer possa ser caracterizado um falso positivo.

4.2 Experimentação e Medidas de Desempenho

Nesta seção são apresentadas medidas de desempenho relativos à execução do código gerado pela metodologia apresentada. Todos os testes foram realizados no CPAD–PUCRS (Centro de Pesquisa em Alto Desempenho).

Inicialmente são apresentados dois testes para avaliar o desempenho do sistema enviando e recebendo mensagens. O objetivo principal destes testes é medir a estabilidade do modelo enquanto executando operações simples. A Figura 18 apresenta o conjunto de regras utilizadas no modelo, e na Figura 19 são apresentados dois exemplos de grafos iniciais relativos aos testes realizados. Os demais grafos iniciais não serão mostrados por questões de espaço.

4.2.1 Modelo Troca de Mensagens – ReSend, Rajada

No teste 1 existem dois processos GGBO (A e B) e apenas uma mensagem *SendMe* sempre em trânsito, sendo continuamente recebida e reenviada um determinado número de vezes (regra *Test_ReSend*). Em cada execução, os tempos foram tomados na primeira aplicação da regra *Test_ReSend* e na última aplicação da regra *Test_Terminate*. Como esperado, o comportamento do sistema gerado se manteve uniforme, com o tempo de execução aumentando linearmente devido aumento do número de reenvios.

No segundo teste, o processo A age como gerador de uma seqüência de mensagens destinadas ao processo B. Aqui, a aplicação de uma única regra *Test_Rajada* gera uma rajada de mensagens de A para B. O gráfico mede a variação do tempo conforme o aumento gradual do número total de mensagens da rajada. Os tempos foram tomados na aplicação de *Test_ReSend*

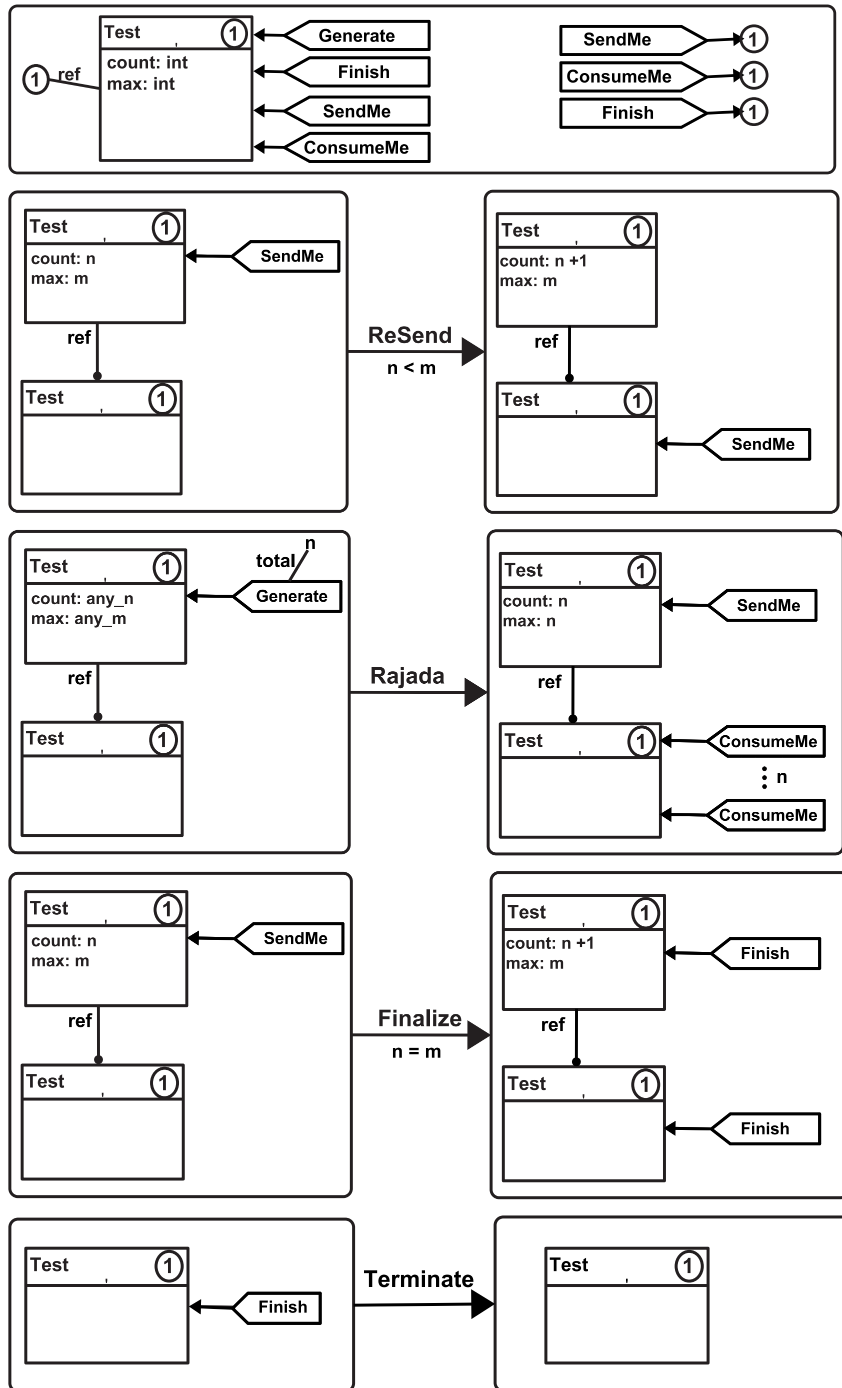
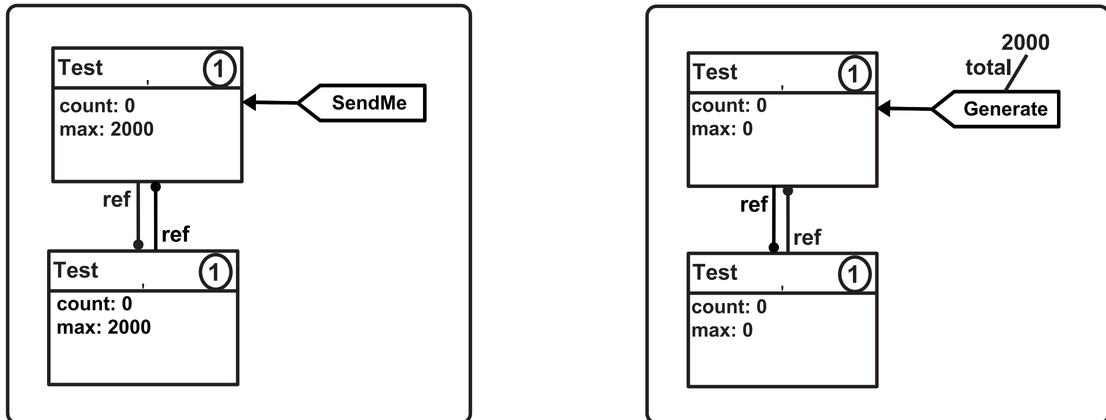


Figura 18: Grafo–tipo e regras para o objeto Teste.

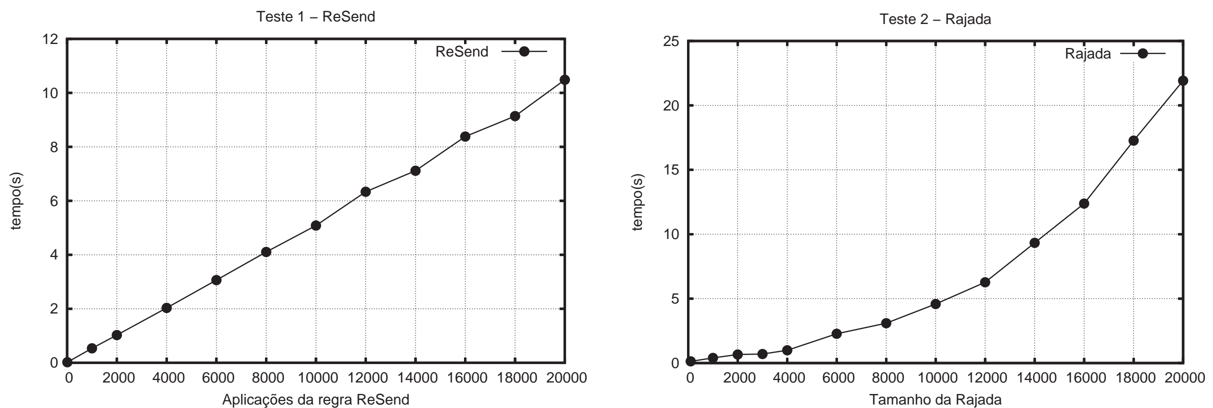
e *Test_Terminate*.

A curva obtida para o segundo teste apresenta um crescimento polinomial em relação ao aumento do conjunto de mensagens. Isso ocorre devido ao *overhead* introduzido pelo aumento



(a) Grafo Inicial – ReSend de Mensagens. (b) Grafo Inicial – Rajada de Mensagens.

Figura 19: Testes para Mensagens – Grafos Iniciais.



(a) Tempos para ReSend de Mensagens. (b) Tempos para Rajada de Mensagens.

Figura 20: Testes para mensagens – Tempos de Execução.

no número de *matchings* existentes, proporcional ao número de mensagens em L_{in}. O tamanho de L_{in} influencia no número de operações necessárias para a construção de $L_{matches}$[6]. Por sua vez, o tamanho de $L_{matches}$ influencia no tempo para localização do par escolhido dentro dessa lista. Essas influências são somadas ao tempo de envio/recebimento necessário para todas as mensagens do conjunto em cada situação, dando à curva o perfil observado.

[6]O número de regras existente também influencia no custo de construção, porém essa influência é um fator de multiplicação constante durante a execução do sistema

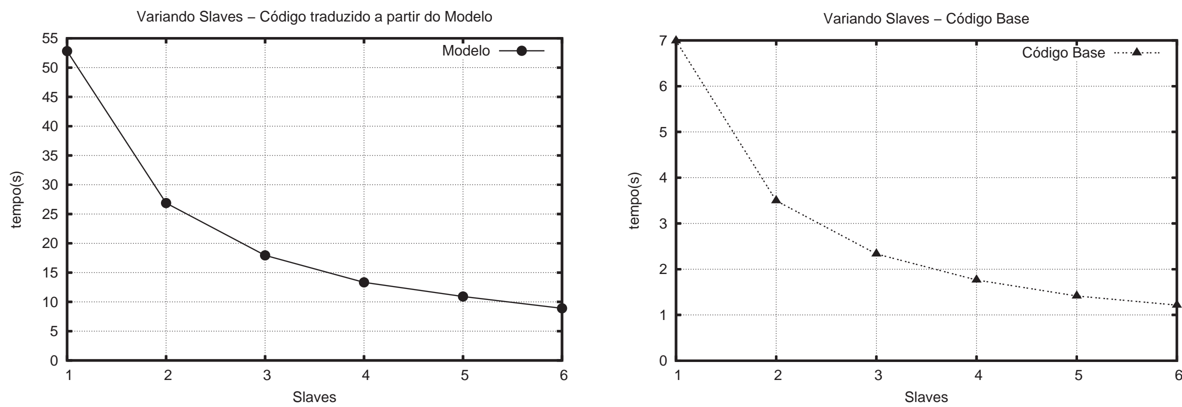
4.2.2 O Modelo Pi

Os gráficos da Figura 22 referem–se ao modelo apresentado na Seção 3.1, variando–se o número de processos do tipo *Slave* participantes e tomando–se o tempo total de execução do sistema. O gráfico 22(a) refere–se à performance observada para um programa compilado a partir do código gerado na tradução. A curva na Figura 22(b) traz medições obtidas para um algoritmo paralelo simples (Figura 21) gerado manualmente, cujo funcionamento é análogo ao do modelo GGBO proposto. Este programa será referenciado como programa base. A Figura 23 mostra uma comparação entre os resultados obtidos.

```
1   void main (n_mesgs)
2   {
3     int inside;                       /* dentro do círculo */
4     int total;                        /* total considerado */
5
6     MPI_Init ();                      /* inicializa MPI */
7
8     if (my_id != 0)                   /* Código SLAVE */
9     {
10          MPI_Recv (&total);          /* recebe parcela */
11
12         for (i = 0; i < total; i++)  /*loop principal */
13         {
14              px = rn (0, 100);
15              py = rn (0, 100);
16              if (ins (px, py) == true)
17              inside++;
18         }
19
20        MPI_Send (&inside);           /* envia resultado */
21        MPI_Recv (&kill);             /* terminação */
22    }
23    else if (my_id == 0)              /* Código MASTER */
24    {
25        total = n_mesgs;
26        buff = n_mesgs / (n_procs - 1);
27
28        for (i = 1; i < n_procs; i++)
29          MPI_Send (&buff);
30
31        for (i = 0; i < n_procs - 1; i++)  /* recebe */
32        {
33             MPI_Recv (&buff);
34             inside = inside + buff;
35        }
36
37        pi = (4 * inside) / total;    /* calcula */
38
39       /* broadcast de mensagem de encerramento */
40       for (i = 1; i < n_procs; i++)
41         MPI_Send (&kill);
42
43       printf (" %f\n ", pi);
44    }
45
46    MPI_Finalize ();                  /* finaliza MPI */
47  }
```

Figura 21: Pseudocódigo de Base.

Pode–se observar que os tempos obtidos a partir do código traduzido são em torno de seis vezes maiores que os relativos ao programa base (apresentando porém o mesmo comportamento). Isso acontece porque no exemplo existe um laço de repetição modelado como aplicação de re-



(a) Código gerado pela tradução do modelo.

(b) Código gerado manualmente.

Figura 22: Medidas de desempenho.

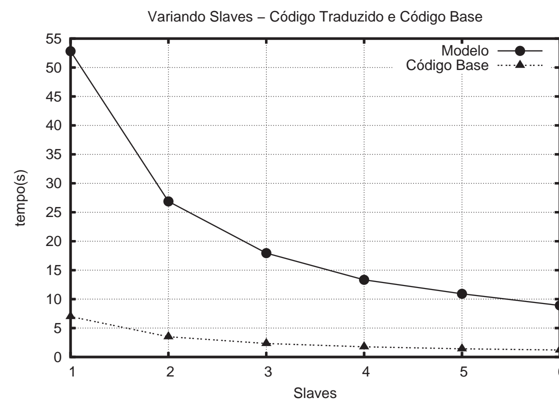


Figura 23: Comparação entre resultados.

gras. Regras como *Slave_Continue* são utilizadas em GGBO para modelar um comportamento seqüencial de objetos. Assim, o que na execução do programa base é feito em cada passada dentro do *loop* principal, no código do processo GGBO é feito através da geração e envio de uma mensagem indicando o próximo ponto a ser avaliado.

Uma modelagem mais realista não utilizaria tal estratégia, e sim faria uso de funções externamente definidas. Assim, substituindo as sucessivas execuções da regra *Slave_Continue* pela chamada a uma função seqüencial idêntica ao laço executado no corpo do processo escravo do programa base (linhas 11 a 17 da Figura 21), têm–se a regra apresentada na Figura 24.

Observe que agora toda a geração e avaliação dos pontos é feita pela função *Rand_Eval*(), cujo resultado é inserido no campo *total* da mensagem *InsPoints* sendo enviada ao objeto

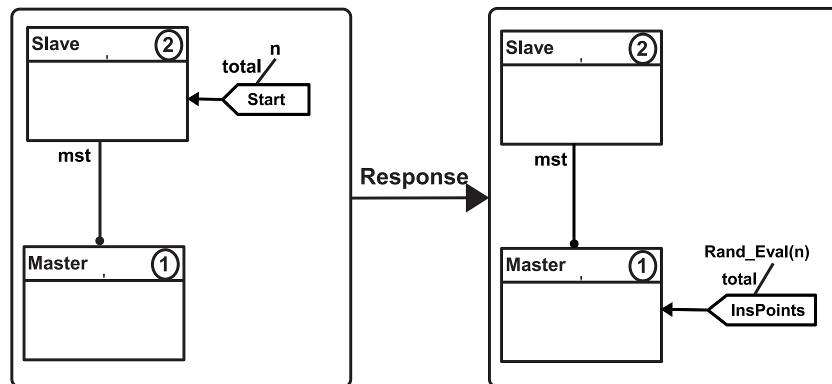


Figura 24: Nova regra para o objeto *Slave*.

Master. Ainda, tal modificação elimina a mensagem *Point* do sistema, diminuindo para dois o número total de campos necessários nas mensagens sendo trocadas. Feita a geração do código relativo ao modelo modificado, obtiveram–se medidas de desempenho próximas ao registrado com o programa base, como pode ser visto na Figura 25.

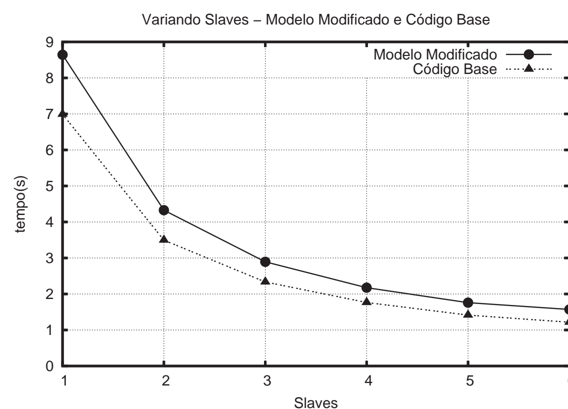


Figura 25: Comparando programa base e modelo modificado.

A diferença encontrada nas medições de tempo aproxima–se gradativamente de um valor constante, à medida que a paralelização diminui o processamento seqüencial em cada nodo. Esta diferença existe porque os custos de comunicação entre os modelos são diferentes – enquanto no algoritmo base somente é necessário o envio de um valor inteiro entre mestre e escravo (o que é realizado em uma única chamada à plataforma MPI) no código traduzido são necessárias (neste caso) quatro operações relativas ao empacotamento/desempacotamento de mensagens, assim como a manipulação de listas encadeadas. Outro fator que leva a tempos diferentes são as

operações de *matching* realizadas pelo objeto antes da execução de regras.

É importante lembrar que o algoritmo modelado é bastante simples, sendo apenas utilizado como exemplo inicial para o estudo de aplicações paralelas. A utilização de geração de código a partir de modelagem e tradução se aplica melhor a sistemas que contêm cenários mais complexos durante a execução, onde as possibilidades de análise oferecidas pela utilização de GGBO – como verificação e simulação – podem também ser utilizadas como apoio durante o desenvolvimento.

5 Semântica do Modelo Traduzido

Na Seção anterior foram apresentadas as estruturas de programação utilizadas na tradução de um modelo GGBO para código fonte C/MPI. Nesta seção, será apresenta a verificação do algoritmo proposto para os objeto GGBO. Isso será feito para provar que a semântica de um objeto GGBO é respeitada pelo programa gerado na tradução. Outra alternativa para tal objetivo seria realizar uma prova formal de que a correspondência existe. Porém, tal prova necessitaria da definição formal tanto para a linguagem de programação quanto para a biblioteca de comunicação utilizadas.

Para possibilitar a verificação do algoritmo, a estrutura básica gerada pela tradução GGBO–C/MPI – um *template* para processos – será convertida em um modelo PROMELA e verificada com a ferramenta SPIN. Será elencado um conjunto de propriedades – os requisitos do sistema – a serem oferecidos por um objeto GGBO. Essa propriedades todas deverão ser obedecidas pelo programa gerado para que o comportamento apresentado seja correto.

Na elaboração do modelo para verificação será mantida uma grande correspondência entre o código fonte C/MPI e o código PROMELA utilizado, de modo que o conjunto de propriedades verificado possa ser aceito como verdadeiro para o código gerado. Durante o transcorrer do Capítulo serão discutidos em mais detalhes a correspondência entre as estruturas de código propostas e as abstrações utilizadas para a verificação.

5.1 O Modelo GGBO – Propriedades

São esperadas as seguintes propriedades de um programa que implemente um objeto GGBO:

- **Propriedade I** – ***Todas as mensagens com match são consumidas:*** As mensagens existentes são consumidas até que não exista mais nenhuma presente no grafo de estado,

ou que não existam mais regras habilitadas por essas mensagens.

- **Propriedade II** – ***Qualquer mensagem com match pode ser consumida:*** Existindo mais de uma mensagem disponível no grafo de estado do sistema, endereçadas a um mesmo objeto, qualquer uma dessas mensagens pode ser consumida se existir pelo menos uma regra habilitada.

- **Propriedade III** – ***Mensagens podem ser consumidas em qualquer ordem:*** Existindo mais de uma mensagem disponível para um mesmo objeto no grafo de estado do sistema, não deve existir nenhuma ordenação implícita no consumo de mensagens resultante de alguma característica presente no mecanismo de *matching* utilizado, a menos que existam condições associadas às regras que determinem tal ordenação.

- **Propriedade IV** – ***Regras independentes podem ser aplicadas em qualquer ordem:*** Da mesma forma que foi apontado no item anterior, não deve existir nenhuma ordenação implícita na aplicação de regras independentes. Essa propriedade deve ser verdadeira mesmo quando essas regras puderem ser disparadas por mensagens idênticas.

Para verificar essas propriedades, foi elaborado o seguinte modelo GGBO:

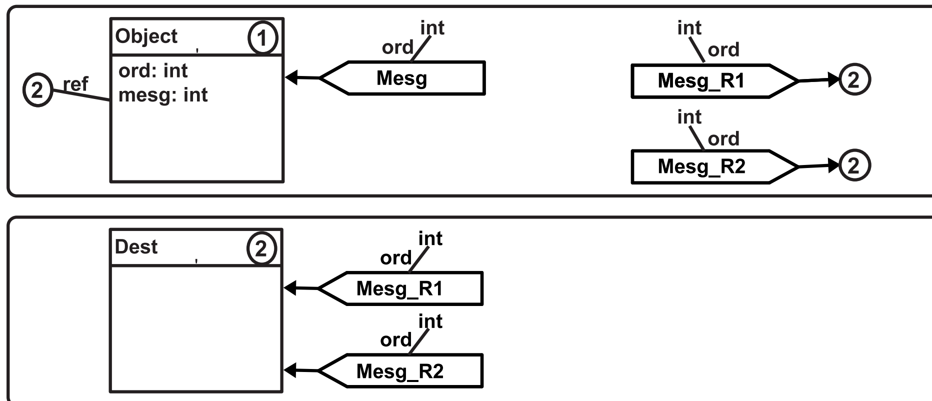


Figura 26: Modelo para Verificação – Grafos–Tipo.

A mensagem consumida por *Object* carrega apenas um parâmetro inteiro. No modelo, existem duas regras (*Rule*_1 e *Rule*_2), sem nenhuma condição associada, ambas relacionadas ao consumo do mesmo tipo de mensagem. Na aplicação de uma regra, *Object* consome a mensagem de entrada e envia outra para o objeto *Dest* referenciado, atualizando seus atributos internos de maneira a armazenar o inteiro contido na mensagem (atributo *mesg*) e o total de mensagens

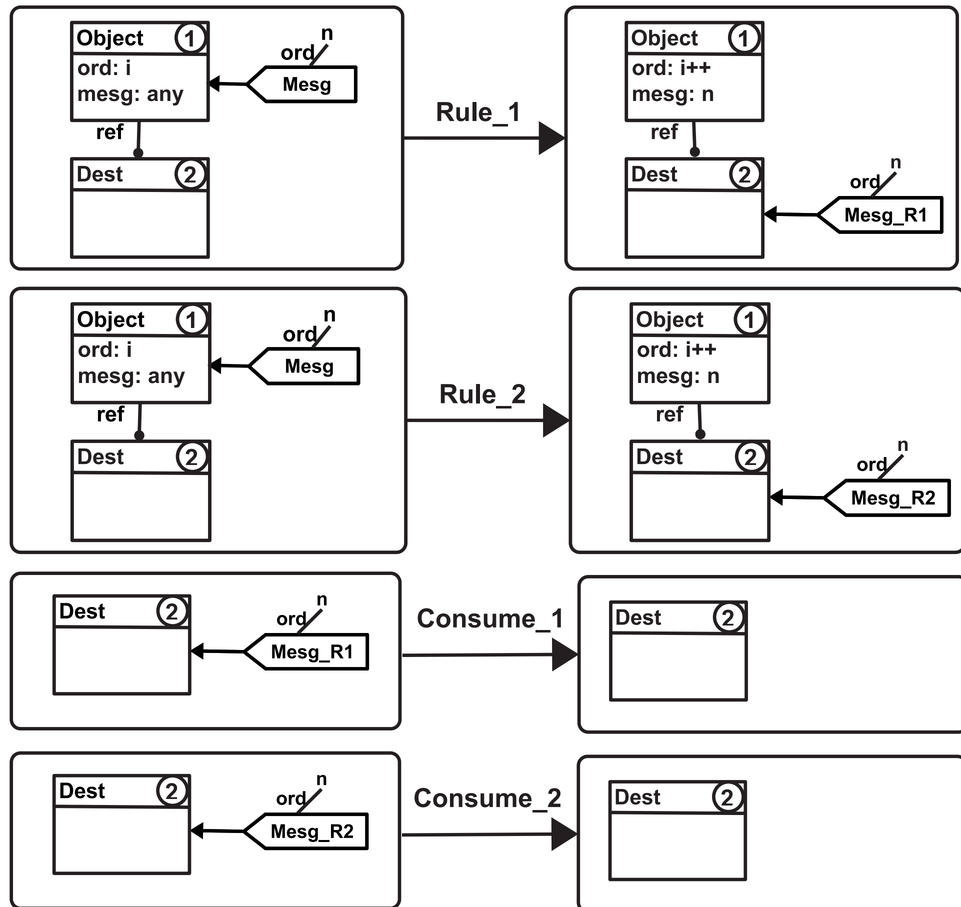


Figura 27: Modelo para Verificação – Regras.

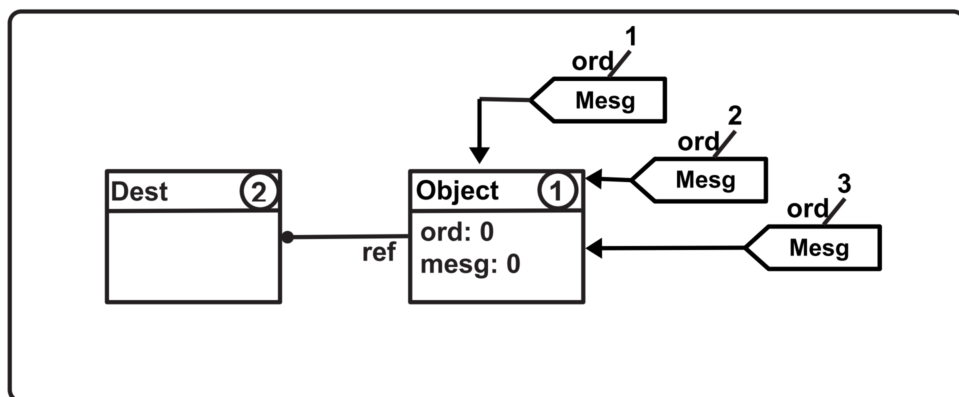


Figura 28: Modelo para Verificação – Grafo Inicial

consumidas (atributo *ord*). Existem dois tipos de mensagem enviadas a *Dest*, cada um associado à regra que a originou. Não existem condições associadas às regras formuladas. Assim, qualquer uma das mensagens presentes pode ser consumida em qualquer ordem, além de poder disparar

a aplicação de qualquer regra.

O modelo proposto é bastante simples, não sendo necessária a utilização de nenhum TAD associado. Assim, o código fonte correspondente é elaborado apenas em termos da estrutura proposta no Capítulo 4. A partir do modelo GGBO apresentado foi gerado o código C/MPI correspondente, e a partir desse código foi elaborado um modelo PROMELA análogo, que foi utilizado para a análise.

5.2 Convertendo o *Template* utilizado

No código do processo GGBO foi utilizada uma estratégia baseada no uso de três *threads* distintas para o recebimento, tratamento e envio de mensagens. Essas *threads* interagem entre si através do uso de listas encadeadas para o armazenamento das mensagens. Cada uma das *threads* de processamento será mapeada para um processo PROMELA distinto.

Chamadas a procedimentos e funções serão substituídas no modelo por chamadas a estruturas PROMELA *inline*. Tais extruturas definem trechos de código PROMELA que são inseridos no modelo no ponto em que foram invocados. Essas estruturas não possuem parâmetros de invocação ou variáveis de retorno. Dessa forma, o modelo para procedimentos/funções é elaborado de modo que as atualizações ocorram modificando diretamente as variáveis do processo que chamou o *inline*.

Semáforos binários são mapeados para canais de comunicação de tamanho 1. Assim, quando um processo execute *down(mutex)* (que fica mapeado para uma operação de *receive* no canal), este ficará bloqueado caso o valor de mutex seja zero (ou seja, se o canal estiver vazio). De maneira análoga, variáveis de condição ficam modeladas através do uso de canais de comunicação, porém nesse caso são utilizados canais com uma maior capacidade de armazenamento. Quando outro processo realizar um *up(mutex)* (*i.e.* enviar uma mensagem para o canal), o processo bloqueado poderá continuar normalmente.

Existem elementos na linguagem de programação (*e.g.* ponteiros) que não são oferecidos pela linguagem de modelagem. Isso impossibilita a implementação de uma lista encadeada diretamente em PROMELA. Para a verificação, listas encadeadas serão representadas como *canais de comunicação*, e os elos da lista passarão a ser representados por mensagens presentes nesse canais. Canais de comunicação possuem operações fundamentais (inclusão de elemento, retirada de elemento e inspeção do primeiro elemento), semelhantes às de uma lista, possibilitando

```
1   proctype receiver( chan from_outside; chan to_evaluate){
2    int destino = 0;
3    mtype header;
4    int body;
5
6   do
7       ::(receiver_continue == true)->
8
9               /*pthread_mutex_lock (&mutex_MPI);    */
10              mutex_MPI ?dummy;
11
12              /*MPI_Iprobe( MPI_ANY_SOURCE, ... ,  &flag, ...);*/
13
14              if
15              :: ( nempty(from_evaluate))->
16                     /*MPI_Recv( buff, TAM, MPI_PACKED, ...);  */
17                     from_outside?header, body;
18
29                     /*pthread_mutex_unlock (&mutex_MPI);     */
20                     mutex_MPI!dummy;
21
22                     /*MPI_Unpack(buff, TAM, &position, &buffer_mesg.type, ...); */
23
24                     /*pthread_mutex_lock  (&mutex_Incoming);    */
25                     mutex_Incoming?dummy;
26
27                     /*inseriu = Insere_Dir (&Incoming, buffer_mesg.type, &buffer_mesg, 0);  */
28                     to_evaluate!destino, header, body;
29
30                     /*pthread_mutex_unlock(&mutex_evaluate); */
31                     atomic{
32                     if
33                     :: (empty(cmutex_evaluate_simples))->mutex_evaluate! dummy;
34                     :: (nempty(cmutex_evaluate_simples))-> skip;
35                     fi;
36                     }
37
38                     /*pthread_mutex_unlock(&mutex_Incoming);*/
39                     mutex_Incoming!dummy;
40
41             ::else ->
42                     /*pthread_mutex_unlock (&mutex_MPI);   */
43                     mutex_MPI ?dummy;
44
45                     /*sched_yield();*/
46                     skip;
47              fi;
48
49              /*while (receiver_continue == true);*/
50      :: (receiver_continue == false)->  break;
51  od;
52  }
```

Figura 29: Modelo PROMELA para *Receiver*.

a modelagem do conjunto de operações utilizada no código gerado (concatenação, exclusão de elemento, inserção, pesquisa).

Na implementação original, foi utilizada alocação dinâmica de memória para as funções de manipulação de lista. Porém, canais são limitados a um número máximo de elementos. Apesar da utilização de alocação dinâmica oferecer uma grande flexibilidade no número de mensagens, ainda assim existe um limite no volume máximo de mensagens passivo de ser tratado. Dessa forma, a abstração de uma lista encadeada em um canal de mensagens finito não representa um impacto maior do que limitar o tamanho da memória disponível para o programa.

A exclusão mútua quanto ao acesso às listas de mensagens fica modelada através da utilização dos semáforos PROMELA correspondentes aos utilizados no código C. Referências a outros objetos GGBO são mapeadas para referências ao canal de comunicação utilizado pelo

```
1  proctype sender( chan from_evaluate; chan to_evaluate;  chan to_outside){
2  mtype header;
3  int body;
4  int destino;
5  int dummy;
6
7  do
8     ::(sender_continue == true)->
9        /*sem_wait (mutex_send)*/
10       mutex_send?dummy;
11
12       /*pthread_mutex_lock (&mutex_Outcoming);*/
13       mutex_Outcoming?dummy;
14
15       if
16       ::(nempty(from_evaluate))->
17           /*retirou = Retira_Esq (&Outcoming, &buffer_mesg, &dest); */
18           from_evaluate? destino, header, body;
19
20           /*pthread_mutex_unlock (&mutex_Outcoming);   */
21           mutex_Outcoming!dummy;
22
23           if
24           /*if(buffer_mesg.type == control_mesg)*/
25           ::(header == control_mesg)->
26                 skip;
27
28            /*if(dest!=my_id)*/
29           ::(destino != my_id)->
30                 /*MPI_Pack(&buffer_mesg.type,   1, MPI_INT, buff, ...); */
31
32                 /*pthread_mutex_lock (&mutex_MPI);    */
33                 mutex_MPI ?dummy;
34
35                 /*MPI_Send( buff, position, MPI_PACKED, dest, ...);    */
36                 to_outside!header, body;
37
38                 /*pthread_mutex_unlock (&mutex_MPI); */
39                 mutex_MPI !dummy;
40
41           fi;
42       /*else */
43       ::(empty(from_evaluate)) ->
44            /*pthread_mutex_unlock (&mutex_Outcoming);   */
45            mutex_Outcoming!dummy;
46
47       fi;
48
49    /*while (sender_continue == true); */
50    ::(sender_continue == false)->  break;
51 od;
52 }
```

Figura 30: Modelo PROMELA para *Sender*.

processo PROMELA que abstrai a *thread Receiver* do processo GGBO referenciado. Operações MPI síncronas *send/receive* são mapeadas para operações de envio e recebimento sobre canais PROMELA síncronos. Tais canais suportam apenas um tipo de estrutura de mensagem. A operação de checagem *MPI_ Iprobe()* responsável por identificar se existe ou não alguma mensagem a ser recebida será mapeada para o comando PROMELA *nempty()*, que retorna *true* caso exista alguma mensagem no canal, e *false* caso contrário.

Devido à grande correspondência entre o código gerado e o modelo PROMELA proposto, o espaço de estados gerado pelo modelo impossibilita a verificação de um modelo composto por dois processos GGBO completos. Para restringir o espaço de estados do modelo verificado, foi utilizada a técnica de verificação modular (*module checking* [48, 31]) na análise do comportamento

```
1   proctype evaluate ( chan from_Lin; chan to_Lout){
2     int destino;
3     mtype header;
4     int body;
5     chan Selected  = [10]  of {int,  mtype, int};
6     chan LMatches = [20]   of {int, int ,  int, mtype, int};
7
8    int retirou = false;
9    int total_matches=0;
10   int selected_match=0;
11   int selected_rule=0;
12   int dummy;
13
14  do
15    :: (evaluate_continue == true) ->
16              /*pthread_mutex_lock(&mutex_Incoming);*/
17              mutex_Incoming?dummy;
18
19             /*Concatena_Lista(&Selected,&Incoming);*/
20             concatena_lista_selected_incoming();
21
22             /*pthread_mutex_unlock(&mutex_Incoming);*/
23             mutex_Incoming!dummy;
24
25             /*total_matches = Do_Matching(Selected,&Matches);*/
26             total_matches_do_matching();
27
28             /*if (total_matches > 0)*/
29             if
30             :: (total_matches > 0) ->
31                     /*selected_match= rand()%total_matches;*/
32                     selected_match_rand_total_matches();
33
34                     /*Elo_temp = Consulta_Pos_par (Matches, selected_match, &selected_rule); */
35                     elo_temp_consulta_pos_par();
36
37                     /*retirou = Retira_No (&Selected, Elo_temp, &buffer, &dummy);*/
38                     retirou_retira_no()
39
40                     /*Libera_Memoria_par(&Matches); */
41                     libera_memoria_par_Matches();
42
43                     if
44                     :: (retirou == true)->
45                              if
46                              :: (selected_rule ==1 ) ->
47                                 n_mesgs_created = 1;
48                                 /*pthread_mutex_lock(&mutex_Outcoming);*/
49                                mutex_Outcoming?dummy;
50                                 to_Lout!destino, header, body;
51                                 /*or(loop_count=0;loop_count<n_mesgs_created;loop_count++)sem_post(&mutex_send);     ;*/
52                                 mutex_send!dummy;
53                                 event_rule_name= rule_1;
54                                /*pthread_mutex_unlock(&mutex_Outcoming);*/
55                                mutex_Outcoming!dummy;
56
57                              :: (selected_rule ==2 ) ->
58                                 n_mesgs_created = 1;
59                                 /*pthread_mutex_lock(&mutex_Outcoming);*/
60                                mutex_Outcoming?dummy;
61                                 to_Lout!destino, header, body;
62                                 /*for(loop_count=0;loop_count<n_mesgs_created;loop_count++)sem_post(&mutex_send);     */
63                                 mutex_send!dummy;
64                                 event_rule_name= rule_2;
65                                 /*pthread_mutex_unlock(&mutex_Outcoming);*/
66                                mutex_Outcoming!dummy;
67                              fi;
68                     fi;
69
70             ::(total_matches == 0) ->
71                    /*pthread_mutex_lock  (&mutex_evaluate); */
72                    mutex_evaluate?dummy;
73             fi;
74
75     /*while(evaluate_continue==true); */
76     :: (evaluate_continue == false) -> break;
77  od
78  }
```

Figura 31: Modelo PROMELA para *Evaluate*.

```
1   inline concatena_lista_selected_incoming(){
2     atomic{
3     do
4       :: (len(from_Lin)>0) ->
5            from_Lin?destino, header, body ;
6         Selected!destino, header, body ;
7       :: else -> break;
8     od
9    }
10  }
```

```
1   inline libera_memoria_par_Matches(){
2   int dummy1, dummy2, dummy3, dummy4, dummy5;
3    atomic{
4      do
5        :: (nempty(LMatches)) ->
6            LMatches?dummy1, dummy2, dummy3, dummy4, dummy5;
7        :: (empty(LMatches)) -> break;
8      od
9    }
10  }
```

Figura 32: Modelos PROMELA – Concatenação de Listas e Deleção de Lista.

```
1   inline total_matches_do_matching(){
2           /*total_matches =1;*/
3           /*tirar todas as mesgs de Selected*/
4           /**/
5           int count;
6           int max ;
7           count=0;
8     atomic{
9           max = len(Selected);
10          do
11          ::(count < max)->
12              count ++;
13              LMatches! 1,  count, 0, mesg,1;
14              LMatches! 2,  count, 0, mesg,1;
15          ::(count >=max)-> break;
16          od;
17          total_matches = len(LMatches);
18    }
19  }
```

```
1   inline elo_temp_consulta_pos_par(){
2     int count;
3           atomic{
4           count= 0;
5           do
6           ::(count != selected_match)->
7                   count++;
8                   LMatches?selected_rule, elo_temp, destino, header, body;
9           ::(count ==selected_match)-> break;
10          od;
11          the_elo_temp = elo_temp;
12          }
13  }
```

Figura 33: Modelos PROMELA – *Matching* e Retirada de par Mensagem–Regra.

```
1   inline selected_match_rand_total_matches(){
2       /*selected_match= rand()%total_matches;*/
3     atomic{
4       if
5           ::(total_matches >=1  ) ->  selected_match = 1;
6           ::(total_matches >=2  ) ->  selected_match = 2;
7           ::(total_matches >=3  ) ->  selected_match = 3;

...         ...

24          ::(total_matches >=20)->  selected_match = 20 ;
25          :: else -> printf("MSC: chabu");
26      fi;
27      the_selected_match =  selected_match;
28      printf("MSC: total: %d  -  selected_match: %d\n", total_matches,  selected_match);
29    }
30  }
```

```
1   inline retirou_retira_no(){
2    int cont;
3    int destino;
4    mtype  header;
5    int body;
6    atomic{
7           cont=1;
8           if
9            ::nempty(Selected) ->
10               do
11                   ::(cont<elo_temp)->
12                      Selected? destino, header, body;
13                      Selected! destino, header, body;
14                       cont++;
15                   ::(cont==elo_temp)-> break;
16                   od;
17
18                   retirou = true;
19                   Selected?destino, header,body;
20                   the_consumed_mesg=body;
21                   consumed_mesg++;
22           ::empty(Selected) -> retirou = false;
23          fi;
24   }
25  }
```

Figura 34: Modelos PROMELA – Escolha de par Mensagem–Regra e Consumo de Mensagem.

do processo GGBO gerado. Na verificação modular, a verificação do sistema é realizada através da verificação de suas partes integrantes.

Além dos processos necessários ao funcionamento do módulo sendo analisado são gerados dois processos adicionais: um *driver*, responsável por gerar as mensagens que irão disparar o funcionamento do elemento sendo verificado, e um *stub*, responsável por consumir as mensagens geradas e, eventualmente, gerar as respostas necessárias para o funcionamento do módulo.

Para a verificação do modelo apresentado, será tomado como módulo do processo GGBO os três processos PROMELA correspondentes às *threads Sender*, *Evaluate* e *Receiver*. O *driver* PROMELA ficará responsável por gerar três mensagens *mesg* com parâmetros em ordem crescente. O *stub* utilizado realiza apenas o consumo das mensagens enviadas pelo módulo. Observe que, apesar dessa abstração, existe uma grande semelhança entre as funções desempenhadas por INIT e pelo *driver*, assim como entre o objeto *Dest* e o *stub*.

Uma vez definida a estrutura do modelo PROMELA a ser utilizado, pode–se partir para a verificação das propriedades propostas.

5.3 Verificando as propriedades definidas

5.3.1 Propriedade I – *Todas as mensagens com match são consumidas*

Para provar que todas as mensagens com *match* podem ser consumidas foi incluída no modelo a variável global *consumed_mesg_value*, atualizada por *Evaluate*. Nessa variável é copiado o valor do atributo *mesg* de *Object*. Esse atributo duplica o valor contido na mensagem consumida pela aplicação de uma regra, tanto para *Rule_1* quanto *Rule_2*. Observe no modelo que essas regras estão sempre habilitadas pela existência de uma mensagem, portanto, sempre existem *matches* no modelo, enquanto existirem mensagens.

A partir dessa variável torna–se possível observar qual o estado em que o modelo se encontra durante o seu funcionamento. Sobre ela foram definidas as seguintes proposições atômicas:

#define m1 (consumed_mesg_value ==1)
#define m2 (consumed_mesg_value ==2)
#define m3 (consumed_mesg_value ==3)

A partir dessas proposições foram elaboradas e verificadas as seguintes sentenças:

Eventualmente m1 é consumida: (<> m1) : *true*;
Eventualmente m2 é consumida: (<> m2) : *true*;
Eventualmente m3 é consumida: (<> m3) : *true*;

É importante lembrar que em toda sentença LTL existe implicitamente subentendido o operador de caminhos A, ou seja, a sentença sempre é verificada “para todas as possíveis seqüências de estados do modelo”. Assim, fica provado que, em todos os possíveis caminhos oferecidos pela

execução do modelo, em algum estado futuro a partir do inicial, a mensagem $m1$ é consumida, assim como no futuro $m2$ e $m3$ são consumidas.

5.3.2 Propriedade II – *Qualquer mensagem com match pode ser consumida*

Para provar que mensagens podem ser consumidas em qualquer ordem foi definida mais uma variável global, *consumed_mesg_ord*, também modificada por *Evaluate*. Nessa variável, inicializada em zero, é copiado valor presente no atributo *ord* de *Object* atualizado na aplicação de uma regra. Esse atributo sempre é incrementado na aplicação de uma regra, e indica qual é a ordem da mensagem sendo consumida (primeira, segunda, *etc.*) no momento da aplicação.

Sobre essas duas variáveis foram definidos as seguintes proposições atômicas:

```
#define first_m1 (consumed_mesg_value ==1 && consumed_mesg_ord ==1)
#define first_m2 (consumed_mesg_value ==2 && consumed_mesg_ord ==1)
#define first_m3 (consumed_mesg_value ==3 && consumed_mesg_ord ==1)
```

Inicialmente, foram verificadas as seguintes sentenças:

Eventualmente m1 é consumida primeiro: (<> first_m1) : *false*
Nunca m1 é consumida primeiro: ([ ] ! first_m1) : *false*

Pelas propriedades provadas na Seção anterior, sabe–se que tanto $m1$ quanto $m2$ e $m3$ são efetivamente consumidas em um determinado momento da execução do modelo. Pelas primeiras duas sentenças verificadas nesta Seção, têm–se que existem tanto casos onde $m1$ não é a primeira mensagem consumida (nos contra–exemplos resultantes da primeira verificação), quanto casos onde $m1$ é a primeira mensagem consumida (nos contra–exemplos da segunda).

Analogamente, pode–se expressar as sentenças para $m2$:

Eventualmente m2 é consumida primeiro: (<> first_m2) : *false*
Nunca m2 é consumida primeiro: ([ ] ! first_m2) : *false*

Ou seja, pelos contra–exemplos, existem casos onde $m2$ não é a primeira consumida, assim como casos onde $m2$ é a primeira consumida. Igualmente para m3:

Eventualmente m3 é consumida primeiro: (<> first_m3) : *false*
Nunca m3 é consumida primeiro: ([ ] ! first_m3) : *false*

Dessa forma, pode–se concluir que qualquer uma das três mensagens pode ser consumida em primeiro lugar. Pode–se ainda provar que qualquer uma das mensagens pode ser a primeira a ser consumida em uma única sentença verdadeira. Para tanto são aplicadas ao modelo:

Eventualmente m1 é consumida primeiro,
ou eventualmente m2 é consumida primeiro:
(<> first_m1)||(<> first_m2) : *false*

Eventualmente m1 é consumida primeiro,
ou eventualmente m3 é consumida primeiro:
(<> first_m1)||(<> first_m3) : *false*

Eventualmente m2 é consumida primeiro,
ou eventualmente m3 é consumida primeiro:
(<> first_m2)||(<> first_m3) : *false*

Provadas essas propriedades, foi verificada a seguinte sentença:

Eventualmente m1 é consumida primeiro,
ou Eventualmente m2 é consumida primeiro,
ou Eventualmente m3 é consumida primeiro:
(<> first_m1)||(<> first_m2)||(<> first_m3) : *true*

Conforme observado nas propriedades verificadas, cada um dos termos do "ou" é falso isoladamente, e também cada par possível de ser composto a partir das subfórmulas[1]. A sentença é verdadeira somente quando todos as mensagens forem consideradas, ou seja, quando enunciar que tanto $m1$ quanto $m2$ quanto $m3$ puderem ser a primeira mensagem consumida.

De maneira análoga, a partir da definição das proposições:

```
#define sec_m1 (consumed_mesg_value ==1 && consumed_mesg_ord ==2)
#define sec_m2 (consumed_mesg_value ==2 && consumed_mesg_ord ==2)
#define sec_m3 (consumed_mesg_value ==3 && consumed_mesg_ord ==2)

#define third_m1 (consumed_mesg_value ==1 && consumed_mesg_ord ==3)
#define third_m2 (consumed_mesg_value ==2 && consumed_mesg_ord ==3)
#define third_m3 (consumed_mesg_value ==3 && consumed_mesg_ord ==3)
```

[1] Isso porque o termo não considerado na fórmula surge como contra–exemplo para a sentença.

foi provado para o modelo utilizado que:

Eventualmente m1 é a segunda consumida: (<> sec_m1) : *false*

Nunca m1 é a segunda consumida: ([ ] ! sec_m1) : *false*

Eventualmente m1 é a terceira consumida: (<> third_m1) : *false*

Nunca m1 é a terceira consumida: ([ ] ! third_m1) : *false*

Eventualmente m2 é a segunda consumida: (<> sec_m2) : *false*

Nunca m2 é a segunda consumida: ([ ] ! sec_m2) : *false*

Eventualmente m2 é a terceira consumida: (<> third_m2) : *false*

Nunca m2 é a terceira consumida: ([ ] ! third_m2) : *false*

Eventualmente m3 é a segunda consumida: (<> sec_m3) : *false*

Nunca m3 é a segunda consumida: ([ ] ! sec_m3) : *false*

Eventualmente m3 é a terceira consumida: (<> third_m3) : *false*

Nunca m3 é a terceira consumida: ([ ] ! third_m3) : *false*

E, a partir dessas sentenças[2]:

(<> sec_m1)||(<> sec_m2)||(<> sec_m3) : *true*

(<> third_m1)||(<> third_m2)||(<> third_m3) : *true*

A partir das propriedades provadas para o modelo, pode–se concluir que qualquer uma das três mensagens geradas pelo *driver* pode ser consumida na aplicação da primeira, da segunda ou da terceira regra.

5.3.3 Propriedade III – *Mensagens podem ser consumidas em qualquer ordem*

Apesar de $m1$ poder ser consumida tanto na aplicação da primeira quanto da segunda ou terceira regra (o mesmo valendo para $m2$ e $m3$), ainda pode existir uma ordem implícita no consumo das mensagens. Por exemplo, suponha que exista uma dependência circular no consumo de mensagens em uma ordem $(m1, m2, m3)$, sendo que tal dependência faça que o consumo de

[2] Da mesma forma que foi realizado para a sentença anterior que considera as três mensagens, todas as combinações de subfórmulas possíveis dentro de cada uma destas sentença foram verificadas, resultando todas em falsidade. Tais sentenças foram omitidas por questão de espaço.

$m1$ primeiro force o consumo de $m2$ e $m3$ nessa seqüência, o consumo de $m2$ primeiro imponha $m3$ e $m1$ como conseqüentes, e o consumo de $m3$ primeiro faça $m1$ e $m2$ serem os próximos. Tal ordenação não fica excluída nas propriedades provadas até então. Assim, resta provar que a ordem em que uma mensagem é consumida não influencia na ordem que as demais o são[3].

A partir dos predicados definidos para provar a propriedade anterior, as seguintes propriedades foram verificadas:

- $m1$ é consumida antes de $m2$;
- $m1$ é consumida depois de $m2$;

As propriedades enunciadas foram formalizadas em LTL utilizando os padrões de sentenças apresentados em [16, 59].

m1 é consumida antes de m2: <> m2 -> ! (! m1 U m2) : *false*

m1 é consumida depois de m2: <> m2 -> <>(m2 && <> m1) : *false*

Para a primeira sentença, os contra–exemplos mostrados envolviam tanto $m1$ sendo consumido depois de $m2$ quanto $m3$ sendo consumido antes de $m2$. A segunda sentença por sua vez pode ser violada fazendo–se $m1$ ser consumida antes de $m2$ (exatamente o comportamento verificado na sentença anterior) ou fazendo–se $m3$ ser consumida após $m2$. De maneira análoga ao realizado na subseção anterior, foi elaborada uma terceira sentença enunciando que

m1 é consumida antes de m2, *ou* m1 é consumida depois de m2:
(<> m2 -> ! (! m1 U m2)) || (<> m2 -> <>(m2 && <> m1)) : *true*

Assim, tem–se que $m1$ pode tanto ser consumida antes de $m2$ quanto depois de $m2$, ou seja, não existe uma dependência no consumo de $m1$ e $m2$. Da mesma forma, foram verificadas as seguintes sentenças sobre a ordenação entre $m1$ e $m3$:

m1 é consumida antes de m3: <> m3 -> ! (! m1 U m3) : *false*

m1 é consumida depois de m3: <> m3 -> <>(m3 && <> m1) : *false*

m1 é consumida antes de m3, *ou* m1 é consumida depois de m3:
(<> m3 -> ! (! m1 U m3)) || (<> m3 -> <>(m3 && <> m1)) : *true*

Por último, as seguintes sentenças correspondentes a m2 e m3 foram aplicadas ao modelo:

[3] Para os casos em que isso não estiver explicitamente previsto no modelo sob a forma de uma condição associada às regras incluídas no modelo

m2 é consumida antes de m3: <> m3 -> ! (! m2 U m3) : *false*

m2 é consumida depois de m3: <> m3 -> <>(m3 && <> m2): *false*

m2 é consumida antes de m3, *ou* m2 é consumida depois de m3:

(<> m3 -> ! (! m2 U m3)) || (<> m3 -> <>(m3 && <> m2)) : *true*

A partir das propriedades verificadas, pode–se concluir que não existe nenhuma ordem relativa no consumo de mensagens. Ainda, associando esse resultado ao obtido na subseção anterior, onde foi verificado que uma mensagem pode ser consumida em qualquer ordem, pode–se concluir que as mensagens são consumidas em todas as ordens possíveis.

5.3.4 Propriedade IV – *Regras podem ser aplicadas em qualquer ordem*

Durante o funcionamento do modelo, uma mesma regra pode ser aplicada várias vezes, ou pode haver uma intercalação entre a aplicação da primeira ou da segunda regra. Para possibilitar a detecção de qual regra foi aplicada durante o funcionamento do modelo, foi incluído no modelo a variável global *event_rule_name*, atualizada por *Evaluate* imediatamente após a execução do último comando relativo à regra aplicada. Essa variável recebe "rule_1" caso a regra aplicada tenha sido *Rule*_1, ou "rule_2", caso *Rule*_2 tenha sido executada. Ainda, antes da determinação de qual mensagem será consumida na aplicação de uma regra, o estado de *event_rule_name* é alterado para "nothing", possibilitando assim a detecção da aplicação de uma mesma regra sucessivas vezes.

Sobre o estado de *event_rule_name* foram definidas as seguintes proposições atômicas:

#define Rule1 (event_rule_name == rule_1)
#define Rule2 (event_rule_name == rule_2)

Para provar que regras podem ser aplicadas em qualquer ordem, inicialmente verificaram–se as seguintes sentenças:

Eventualmente regra1 é aplicada: <> Rule1 : *false*

Eventualmente regra2 e aplicada: <> Rule2 : *false*

Eventualmente regra1 é aplicada, ou Eventualmente regra2 é aplicada:

(<> Rule1) ||(<> Rule2) : *true*

De maneira análoga ao que ocorre com as sentenças simples verificadas para as propriedades anteriores, especificando–se apenas <>*Rule*1, obtém–se contra–exemplos mostrando que existem

casos em que apenas $Rule2$ é aplicada sucessivas vezes. Especificando–se apenas <>$Rule2$, obtém–se contra exemplos mostrando sucessivas aplicações de $Rule1$. Com a terceira sentença, prova–se que tanto $Rule1$ quanto $Rule2$ podem ocorrer para ao modelo.

Uma vez provado que ambas as regras podem ser aplicadas, resta provar que pode existir alternância nessa aplicação, ou seja, que os contra exemplos mostrados não são as únicas possibilidades em que $Rule1$ e $Rule2$ são aplicadas.

Para tanto, foram verificadas para o modelo as sentenças:

Regra 1 não é aplicada após Regra2:

[ ] (Rule2 -> [ ] ! Rule1) : *false*

Regra 1 não é aplicada antes de Regra2:

<> Rule2 -> (! Rule1 U Rule2) : *false*

Regra1 ou Regra2 podem ser aplicadas depois de Regra2:

<> Rule2 -> <> (Rule2 && <> (Rule1 || Rule2)) : *true*

Regra1 ou Regra2 podem ser aplicadas depois de Regra1:

<> Rule1 -> <> (Rule1 && <> (Rule1 || Rule2)) : *true*

Do primeiro conjunto de sentenças verificado, tem–se que tanto $Rule1$ quanto $Rule2$ podem ser aplicados no sistema. A partir dos contra–exemplos obtidos para as sentenças, pode–se concluir que existem tanto casos em que a mesma regra é aplicada sucessivamente quanto casos em que existe alternância na aplicação de regras.

Apesar do modelo apresentar algumas abstrações em relação ao *template* original, o código PROMELA foi elaborado procurando manter–se a mesma estrutura e semântica do *template* proposto. As estruturas *inline* para funções e procedimentos foram elaboradas de maneira a apresentar o mesmo comportamento presente nas oferecidas pelo programa original. Apesar de serem utilizados canais para representar listas encadeadas e semáforos, as operações sobre canais permitem simular o funcionamento das operações básicas, facilitando a modelagem de maneira geral. Assim, uma vez provadas as propriedades para o modelo do *template* utilizado, pode–se esperar que estas sejam obedecidas pelos programas gerados a partir do esqueleto proposto.

A seguir serão realizados estudos de caso, visando exemplificar a utilização de GGBO para a implementação de aplicações paralelas.

6 Estudos de Casos

Nesse capítulo são desenvolvidos estudos de casos visando ilustrar a utilização da técnica proposta. Para tanto, será apresentado um problema cuja solução possa ser implementada de maneira distribuída utilizando–se uma estratégia em fases paralelas. São apresentados dois modelos GGBO semelhantes para o problema. Finalmente, os modelos são verificados, sendo provadas propriedades relativas ao correto funcionamento e terminação.

6.1 Estudo de Caso – Perfil de Temperaturas em uma Peça Metálica

Considere uma fina peça de metal quadrada, da qual seja necessário determinar a distribuição de temperatura quando suas laterais estiverem em contato com diversas fontes de calor. Nessa situação, dentro de uma determinada faixa de valores[1], a temperatura do ambiente altera a temperatura do corpo, até que o sistema alcance uma configuração onde a temperatura em cada ponto da peça permanece inalterada.

A temperatura dos pontos presentes nas laterais da placa rapidamente alcança uma temperatura constante e igual à dos corpos aos quais ela está em contato. A temperatura dos pontos presentes na superfície interior da placa, por sua vez, depende da temperatura em torno dela, onde encontram–se as fontes de calor. A temperatura do interior pode ser calculada dividindo–se a placa em uma grade de pontos, e considerando na temperatura de cada ponto a influência dos pontos adjacentes. Recalculando sucessivas vezes essa influência, pode–se aproximar a variação de temperatura que ocorre dentro da peça. Essa estratégia de cálculo é conhecida como *método*

[1] A partir de um determinado patamar de temperatura, o material pode entrar em fusão. Para a situação analisada no decorrer desse Capítulo todos os valores de temperatura serão considerados abaixo de tal limite.

das diferenças finitas.

Para uma peça bidimensional como uma placa, a temperatura de cada ponto no interior do corpo é calculada como a média das temperaturas de seus quatro pontos vizinhos (quatro lados). O método pode ser estendido para um volume tridimensional considerando–se seis pontos vizinhos em vez de apenas quatro (quatro lados, acima, abaixo).

Suponha que a distribuição de temperatura original de todos os pontos da placa esteja armazenada em uma matriz M de dimensões $l \times c$ relativas ao tamanho da peça, e os pontos laterais estejam inicializados com o valor da temperatura externa correspondente. Cada posição $M_{i,j}$ da matriz associa–se a um ponto $h_{i,j}$ da peça. O cálculo da temperatura interna da placa é feito varrendo–se a matriz (exceto pelas linhas e colunas mais externas, em contato direto com o ambiente) e calculando–se a nova temperatura $h'_{i,j}$ de cada ponto como $h'_{i,j} = (h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}) \div 4$. Observe que existe a necessidade de uma segunda matriz para armazenar os novos valores calculados, caso contrário pode ocorrer uma situação em que sejam considerados pontos atualizados e pontos não–atualizados em uma mesma aplicação da função. Dessa forma, são utilizadas duas matrizes, M para armazenar o estado atual da placa, e M' para armazenar o próximo estado, calculado em função da temperatura de cada ponto no estado atual. Feito isso, copiam–se para M os novos valores contidos em M' e o processo continua[2].

Dentro de um mesmo passo de atualização, o novo valor de temperatura para cada ponto poderia ser calculado em paralelo com os demais. Por exemplo, utilizando uma *cellular processing language*, poderia–se reproduzir tanto a topologia quanto o cálculo descrito no problema. Pode–se também paralelizar o cálculo da distribuição utilizando–se troca de mensagens. Uma abordagem simples seria mapear cada ponto da peça em um processo separado, utilizando mecanismos de comunicação para consultar a temperatura do ponto adjacente. Dessa forma, cada processo enviaria quatro mensagens para seus vizinhos (informando sua própria temperatura atual) e obteria a temperatura correspondente da circunvizinhança. Após a troca de dados, todos atualizariam seus valores em um passo paralelo e novamente trocariam mensagens contendo suas temperaturas puntuais.

Tal estratégia, porém, seria extremamente custosa em termos de troca de mensagens. O

[2]Esse passo de cópia não é estritamente necessário. Contanto que tanto M quanto M' possuam as mesmas dimensões e a os mesmos valores para as bordas, pode–se realizar o cálculo de maneira alternada, avaliando os pontos de M para compor um perfil atualizado em M', e depois considerando os pontos de M' e definindo o novo perfil para M, sucessivas vezes.

problema pode ser amenizado fazendo–se cada processo responsável por uma área maior da peça, e enviando nas mensagens todo o perfil lateral de temperatura do segmento na forma de um vetor. Dessa forma, têm–se um número menor de mensagens sendo enviadas, porém cada mensagem carrega um conjunto maior de dados.

Devido ao fracionamento da matriz original, para a realização do cálculo indicado torna–se necessário replicar junto a cada pedaço da peça os pontos das peças vizinhas com o qual a fração faz contato. Isso pode ser feito associando ao perímetro de cada pedaço da placa original um conjunto extra de pontos, responsável por replicar o perfil de temperatura das bordas vizinhas. Assim, é adicionando um "contorno" de pontos à fração da peça, e para esse contorno é copiado o vetor de temperaturas presente em cada mensagem. Incluindo–se na mensagem um parâmetro para indicar a qual lado da peça a mensagem está relacionada, pode–se atualizar o trecho do contorno relativo à face correspondente. Uma vez que todo o perímetro esteja preenchido, a nova distribuição de temperaturas dentro da placa é calculada e, novamente, são enviados aos vizinhos mensagens contendo o perfil de temperatura do lado correspondente. Essas mensagens serão consumidas durante a fase seguinte.

Observe que peças cujas bordas estão em contato direto com o ambiente nunca recebem mensagens indicando a atualização daquele trecho do contorno. Assim, durante todo o processo de cálculo, os valores setados para o contorno na inicialização do sistema mantém–se constantes. Aproveitando essa característica, fazendo–se os trechos relativos ao contorno mais externo da placa original receber os valores de temperatura do ambiente, pode–se reproduzir a influência externa no aquecimento da peça.

Um arranjo onde todos os processos enviam suas mensagens primeiro para então receber as mensagens dos demais, conforme apresentado aqui, é apontado pela literatura como *inseguro* quando utilizando–se diretamente a biblioteca MPI [82]. Por exemplo, caso todos os nodos estejam utilizando primitivas síncronas de comunicação, inicializadas todas em mesma ordem, pode–se criar um cenário de *deadlock*, com todos os processos aguardando em um *receive*, por exemplo. Outra possibilidade seria a utilização de primitivas de comunicação bufferizadas (*BSend/BRecv*). A implementação de tais chamadas permite ao programa disponibilizar um volume de dados para o *deamon* de comunicação e continuar o processamento. Novamente, essa estratégia poder carregar problemas implicitamente, uma vez que o tamanho dos *buffers* não é especificado pelo padrão MPI [82].

Uma alternativa para tornar o código mais seguro seria a utilização alternada de *sends* e

receives, orquestrada de maneira a garantir que os processos vizinhos executem sempre operações complementares em uma ordem que evite o *deadlock*. Operações não–bloqueantes também podem ser utilizadas, disparando–se todos os *sends* e *receives* e utilizando–se uma chamada para aguardar que todas as comunicações se completem. Porém, ambas as soluções requerem um balanceamento natural no número de operações de envio e recebimento entre os participantes, o que nem sempre é trivial dentro de um sistema. Assim, tais estratégias não são suficientemente genéricas para que possam ser adotadas como uso comum.

A utilização de listas encadeadas para o armazenamento de mensagens, como é feito no código gerado a partir de um modelo GGBO, permite uma maior flexibilidade nos mecanismos de comunicação. Dessa forma, é oferecido um nível de abstração mais natural ao problema abordado, evitando que o programador preocupe–se em evitar alguma arranjo desastroso na ordem das operações de comunicação, facilitando a modelagem e correspondente implementação da solução.

6.1.1 O Modelo *HeatCell*

O modelo GGBO apresentado nas Figuras 35 a 40 implementa o método discutido. O sistema é composto apenas pela classe de objetos *Cell*. Cada objeto possui referências apontando os objetos vizinhos (atributos *up*, *dw*, *lf* e *rg*). Cada objeto possui um atributo *part*, relativo ao TAD *HeatCalc*, onde estão implementadas as estruturas de dados necessárias para a simulação de uma fração da peça metálica por um objeto *Cell*.

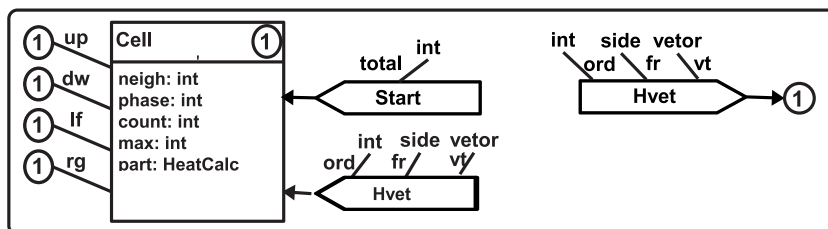


Figura 35: Grafo Tipo para Objeto *Cell*.

Recebendo uma mensagem *Start*, que indica no atributo m quantos passos devem ser realizados, *Cell* executa a regra *Cell_HStart*, enviando para toda a circunvizinhança o perfil de temperatura (parâmetro *vt*) nas bordas de sua fração da peça através de mensagens *Hvet*. Os demais parâmetros indicam qual é o passo de iteração que deve consumir a mensagem (parâmetro *ord*) e qual a procedência da mensagem (parâmetro fr, d significando *down*, u para *up*, r como

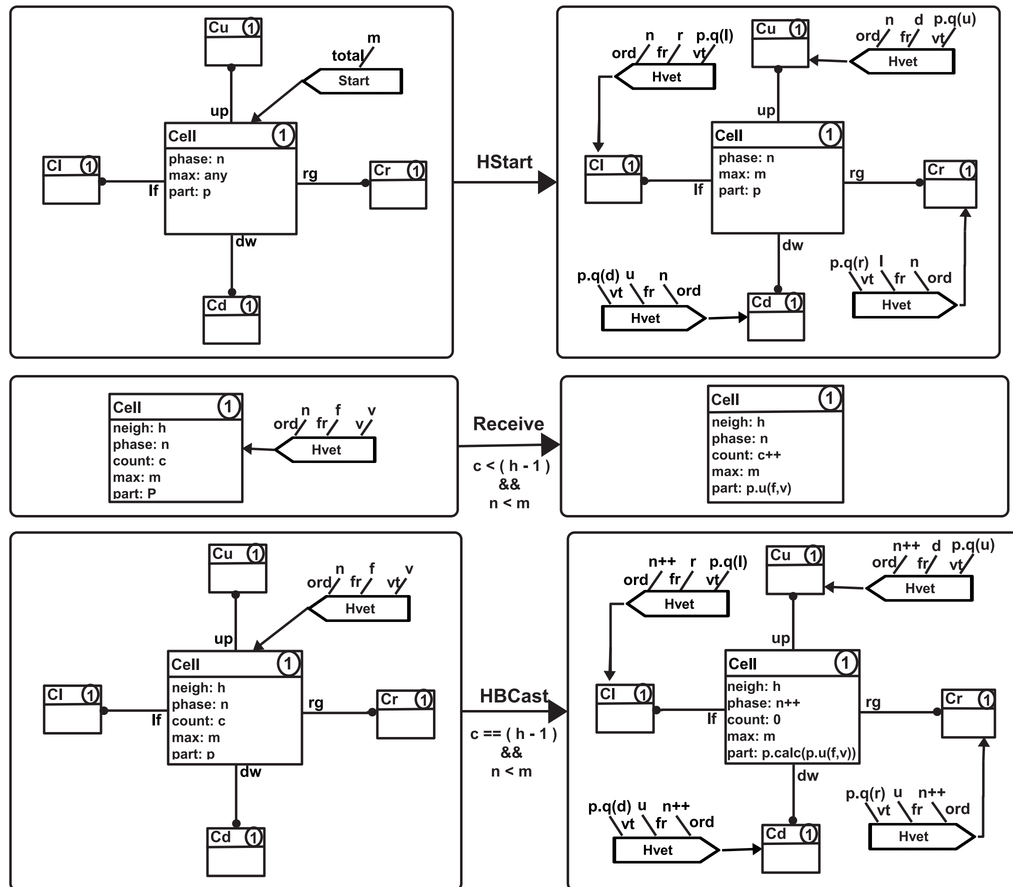


Figura 36: Regras para o Objeto *Cell*.

right e *l* para *left*).

Uma vez realizado *Cell_Start*, todos os objetos passam a consumir as mensagens enviadas pelos objetos adjacentes, considerando o passo de iteração (atributo *phase*) em sucessivas aplicações da regra *Cell_Receive*. Observe que existe uma comparação implícita entre o valor contido no atributo *phase* e o valor contido no parâmetro *ord* da mensagem *Hvet*, dado que ambos estão representados por n nas regras. O atributo *count* mantém o controle de quantas mensagens já foram processadas dentro de uma mesma fase. Ainda, no consumo de cada mensagem é atualizado o perfil de temperatura na borda da fração da peça, a partir da função $p.u(\ )$. O controle de quantas mensagens devem ser consumidas antes de realizar o cálculo e o *broadcast* do perfil é feito através da condição associada à regra, que considera o atributo *neigh*, onde fica setado o número de objetos adjacentes ao nodo em questão.

Quando a última mensagem necessária para completar o passo de iteração é consumida (regra *Cell_HBCast*), o objeto ajusta o último perfil e atualiza a temperatura de todos os pontos de

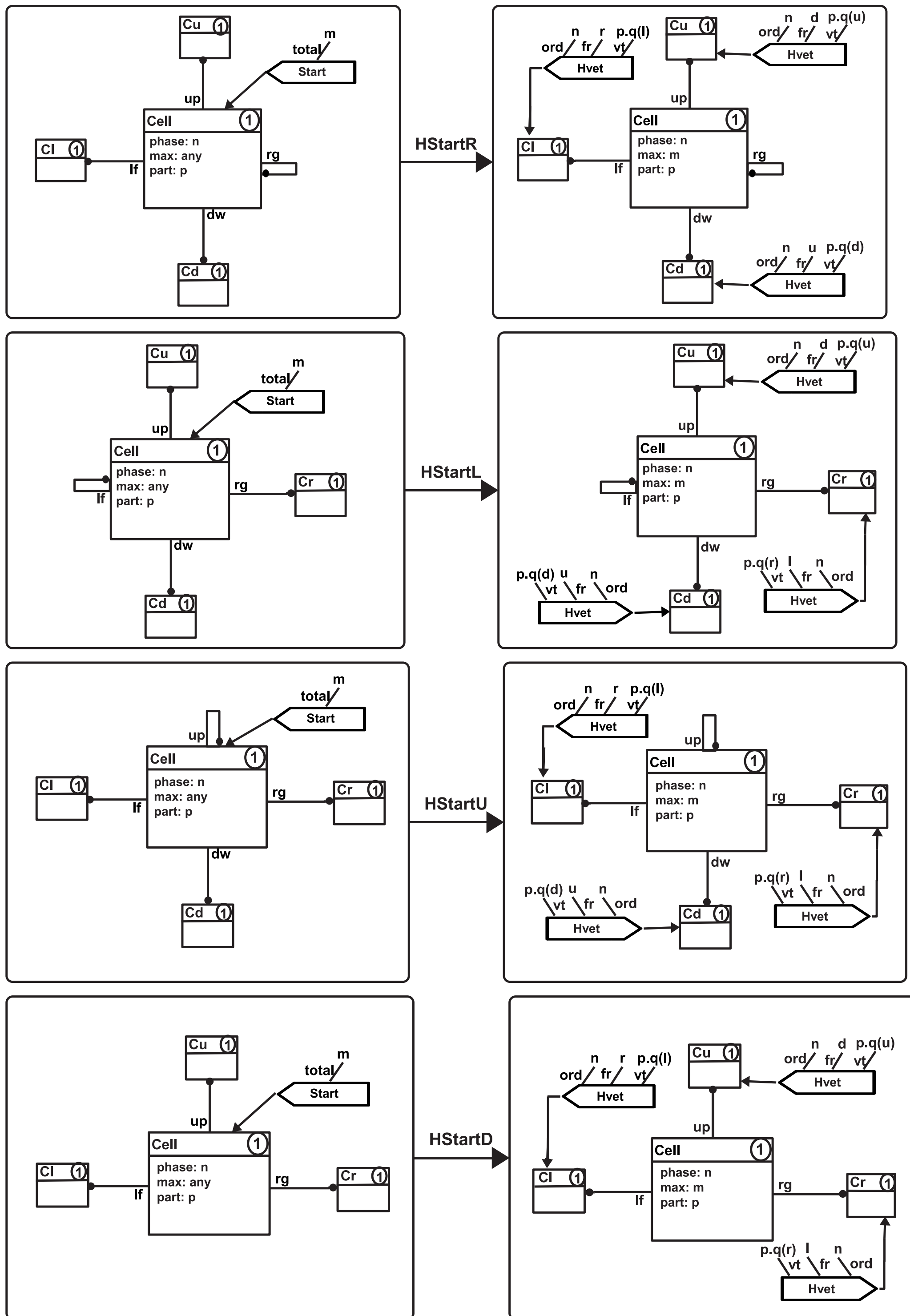


Figura 37: Regras *HStart* para Objeto *Cell* com três objetos adjacentes.

seu segmento, executando $p.hcalc(p.u(\ ))$. Definido o novo perfil de temperatura, o objeto envia novamente os novos valores para os processos adjacentes. Observe que o passo de execução

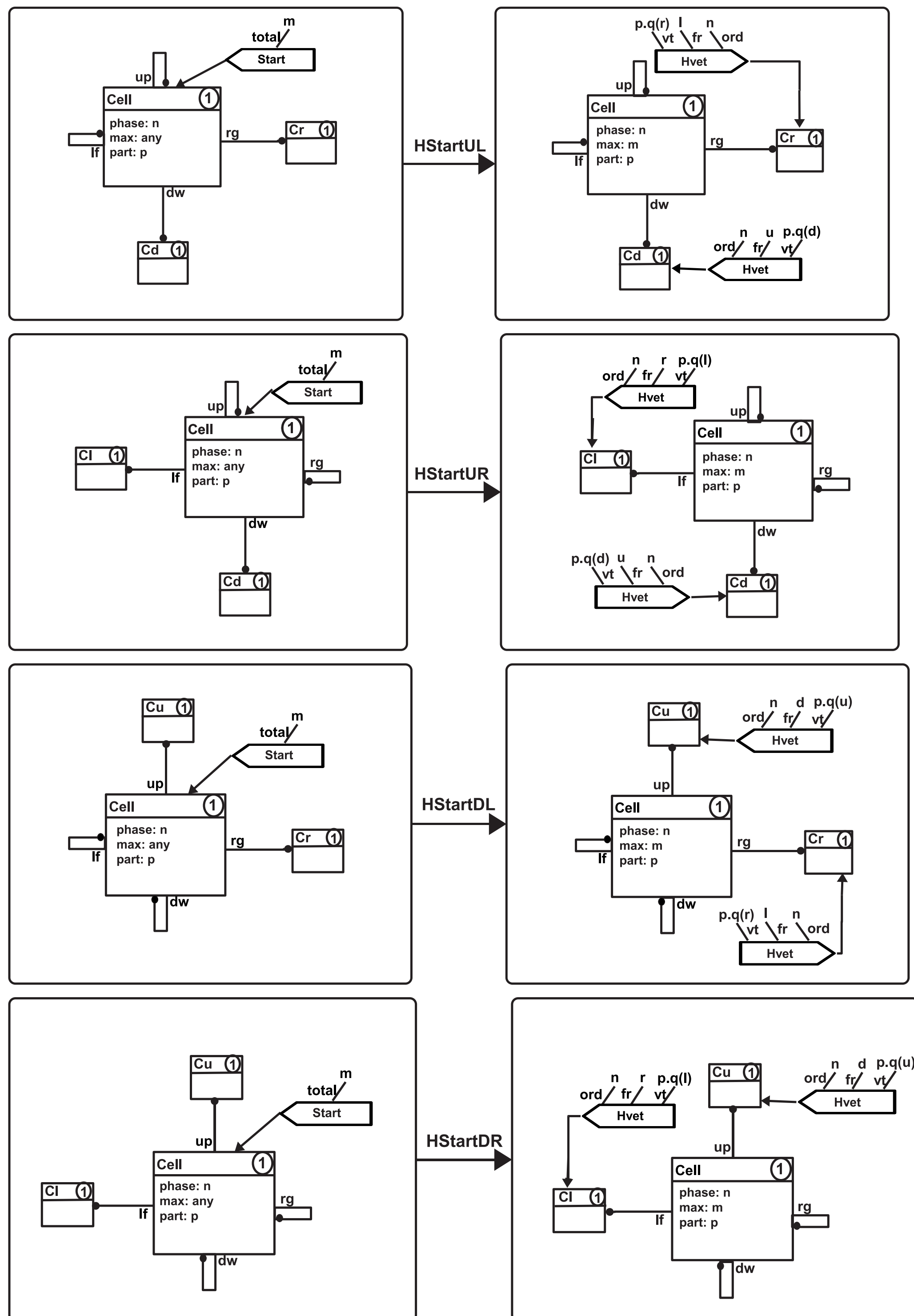


Figura 38: Regras *HStart* para Objeto *Cell* com dois objetos adjacentes.

(atributo *phase*) é incrementado, e *count* é setado como zero, indicando o início da próxima fase. O processo se repete até que sejam executados um número suficiente de passos.

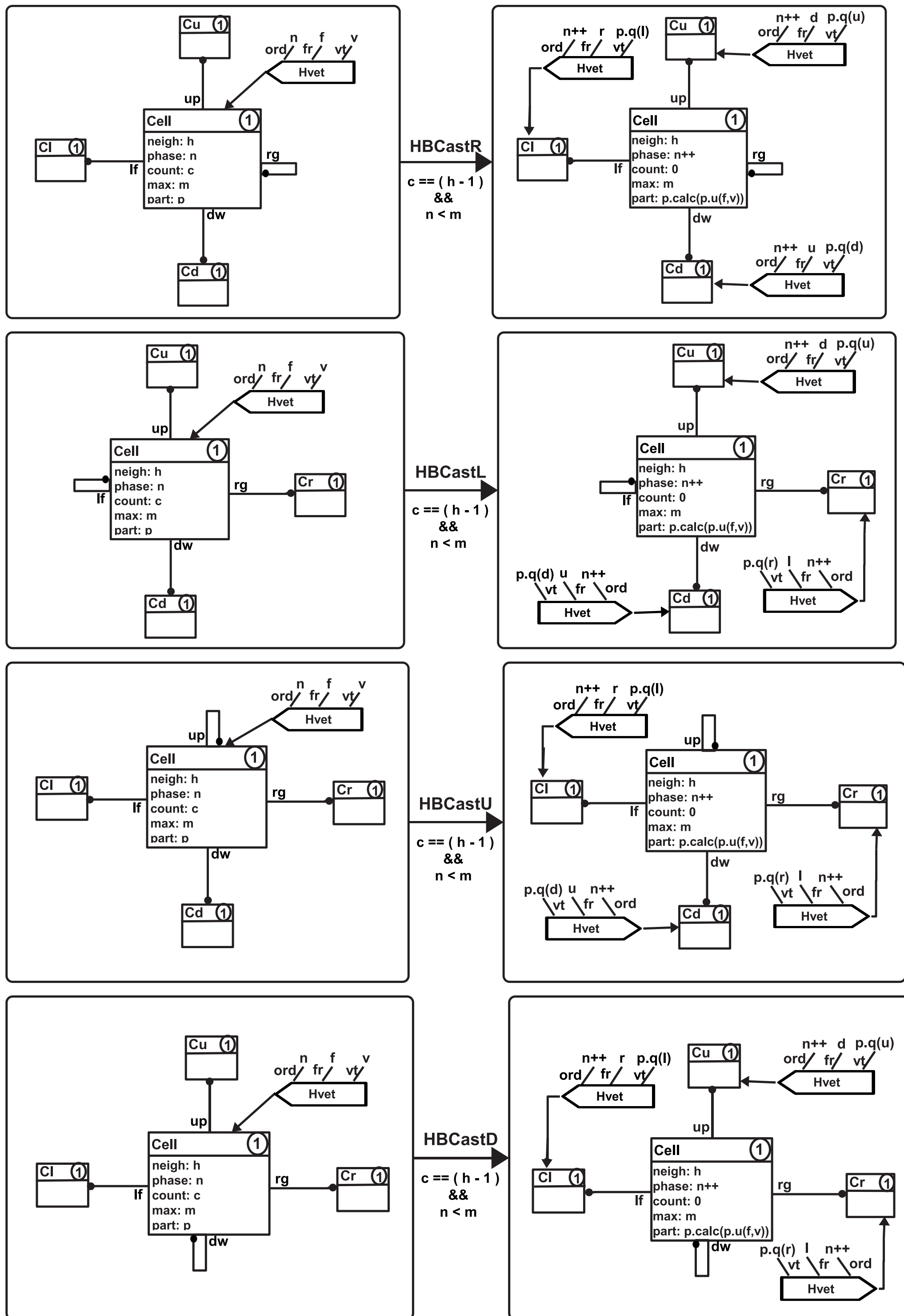


Figura 39: Regras *HBCast* para Objeto *Cell* com três objetos adjacentes.

Note que a existência de uma regra para a finalização do sistema não é estritamente necessária, porém dessa forma são deixadas mensagens ainda presentes no grafo de estado do sistema (*i.e.*,

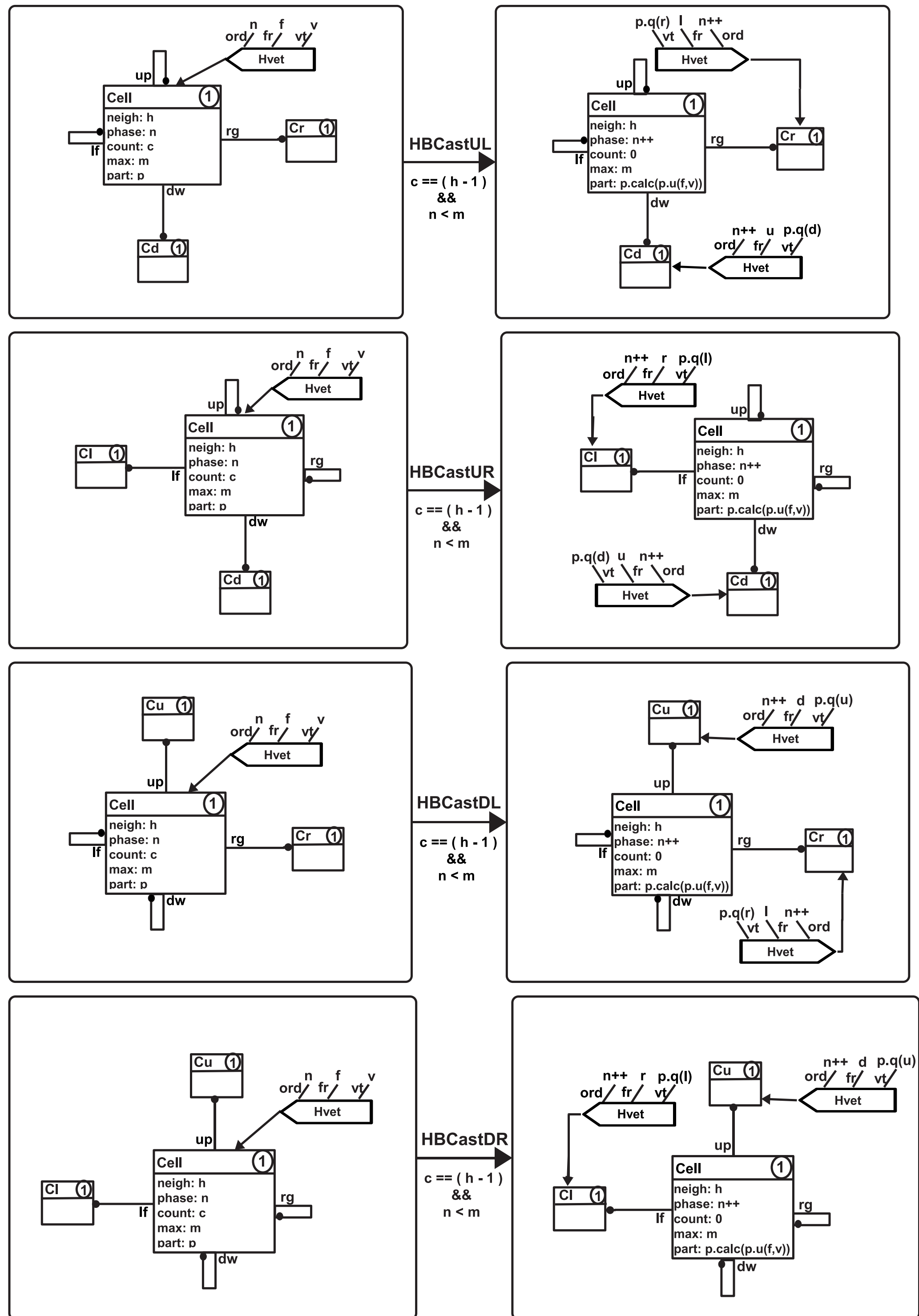


Figura 40: Regras *HBCast* para Objeto *Cell* com dois objetos adjacentes.

nas listas internas ao objeto). Isso poderia ser ajustado, entre outras possibilidades, adicionando–se uma nova regra que realize apenas o consumo das mensagens pendentes no grafo. Essa regra

representaria entretanto um atraso na finalização do sistema.

O objeto *Cell* apresentado nas Figuras 35 a 36 possui regras considerando quatro nodos adjacentes. Assim, as regras apresentadas não aplicam–se aos objetos responsáveis pelas partes laterais da peça metálica. Portanto, o modelo apresentado deve ser estendido, de maneira a possibilitar o *matching* para células presentes na borda da grade. As regras contidas nas Figura 37 e 38 adaptam a regra *Cell_HStart* para situações em que o objeto possua apenas três vizinhos e dois vizinhos, respectivamente. As Figuras 39 e 40, por sua vez, adequam a regra *Cell_HBCast* para os casos onde existam três e dois processos adjacentes.

As funções utilizadas no modelo são definida a partir do seguinte TAD:

Abstract Data Type *HeatCalc*:

type **vetor** is float[max];

type **side** is {u, d, l, r};

type **matrix** is float [col] [lin];

type **initCalc** is ***Struct***:

int col;

int lin;

int max;

vetor perf_up;

vetor perf_dw;

vetor perf_lf;

vetor perf_rg;

float temperatura;

EndStruct;

type **heat** is ***Struct***:

matrix points;

matrix ghostpoints;

EndStruct.

Operations:

HeatCalc new_HeatC (*initCalc* intc);

Inicializa o TAD. Aloca as matrizes *points* e *ghostpoints*, dimensionadas por *intc.col* e *intc.lin*. Define como *intc.max* o tamanho de *vetor*. Ajusta os perfis de temperatura nas bordas (superior, inferior, esquerda, direita)

conforme os valores passados em *intc.perf_up*, *intc.perf_dw*,*intc.perf_lf* e *intc.perf_rg*, respectivamente. Inicializa todos os pontos presentes na fração da peça com o valor passado para *intc.temperatura*.

matrix u (*side* s, *vetor* vet);

Atualiza o perfil de temperatura do lado *s* de *points* de para os valores indicados em *vet*. Assume *u* para a linha superior, *d* para a linha inferior, *l* para a coluna esquerda e *r* para a coluna direita.

vetor q (*side* s);

Retorna o perfil de temperatura presente no lado *s* de *points*. Assume *u* para a linha superior, *d* para a linha inferior, *l* para a coluna esquerda e *r* para a coluna direita.

matrix calc (*side* s, *vetor* vet);

Atualiza o perfil de temperatura de toda a matriz points. Após sua aplicação, tanto *points* quanto *ghostpoints* possuem o mesmo conjunto de elementos.

End *HeatCalc*.

A Figura 41 apresenta o grafo inicial para o modelo discutido. Observe que não foram explicitados os valores passados como parâmetros para *new_HeatC*. Dessa forma, o grafo inicial apresenta uma situação genérica para o arranjo das referências entre os objetos.

6.1.2 Outra Alternativa para o Particionamento do Problema

Na Seção anterior, o problema foi dividido entre os processos fracionando–se a grade em blocos, dividindo–se os blocos entre os participantes e realizando a comunicação entre cada processo e os demais adjacentes. Outra alternativa para o particionamento do problema é fracionar a grade em tiras verticais. Dessa forma, cada processo comunica–se no máximo com outros dois, sendo necessárias menos mensagens dentro de cada fase. Por outro lado, o conteúdo de cada mensagem fica maior. O modelo GGBO correspondente, considerando essas modificações é apresentado nas Figuras 42 a 45.

De maneira análoga ao modelo *HeatCell*, no modelo *HeatCollumn* é necessário definir regras específicas para possibilitar o *matching* para os processos responsáveis pelas bordas da placa.

A Figura 46 apresenta o grafo inicial para o novo modelo. Para melhorar a legibilidade, os mesmos processos e mensagens são mostrados na Figura 47, em uma distribuição diferente,

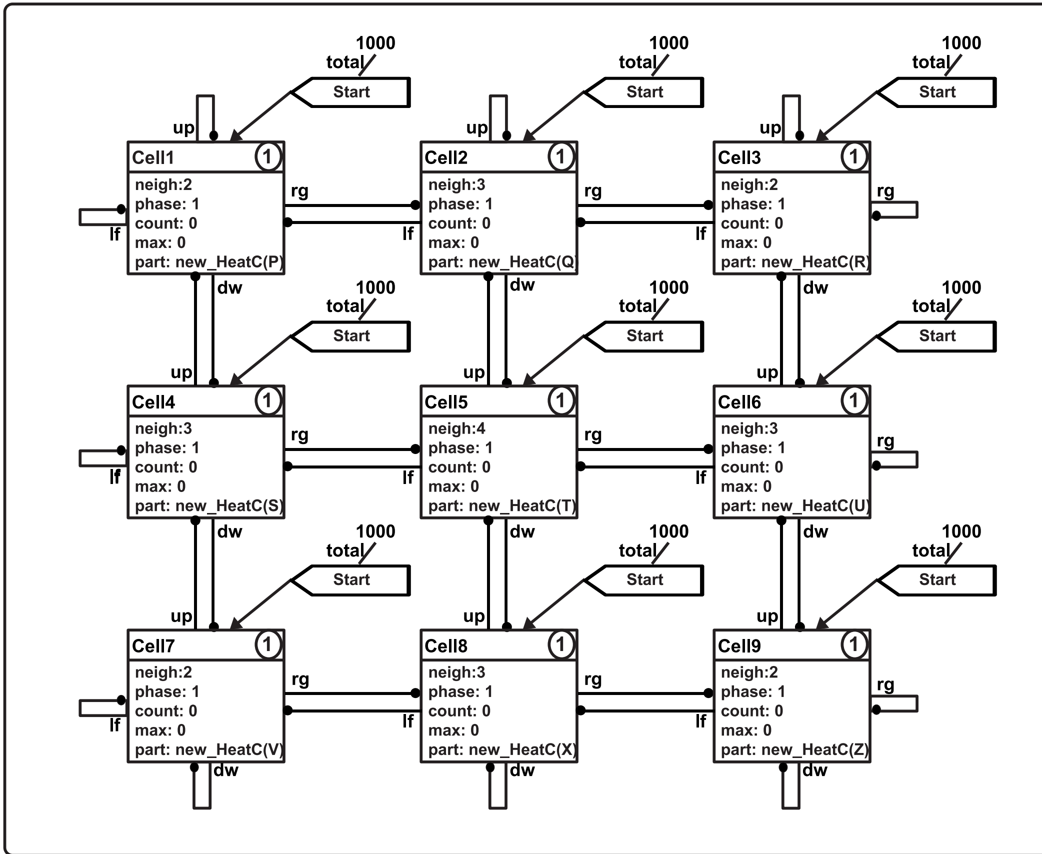


Figura 41: Modelo *HeatCell* – Grafo Inicial.

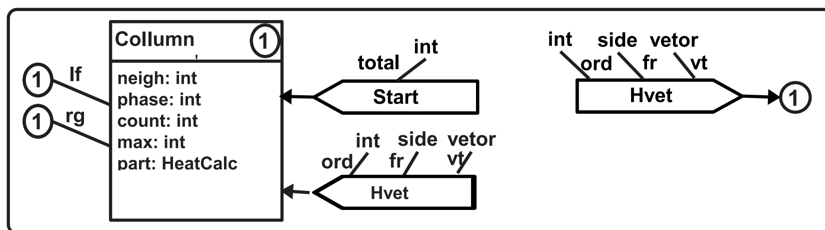


Figura 42: Grafo Tipo para Objeto *Collumn*.

porém definindo a mesma situação. Como no modelo anterior, não foram explicitados os valores passados como parâmetros para new_HeatC.

A tradução dos modelos *Heatcell* e *HeatCollumn* para C/MPI está sendo realizada manualmente, um trabalho ainda em andamento durante a elaboração desse volume. Até o presente momento, foram implementados protótipos para os modelos, sendo realizados testes iniciais utilizando matrizes de 10×10 pontos e a execução de até dez fases.

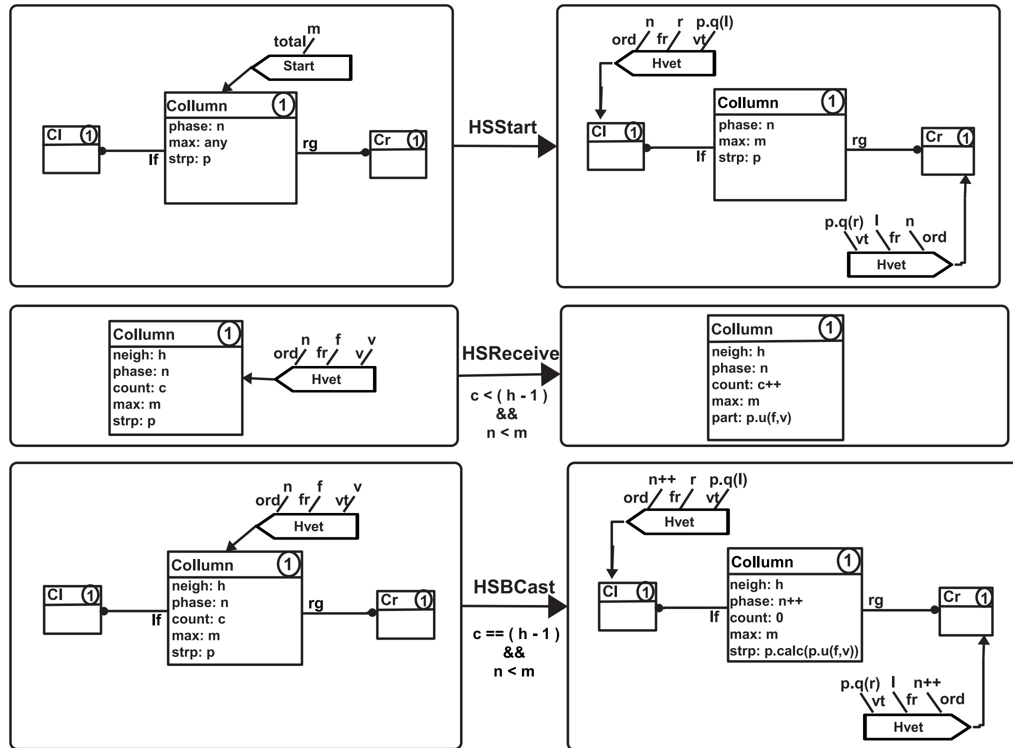


Figura 43: Regras para Objeto *Collumn*.

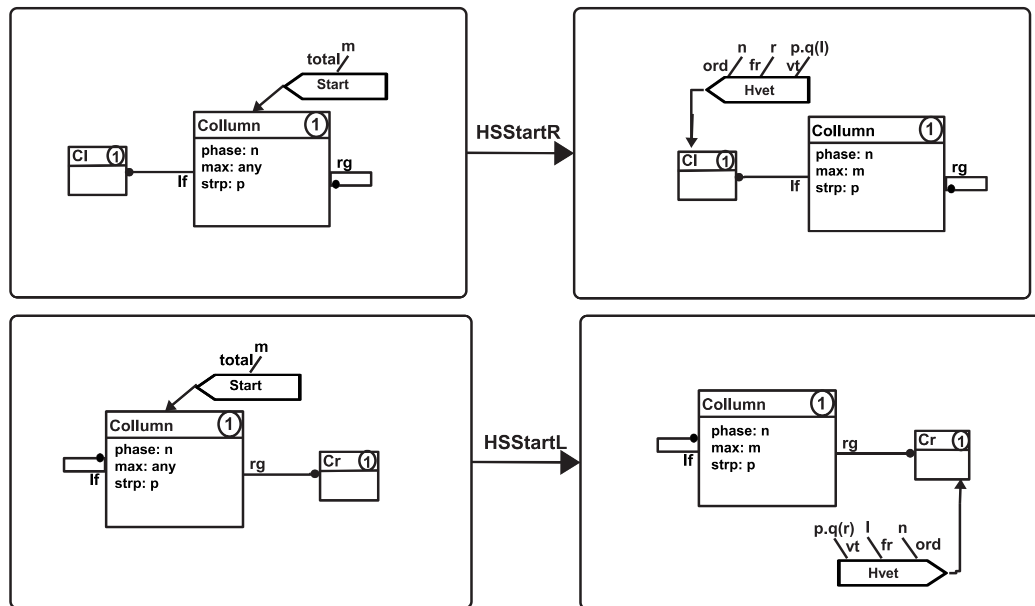


Figura 44: Regras *HSStart* para Objeto *Collumn* com um objeto adjacente.

A tabela 5 reproduz alguns dos valores obtidos para execuções distribuindo–se os nove processos entre dois nodos do *cluster* e utilizando rede *ethernet* para a comunicação. Os tempos

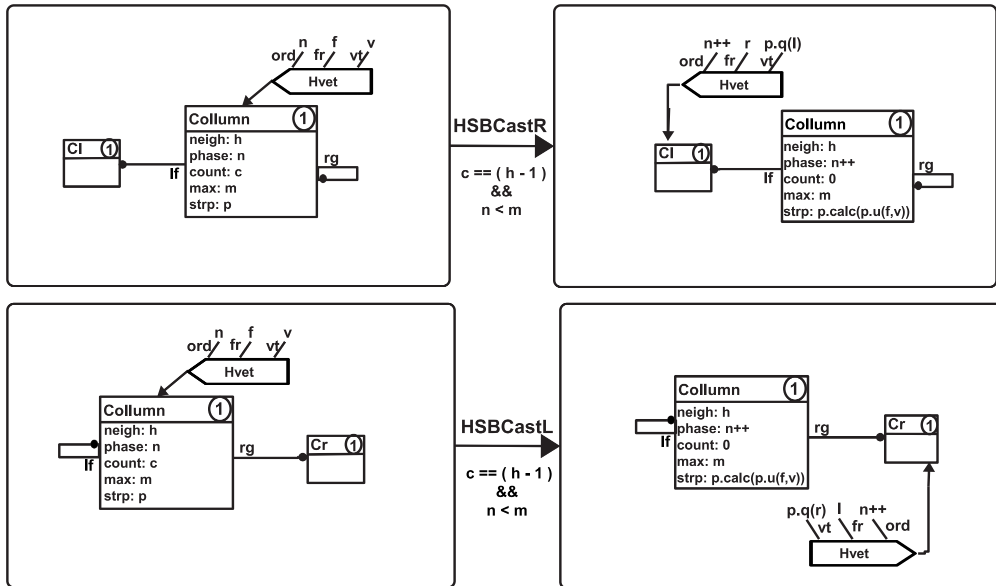


Figura 45: Regras *HSBcast* para Objeto *Collumn* com um objeto adjacente.

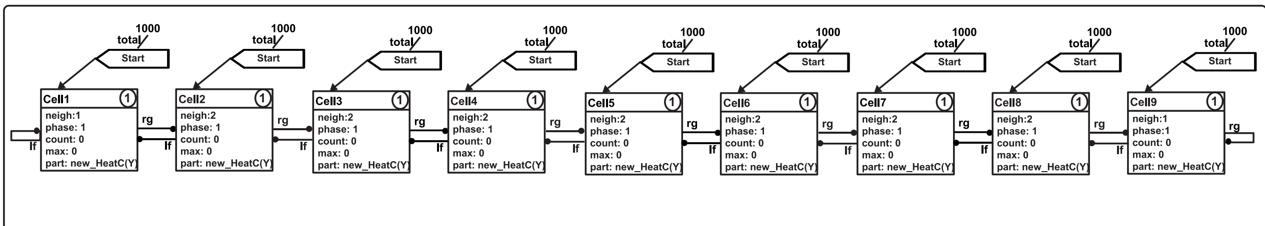


Figura 46: Modelo *HeatCollumn* – Grafo Inicial.

obtidos medem para cada processo o intervalo em segundos entre a inicialização da primeira *thread* e a execução da décima fase.

Para modelo *HeatCell* existe um grande número de regras envolvidas, o que por sua vez implica em um código fonte mais extenso que o dos estudos de casos anteriores, dificultando a depuração. Nesse sentido, a implementação de um gerador de código seria de grande valia, por realizar de maneira sistemática as modificações relativas a cada modelo, diminuindo assim os problemas enfrentados.

6.2 Verificando os modelos *HeatCell* e *HeatCollumn*

Em um modelo de fases paralelas deve ser mantido um determinado nível de sincronicidade durante a execução do programa por todos os participantes. Ainda, como trata–se de um

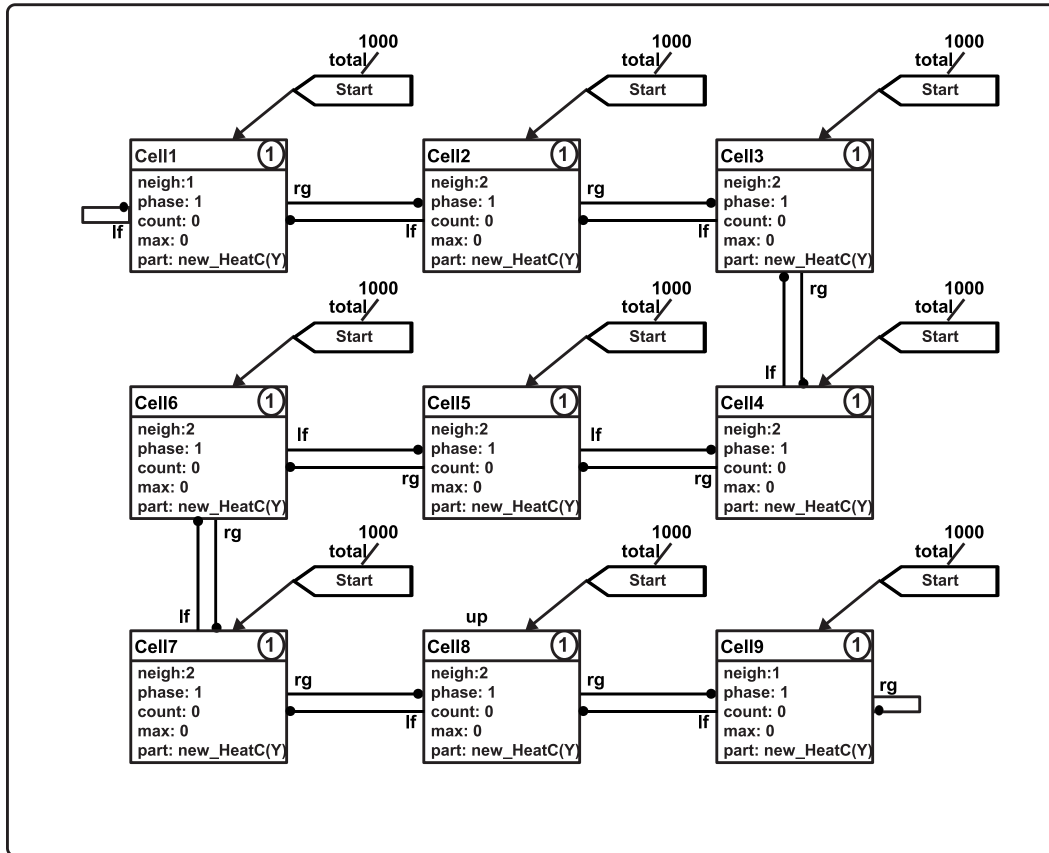


Figura 47: Modelo *HeatCollumn* – Grafo Inicial mais detalhado.

Tabela 5: Execução dos protótipos – Dois nodos alocados, dez fases.

	HeatCell		*HeatCollumn*	
Objeto	Teste 1	Teste 2	Teste 1	Teste 2
Cell1	2.287	7.831	0.189	2.467
Cell2	2.287	7.759	0.189	4.227
Cell3	1.968	7.760	0.190	4.227
Cell4	2.659	7.831	0.657	4.223
Cell5	2.655	7.830	0.184	4.225
Cell6	2.655	7.757	0.184	4.225
Cell7	2.653	7.829	0.180	4.221
Cell8	2.656	7.754	0.181	4.477
Cell9	2.649	7.747	0.175	4.221

algoritmo com grande comunicação e dependência entre os nodos, caso uma célula não possa prosseguir, a computação em todas as demais fica comprometida.

Inicialmente, pode–se considerar que não deve existir nenhuma diferença de fase entre os processos. Ou seja: nenhum processo "se desgarra" dos demais durante a execução. Essa característica pode ser obtida em um programa MPI utilizando–se uma chamada *barrier* no final de cada fase, que garante que todos os processos executem essa mesma função de maneira síncrona. Esse comportamento não pode ser obtido diretamente para o modelo proposto, uma vez que GGBO não disponibiliza nenhuma operação semelhante a um *barrier*, e todo o processamento é completamente assíncrono. Dessa forma, pode existir em um modelo GGBO uma situação em que processos encontrem–se em uma fase (após a execução da regra $Cell_HBCast$) e outros ainda na anterior (antes da execução de $Cell_HBCast$). Essa diferença no passo de execução pode ser acumulada conforme o número de células entre outras duas, de modo que a diferença entre duas células quaisquer suficientemente distantes seja ainda maior.

Para garantir que todos os processos computem os valores de maneira coerente, deve–se garantir que a diferença entre dois processos adjacentes nunca seja maior que uma fase. Ou seja, essa diferença (observável no valor do atributo *phase* presente no objeto) nunca pode ser maior do que uma unidade para células vizinhas.

Outra característica importante a ser certificada é a terminação do algoritmo. Uma vez que o processamento no modelo proposto é feito atrelado a um número máximo de fases (setado no grafo inicial através do parâmetro *total* das mensagens *Start*), tal propriedade fica garantida desde que todos os processos possam executar o número suficiente de passos.

Dessa forma, as propriedades fundamentais em tal estratégia são :

(i) Sincronia: A diferença de fase entre dois processos adjacentes nunca difere em mais de uma unidade;

(ii) Terminação: Eventualmente o modelo finaliza.

Para a verificação do problema, será utilizada uma peça composta por quatro pontos. Na verificação do modelo *HeatCell*, foram utilizados quatro objetos *Cell* para compor a grade. Cada objeto fica responsável por apenas um ponto da grade, e o modelo é configurado pelo grafo inicial de maneira a executar três fases. Para o modelo *HeatCollumn*, foram utilizados quatro objetos *Collumn*, cada um deles responsável por uma coluna de dois pontos, e o modelo executa também três fases[3].

[3]Inicialmente tentou–se realizar a verificação de *HeatCell* para uma grade de nove pontos e nove processos,

Observe que a verificação é realizada apenas considerando aspectos relativos à troca de mensagens entre os participantes. As operações oferecidas pelo TAD utilizado foram modeladas em PROMELA manualmente. A verificação das propriedades propostas foi feita da seguinte forma:

- **(i) Sincronia:** Para verificar a sincronia no modelo *HeatCell*, é declarado um conjunto de variáveis globais $Cell_Phase_i$ (sendo $i = 1, 2, 3, 4$), cada uma associada a um processo GGBO, permitindo assim diferenciar os objetos entre si. Para essa variável é copiado o valor do atributo *phase* presente no processo associado, imediatamente após a aplicação de cada regra. Assim, inspecionando–se o valor dessas variáveis é possível ter uma visão global da fase em que cada objeto se encontra.

  Ainda, foram declaradas mais quatro variáveis chamadas $Diff_Phase_i_j$ para monitorar a diferença entre os valores armazenados em $Cell_Phase_i$ e $Cell_Phase_j$, para $i, j = (1, 2), (1, 3), (2, 4), (3, 4)$. A partir dessas variáveis torna–se possível monitorar a diferença de fase entre dois processos vizinhos quaisquer. Essas variáveis foram incluídas no modelo imediatamente após a aplicação de uma regra e a atualização do conteúdo das variáveis $Cell_Phase$, de maneira que sejam executadas as seguintes operações:

  $$Diff_Phase_1_2 = (Cell_Phase_1 - Cell_Phase_2);$$
  $$Diff_Phase_1_3 = (Cell_Phase_1 - Cell_Phase_3);$$
  $$Diff_Phase_2_4 = (Cell_Phase_2 - Cell_Phase_4);$$
  $$Diff_Phase_3_4 = (Cell_Phase_3 - Cell_Phase_4);$$

  Dessa forma, a diferença de fase entre os objetos vizinhos da grade fica registrada nas variáveis globais, permitindo a definição das seguintes proposições atômicas:

  $$\#define\ dif_12\ (-1 \leq Diff_Phase_1_2 \leq 1).$$
  $$\#define\ dif_13\ (-1 \leq Diff_Phase_1_3 \leq 1).$$
  $$\#define\ dif_24\ (-1 \leq Diff_Phase_2_4 \leq 1).$$
  $$\#define\ dif_34\ (-1 \leq Diff_Phase_3_4 \leq 1).$$

  Para que as proposições sejam verdadeiras, é necessário que a diferença de fase entre os processos vizinhos mantenha–se dentro do intervalo $[-1, 1]$, indicando que não existe diferença (quando $Diff_Phase == 0$), ou então existe uma diferença de apenas uma

mas o grande número de intercalações possíveis para essa configuração impossibilitou o tratamento do modelo.

fase. Caso a diferença de fase seja maior que uma unidade, o conteúdo de $Diff_Phase$ é alterado para um valor fora do intervalo fixado, tornando falsa a proposição.

A partir dessas definições, verificou–se a seguinte sentença para o modelo:

[ ] ($diff_12$ && $diff_13$ && $diff_24$ && $diff_34$)

Após a verificação obteve–se como resultado que a propriedade é verdadeira para o modelo proposto. Assim, o modelo sempre computa os pontos de maneira coerente, não apresentando um deslizamento entre as fases dos participantes que possa causar a geração de dados inconsistentes.

O modelo *HeatCollumn* apresenta o mesmo número de processos porém uma topologia diferente. Assim, a associação de variáveis $Cell_Phase$ permanece a mesma para os processos, porém foram declaradas as seguintes variáveis globais para monitorar a diferença de fase:

$$Diff_Phase_1_2 = (Cell_Phase_1 - Cell_Phase_2);$$
$$Diff_Phase_2_3 = (Cell_Phase_2 - Cell_Phase_3);$$
$$Diff_Phase_3_4 = (Cell_Phase_3 - Cell_Phase_4);$$

Sobre essas variáveis, as seguintes proposições atômicas foram elaboradas:

$$\#define\ dif_12\ (-1 \leq Diff_Phase_1_2 \leq 1).$$
$$\#define\ dif_23\ (-1 \leq Diff_Phase_2_3 \leq 1).$$
$$\#define\ dif_34\ (-1 \leq Diff_Phase_3_4 \leq 1).$$

E foi aplicada ao modelo a fórmula:

[ ] ($diff_12$ && $diff_23$ && $diff_34$)

Da mesma forma que para *HeatCell*, a sentença verificada para o modelo *HeatCollumn* é verdadeira.

- **(ii) Terminação:** Para verificar a terminação, a partir das mesmas variáveis globais incluídas para verificar a propriedade anterior, foram definidas as seguintes proposições atômicas:

$$\#define\ Cell_Finish_1\ (Cell_Phase_1 == m)$$
$$\#define\ Cell_Finish_2\ (Cell_Phase_2 == m)$$

$\ldots$

$\#define\ Cell_Finish_i\ (Cell_Phase_i == m)$

para m igual ao parâmetro *total* da mensagem *Start* incluída no grafo inicial do sistema, e i variando de 1 a 9. Essa configuração corresponde ao estado final previsto para os objetos dentro do modelo (número de fases executadas igual ao valor estipulado).

A partir dessas proposições, aplicou–se a seguinte sentença aos modelos *HeatCell* e *HeatCollumn*:

$<> (Cell_Finish_1\ \&\&\ Cell_Finish_2\ \&\& \ldots \&\&\ Cell_Finish_i)$

A verificação dessa sentença resultou em verdadeiro, indicando que todos os processos podem executar o número indicado de fases para ambos os modelos.

Neste Capítulo foi exemplificada a utilização de GGBO para a modelagem de uma aplicação paralela real, sendo realizada a verificação e a prova de propriedades para os modelos propostos.

Apoiando–se nos resultados apresentados no Capítulo anterior, onde foi provado que a semântica do código gerado é a mesma do modelo GGBO, pode–se assumir que a semântica do programa gerado a partir da tradução dos modelos apresentados também respeite as propriedades verificadas.

Ainda, desde que a implementação do TAD utilizado no modelo respeite às condições impostas para a sua utilização nesse trabalho, e considerando–se que as propriedades enunciadas dizem respeito somente à troca de mensagens entre os participantes, pode–se manter a correspondência entre o modelo verificado e o programa gerado pela tradução proposta.

7 Conclusão

O presente trabalho apresenta uma estrutura para a tradução de modelos GGBO em código fonte C/MPI, possibilitando a sua utilização como uma linguagem de análise e projeto de aplicações paralelas.

Um dos objetivos desse trabalho é a definição de estruturas de dados e algoritmos que ofereçam a semântica do formalismo de modelagem e ainda permitam a sua geração de maneira automatizável. Dessa forma, as estruturas de programação foram propostas visando reproduzir o comportamento de um processo GGBO genérico. A partir dessa estrutura básica – um *template* – o comportamento dos objetos previstos no modelo sendo traduzido é inserido através da inclusão de modificações específicas referentes ao modelo GGBO sendo tratado. Essas modificações são feitas em quatro pontos: no mecanismo de *matching*, no código executado pelas regras, no processo INIT e nas funções de empacotamento de mensagens. O mecanismo de *matching* é modificado visando reproduzir as condições associadas à execução de uma regra, sendo consideradas também comparações definidas do lado esquerdo da regra. No código relativo às ações desempenhadas pela aplicação de cada regra são refletidas as modificações presentes no lado direito da regra. No código relativo ao processo INIT, que reproduz o grafo inicial do modelo, a ordem do envio de mensagens é definida de maneira aleatória. Também são realizadas modificações relativos ao empacotamento e desempacotamento de dados para as mensagens sendo enviadas.

O *template* proposto foi verificado, visando garantir que o algoritmo utilizado reproduz o comportamento esperado de um objeto GGBO. Essa tradução foi feita manualmente, mantendo-se ao máximo a semelhança entre o código C gerado e o código PROMELA correspondente. Cada uma das *threads* que compõem um processo GGBO foi convertida em um processo PROMELA, dentro dos quais os procedimentos, funções e estruturas de dados foram também traduzidos. A partir da semelhança mantida, pode-se assumir que o comportamento verificado para o modelo

do *template* é mantido pelo código gerado. É importante diferenciar a tradução para PROMELA realizada para o *template* da tradução realizada automaticamente proposta em [62]. Enquanto a primeira foi realizada especificamente para o código C gerado, a segunda apresenta um método genérico para a geração de um modelo para verificação a partir de um modelo GGBO. O presente trabalho usa os resultados obtidos em [62] para provar as propriedades para os modelos GGBO Pi, HeatCell e HeatCollumn.

Apesar de já oferecer resultados que possibilitem a geração e execução de programas a partir de modelos GGBO, existe ainda a necessidade da implementação e incorporação do algoritmo de detecção de parada proposto para a tradução, assim como verificação do modelo correspondente ao código resultante dessa modificação. Após tal implementação torna-se interessante a reelaboração do código relativo aos modelos já estudados e verificação do *template* resultante, possibilitando assim uma análise do impacto de sua utilização.

A partir do esquema de tradução apresentado e análise dos resultados obtidos nesse trabalho, pode-se identificar alguns focos principais de esforços para a melhoria dos algoritmos propostos. A principal melhoria a ser aplicada no processo de tradução envolve a geração de tipos de mensagens diferentes, com um número mínimo de atributos em cada mensagem, assim como a correspondente modificação nos algoritmos para empacotamento e desempacotamento de dados. Essas mudanças podem trazer resultados interessantes ao desempenho do código gerado. Tal modificação se refletiria também nos algoritmos de gerenciamento de listas implementados, possibilitando a utilização de menos atributos em cada elo e um melhor aproveitamento da memória.

Ainda, podem ser realizados estudos visando otimizações no mecanismo de *matching*, que no momento é bastante genérico. Uma possível otimização nesse sentido seria a utilização de uma *hash table* para agilizar a localização do par selecionado dentro de $L_{matches}$, diminuindo o impacto sofrido quando existirem muitas possibilidades de escolha mensagem-regra.

Existem ainda trabalhos a serem conduzidos para oferecer suporte à criação dinâmica de processos, que podem tanto envolver a adaptação da presente proposta de tradução para o padrão MPI-2, ou mesmo buscar uma solução utilizando o padrão atual.

Além das modificações discutidas, outro importante trabalho futuro é a implementação do protótipo de um gerador de código baseado nas estruturas propostas, a ser incorporado à ferramenta de edição de modelos GGBO existente.

A elaboração desse trabalho também oferece como resultados argumentos que motivam a

realização de análise quanto à extensão da linguagem de modelagem utilizada. A partir do estudo de caso desenvolvido no Capítulo 6, pode-se observar que determinadas características presentes no modelo podem exigir um grande número de regras para possibilitar um *matching* adequado dentro do grafo de estado. No exemplo, a partir de duas regras elaboradas prevendo quatro processos vizinhos, outras dezesseis foram derivadas visando adequar-se a situações com três e dois processos adjacentes. Ainda, na Seção 4.2, quanto analisando o comportamento para envio de rajadas de mensagens, por questões de espaço, foi apresentada somente uma regra relativa a uma rajada de duas mensagens, deixando-se comentada a existência das demais regras responsáveis pela geração dos dados apresentados. Nesse sentido, modificações na linguagem de modelagem visando aumentar a expressividade poderiam facilitar a sua utilização, permitindo abranger um número maior de situações com menos regras.

Referências Bibliográficas

[1] S. Ahmed, N. Carriero, and D. Gelernter. “A Program Building Tool for Parallel Applications”. In: *DIMACS Workshop on Specification of Parallel Algorithms*. New Haven, CT, USA: Oxford University Press, 1994, pp. 161–178.

[2] S. Ahuja, N. J. Carriero, D. H. Gelernter, and V. Krishnaswamy. “Matching language and hardware for parallel computation in the linda machine”. *IEEE Transactions on Computers*, vol. 37-8, 1988, pp. 921–929.

[3] C. Anglano, R. Wolski, J. Schopf, and F. Berman. “Developing heterogeneous applications using zoom and hence”. In: *4th. Heterogeneous Computing Workshop (HCW'95)*. Santa Barbara, CA, USA: IEEE Computer Society, 1995, pp. 1–10.

[4] J. Anvik, J. Schae, D. Szafron, and K. Tan. “Why not use a pattern-based parallel programming system?”. In: *International Conference on Parallel and Distributed Computing (EuroPar'03)*. Klagenfurt, Austria: Springer-Verlag, 2003, pp. 81–86.

[5] J. S. Auerbach, A. P. Goldberg, G. S. Goldszmidt, A. S. Gopal, M. T. Kennedy, J. R. Rao, and J. R. Russell. “Concert/c: A language for distributed programming”. In: *USENIX Winter 1994 Technical Conference*. San Francisco, CA, USA: USENIX Association, 1994, pp. 79–96.

[6] L. Augustsson and T. Johnsson. “Parallel Graph Reduction with the $\langle v, G \rangle$-machine”. In: *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. London, UK: ACM Press, 1989, pp. 202–213.

[7] A. Baguelin, J. Dongarra, G. Giest, R. Manchek, and V. Sunderam. “Graphical development tools for network-based concurrent supercomputing”. In: *1991 ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. Albuquerque, NM, USA: ACM Press, 1991, pp. 435–444.

[8] H. E. Bal. "Orca: a language for distributed programming". *ACM SIGPLAN Notices*, vol. 25-5, 1990, pp. 17–24.

[9] K. Birman and R. Cooper. "The isis project: real experience with a fault tolerant programming system". *ACM Operating Systems Review*, vol. 25-2, 1991, pp. 103–107.

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. "Myrinet: A gigabit-per-second local area network". *IEEE Micro*, vol. 15-1, 1995, pp. 29–36.

[11] J. C. Browne. "Code visual parallel programming system". Disponível em: http://www.cs.utexas.edu/users/code. Último acesso em: Abril de 2007.

[12] J. C. Browne, M. Azam, and S. Sobek. "Code: A unified approach to parallel programming". *IEEE Software*, vol. 6-4, 1989, pp. 10–18.

[13] J. C. Browne, J. Dongarra, S. I. Hyder, K. M., and P. Newton. "Visual programming and parallel computing". University of Tennessee. Knoxville, TN, USA, Tech. Rep. UT-CS-94-229, 1994.

[14] M. Cannataro, S. D. Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia. "A parallel cellular automata environment on multicomputers for computational science". *Parallel Computing*, vol. 21-5, 1995, pp. 803–823.

[15] N. Carriero and D. Gelernter. "Applications experience with linda". In: *ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems (PPEALS'88)*. New Haven, CT, USA: ACM Press, 1988, pp. 173–187.

[16] M. Chechik and D. O. Paun. "Events in property patterns". In: *5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. Trento, Italy: Springer-Verlag, 1999, pp. 154–167.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. "Model checking". Cambridge, MA, USA: MIT Press, 1999, 314p.

[18] B. Copstein, M. C. Móra, and L. Ribeiro. "An environment for formal modeling and simulation of control systems". In: *33rd Annual Simulation Symposium*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 74–82.

[19] R. Couturier and B. Coutourier. "A compiler for parallel unity programs using openmp". In: *Parallel and Distributed Processing Techniques and Applications (PDPTA'99).* Las Vegas, NV, USA: CSREA Press, 1999, pp. 1992–1998.

[20] E. W. Dijkstra. "Structured programming". Disponível em: http://www.cs.utexas.edu/users/EWD/indexBibTeX.html. Último acesso em: Abril de 2007.

[21] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. "Derivation of a termination detection algorithm for distributed computations.". *Information Processing Letters*, vol. 16-5, 1983, pp. 217–219.

[22] E. W. D. Dijkstra and C. S. Scholten. "Termination detection for diffusing computation". *Information Processing Letters*, vol. 11-1, 1980, pp. 1–4.

[23] F. L. Dotti, L. M. Duarte, L. Foss, L. Ribeiro, D. Russi, and O. M. Santos. "An environment for the development of concurrent object-based applications". In: *2nd International Workshop on Graph-Based Tools (GraBaTs'04).* Rome, Italy: Elsevier Science Publishers B. V., 2004, pp. 3–13.

[24] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos. "Especificação e verificação formal de sistemas distribuídos". In: *17º Simpósio Brasileiro de Engenharia de Software (SBES'03).* Manaus, AM, Brasil: UFAM - DCC, 2003, pp. 225–240.

[25] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos. "Verification of object-based distributed systems". In: *6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'03).* Paris, France: Springer-Verlag, 2003, pp. 261–275.

[26] F. L. Dotti and L. Ribeiro. "Specification of mobile code systems using graph grammars". In: *4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'00).* Stanford, CA, USA: Kluwer, 2000, pp. 45–63.

[27] F. L. Dotti, L. Ribeiro, O. M. Santos, and F. Pasini. "Verifying object-based graph grammars: An assume-guarantee approach". *Software and Systems Modeling (SoSyM)*, vol. 5-3, 2006, pp. 289–311.

[28] D. Duke, T. Green, and J. Pasko. "Research toward a heterogeneous networked computing cluster: The distributed queuing system version 3.0". Supercomputer Compu-

tations Research Institute, Florida State University. Tallahassee, FL, USA, Tech. Rep. DOE/ER/25147–T1, 1996.

[29] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. “Property specification patterns for finite-state verification”. In: *2nd Workshop on Formal Methods in Software Practice (FMSP’98)*. Clearwater Beach, FL, USA: ACM Press, 1998, pp. 7–15.

[30] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. “Patterns in property specifications for finite-state verification”. In: *21st International Conference on Software Engineering (ICSE’99)*. Los Angeles, CA, USA: IEEE Computer Society Press, 1999, pp. 411–420.

[31] M. B. Dwyer and C. S. Pasareanu. “Filter-based model checking of partial systems”. In: *6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT’98/FSE-6)*. Lake Buena Vista, FL, USA: ACM Press, 1998, pp. 189–202.

[32] J. D. Eckart. “Cellang 2.0: language reference manual”. *SIGPLAN Notices*, vol. 27-8, 1992, pp. 107–112.

[33] H. Ehrig. “Introduction to the algebraic theory of graph grammars (a survey)”. In: *International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*. Bad Honnef, Germany: Springer-Verlag, 1979, pp. 1–69.

[34] R. Finkel and U. Manber. “Dib - a distributed implementation of backtracking”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9-2, 1987, pp. 235–256.

[35] The MPI Forum. “Mpi-2: Extensions to the message-passing interface”. Disponível em: http://www.mpi-forum.org/docs/mpi-20.ps. Último acesso em: Abril de 2007.

[36] N. Francez. “Distributed termination”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2-1, 1980, pp. 42–55.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. “Design patterns: elements of reusable object-oriented software”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, 395p.

[38] C. Hochberger and R. Hoffmann. “Cdl – a language for cellular processing”. In: *2nd International Conference on Massively Parallel Computing Systems (MPCS’96)*. Ischia, Italy: IEEE Press, 1996, pp. 41–47.

[39] G. J. Holzmann. "The model checker SPIN". *IEEE Transactions on Software Engineering*, vol. 23-5, 1997, pp. 279–295.

[40] P. Hudak and J. H. Fasel. "A gentle introduction to haskell". *SIGPLAN Notices*, vol. 27-5, 1992, pp. 1–52.

[41] K. Hwang. "Advanced computer architecture: Parallelism, scalability programmability". New York, NY, USA: McGraw-Hill Higher Education, 1993, 771p.

[42] M. Ibel, K. Schauser, C. Scheiman, and M. Weis. "High-performance cluster computing using scalable coherent interface". In: *7th International Workshop on SCI-based High-Performance Low-Cost Computing*. Santa Clara, CA, USA: [s.n.], 1997, pp. 45–54.

[43] P. Iglinski, N. Kazouris, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron, and D. Woloschuk. "Using a template-based parallel programming environment to eliminate errors". In: *High Performance Computing Symposium (HPC'96)*. Ottawa, Canada: Carleton University Press, 1996, pp. 1–23.

[44] T. Johnsson. "Efficient compilation of lazy evaluation". In: *SIGPLAN Symposium on Compiler Construction*. Montreal, Canada: ACM Press, 1984, pp. 58–69.

[45] L. V. Kalé. "The chare kernel parallel programming language and system". In: *International Conference on Parallel Processing (ICPP'90)*. Urbana-Champaign, IL, USA: Pennsylvania State University Press, 1990, pp. 17–25.

[46] J. P. Katoen. "Concepts, algorithms, and tools for model checking". Arbeitsberichte der Informatik, Inst. für Mathematische Maschinen und Datenverarbeitung. Friedrich-Alexander-Universitaet, Erlangen-Nuernberg, Tech. Rep., 1999.

[47] A. Keller and A. Reinefeld. "Anatomy of a resource management system for hpc clusters". *Annual Review of Scalable Computing*, vol. 3-1, 2001, pp. 1–31.

[48] O. Kupferman and M. Y. Vardi. "Module checking". In: *8th International Conference on Computer Aided Verification (CAV'96)*. New Brunswick, NJ, USA: Springer-Verlag, 1996, pp. 75–86.

[49] F. L.Dotti, L. M. Duarte, and L. R. B. Copstein. "Simulation of mobile applications". In: *Communication Networks and Distributed Systems Modeling and Simulation Conference.*

San Antonio, TX, USA: The Society for Modeling and Simulation International, 2002, pp. 261–267.

[50] D. Lea and D. Lea. "Concurrent programming in java: Design principles and patterns". Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996, 339p.

[51] A. B. Loreto, L. Ribeiro, and L. Toscani. "Complexity analysis of reactive graph grammars". *Revista de Informática Teórica e Aplicada - UFRGS*, vol. 7-1, 2000, pp. 109–128.

[52] N. R. Mahapatra and S. Dutt. "An efficient delay-optimal distributed termination detection algorithm". Department of Computer Science and Engineering, University of Buffalo, The State University of New York. Buffalo, NY, USA, Tech. Rep. 2001-16, 2001.

[53] T. A. Marsland, T. Breitkreutz, and S. Sutphen. "A network multi-processor for experiments in parallelism". *Concurrency: Practice and Experience*, vol. 3-3, 1991, pp. 203–219.

[54] O. A. McBryan. "An overview of message passing environments". *Parallel Computing*, vol. 20-4, 1994, pp. 417–443.

[55] D. A. Mundie and D. A. Fisher. "Parallel processing in ada". *Computer*, vol. 19-8, 1986, pp. 20–25.

[56] M. A. S. Netto and C. A. F. D. Rose. "Crono: A configurable and easy to maintain resource manager optimized for small and mid-size gnu/linux cluster". In: *32nd International Conference on Parallel Processing (ICPP'03)*. Kaohsiung, Taiwan: IEEE Computer Society, 2003, pp. 555–562.

[57] P. Newton and J. C. Browne. "The code 2.0 graphical parallel programming language". In: *6th international conference on Supercomputing (ICS'92)*. Washington, DC, United States: ACM Press, 1992, pp. 167–177.

[58] D. O. Paun and M. Chechik. "Events in linear-time properties". In: *4th IEEE International Symposium on Requirements Engineering (RE'99)*. Limerick, Ireland: IEEE Computer Society, 1999, pp. 123–132.

[59] D. O. Paun and M. Chechik. ". *Formal Aspects of Computing*, vol. 14-4, 2003, pp. 342–368.

[60] F. A. Rabhi, H. Cai, and B. C. Tompsett. "A skeleton-based approach for the design and implementation of distributed virtual environments". In: *International Symposium on*

Software Engineering for Parallel and Distributed Systems (PDSE'00). Limerick, Ireland: IEEE Computer Society, 2000, pp. 1–13.

[61] W. P. Roever. "Concurrency verification - introduction to compositional and noncompositional methods". New York, NY, USA: Cambridge University Press, 2001, 798p.

[62] O. M. Santos. "Verificação formal de sistemas distribuídos modelados na gramática de grafos baseada em objetos". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2004, 89p.

[63] O. M. Santos, F. L. Dotti, and L. Ribeiro. "Verifying object-based graph grammars". In: *International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'04).* Barcelona, Spain: Elsevier Science Publishers B. V., 2004, pp. 125–136.

[64] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. "The enterprise model for developing distributed applications". *IEEE Parallel and Distributed Technology*, vol. 1-3, 1993, pp. 85–96.

[65] F. Seutter. "CEPROL: a cellular programming language". *Parallel Computing*, vol. 2-4, Dec. 1985, pp. 327–333.

[66] N. Shavit and N. Francez. "A new approach to detection of locally indicative stability". In: *13th International Colloquium on Automata, Languages and Programming (ICALP'86).* Rennes, France: Springer-Verlag, 1986, pp. 344–358.

[67] W. Shu and L. V. Kale. "Chare kernel - A runtime support system for parallel computations". *Journal of Parallel and Distributed Computing*, vol. 11-3, 1991, pp. 198–211.

[68] A. Singh, J. Schaeffer, and M. Green. "A template-based approach to the generation of distributed applications using a network of workstations". *IEEE Transactions on Parallel and Distributed Systems*, vol. 2-1, 1991, pp. 52–67.

[69] A. Sinha, L. Kale, and B. Ramkumar. "A dynamic and adaptive quiescence detection algorithm". Parallel Programming Laboratory – Department of Computer Science, University of Illinois. Urbana-Champaign, IL, USA, Tech. Rep. 93-11, 1993.

[70] S. Siu, M. D. Simone, D. Goswami, and A. Singh. "Design patterns for parallel programming". In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96).* Sunnyvale, CA, USA: CSREA Press, 1996, pp. 230–240.

[71] S. Siu and A. Singh. “Design patterns for parallel computing using a network of processors”. In: *6th International Symposium on High Performance Distributed Computing (HPDC’97)*. Portland, OR, USA: IEEE Computer Society, 1997, pp. 293–304.

[72] D. B. Skillicorn and D. Talia. “Models and languages for parallel computation”. *ACM Computing Surveys*, vol. 30-2, 1998, pp. 123–169.

[73] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. “Mpi: The complete reference”. Cambridge, MA, USA: MIT Press, 1995, 336p.

[74] G. Spezzano and D. Talia. “A high-level language for programming cellular algorithms on parallel machines”. In: *2nd Conference on Cellular Automata for Research and Industry (ACRI’96)*. Milan, Italy: Springer-Verlag, 1996, pp. 187–196.

[75] V. S. Sunderam. “PVM: a framework for parallel distributed computing”. *Concurrency, Practice and Experience*, vol. 2-4, 1990, pp. 315–340.

[76] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. “Using generative design patterns to generate parallel code for a distributed memory environment”. In: *9nd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP’03)*. San Diego, CA, USA: ACM Press, 2003, pp. 203–215.

[77] H. Tang and T. Yang. “Optimizing threaded mpi execution on smp clusters”. In: *15th international conference on Supercomputing (ICS’01)*. Sorrento, Italy: ACM Press, 2001, pp. 381–392.

[78] G. Tel. “Distributed infimum approximation”. In: *International Conference on Fundamentals of Computation Theory (FCT’87)*. Kazan, USSR: Springer-Verlag, 1987, pp. 440–447.

[79] G. Tel. “Introduction to distributed algorithms”. New York, NY, USA: Cambridge University Press, 1994, 538p.

[80] G. Tel and F. Mattern. “The derivation of distributed termination detection algorithms from garbage collection schemes”. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15-1, 1993, pp. 1–35.

[81] D. W. Walker. “The design of a standard message passing interface for distributed memory concurrent computers”. *Parallel Computing*, vol. 20-4, 1994, pp. 657–673.

[82] B. Wilkinson and M. Allen. "Parallel programming: techniques and applications using networked workstations and parallel computers". Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999, 431p.